P4T: Design and Development of P4 based Networking Testbed

By

Maryam Iftikhar

Fall-2021- MS-IT-21 363376 SEECS

Supervisor

Dr. Salman Abdul Ghafoor, Department of Electrical Engineering

A thesis submitted in partial fulfillment of the requirements for the degree of Masters of Information Technology (MS IT)

In

School of Electrical Engineering and Computer Science,

National University of Sciences and Technology (NUST),

Islamabad, Pakistan.

(Dec 2023)

THESIS ACCEPTANCE CERTIFICATE

Certified that final copy of MS/MPhil thesis entitled "Design and development of a P4 based networking testbed" written by Maryam Iftikhar, (Registration No 00000363376), of SEECS has been vetted by the undersigned, found complete in all respects as per NUST Statutes/Regulations, is free of plagiarism, errors and mistakes and is accepted as partial fulfillment for award of MS/M Phil degree. It is further certified that necessary amendments as pointed out by GEC members of the scholar have also been incorporated in the said thesis.

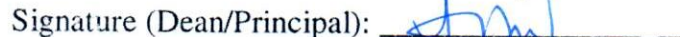
Signature:

Name of Advisor: Dr. Salman Abdul Ghafoor

Date: 22-Nov-2023

HoD/Associate Dean:

Date: 26- Dec - 2023

Signature (Dean/Principal): 

Date: 26 Dec, 2023

Online Printing Date & Time: Monday, 25 December 2023 19:20:43

Approval

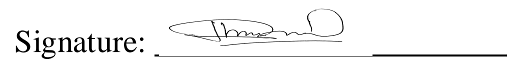
It is certified that the contents and form of the thesis entitled "Design and development of a P4 based networking testbed" submitted by Maryam Iftikhar have been found satisfactory for the requirement of the degree

Advisor : Dr. Salman Abdul Ghafoor

Signature:

Date: 22-Nov-2023

Co-Advisor: Dr. Arsalan Ahmad

Signature: 

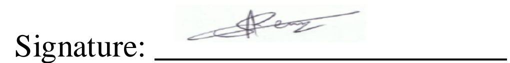
Date: 23-Nov-2023

Committee Member 1:Dr. Syed Taha Ali

Signature:

Date: 23-Nov-2023

Committee Member 2:Dr. Ahmad Salman

Signature: 

Date: 22-Nov-2023

Publish Date & Time: Thursday, 23 November 2023 , 15:21:37

Dedication

This effort is dedicated to my supportive parents, committed professors, and the invaluable assistance of SEECS administration for enabling me to pursue my education alongside my professional responsibilities. Their confidence in my talents and ongoing mentoring have been vital in molding my growth and development. I am eternally grateful for their love and the priceless life lessons they have taught me. This effort is an homage to their persistent commitment and the tremendous influence they have made on my life.

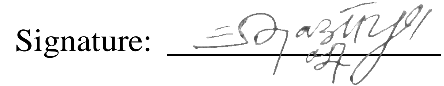
Maryam Iftikhar

Certificate of Originality

I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, nor material which to a substantial extent has been accepted for the award of any degree or diploma at NUST SEECS or at any other educational institute, except where due acknowledgement has been made in the thesis. Any contribution made to the research by others, with whom I have worked at NUST SEECS or elsewhere, is explicitly acknowledged in the thesis.

I also declare that the intellectual content of this thesis is the product of my own work, except for the assistance from others in the projects de-sign and conception or in style, presentation and linguistics which has been acknowledged.

Author Name: **Maryam Iftikhar**

Signature: 

Acknowledgements

I extend my sincere appreciation to everyone who played a vital role in the successful completion of this research endeavor. I would like to express my gratitude to Allah for granting me the strength to take on this challenge, with the intention of emphasizing His majesty.

Moreover, without the support and guidance of my advisors, Dr. Salman Abdul Ghafoor and Dr. Arsalan Ahmed, I would not have been able to successfully complete this research. Their direction and uplifting encouragement were invaluable throughout this journey. I'm also thankful for Hafiz Mati ur Rehman (Lab Incharge) whose guidance and practical assistance significantly contributed to thesis success, and for the time and expertise he generously shared.

Last but not least, I would like to acknowledge the SEECS administration for their support in pursuing my education alongside my professional responsibilities, as well as my family for providing an environment in which I could focus on developing this thesis. Their understanding and encouragement have been truly invaluable.

Table of Contents

Approval **3**
1. Chapter 01 - Introduction **13**
1.1 Motivation *13*
1.2 Revolutionizing Networking *13*
1.3 The Dynamic Architecture of SDN *14*
1.3.1 Application Plane 15
1.3.2 Control Plane 15
1.3.3 Data Plane 16
1.3.4 Proposed Tri-Layer SDN Model 16
1.4 Applications of SDN *17*
1.4.1 Integrated Storage Solutions 17
1.4.2 Empowering Visual Experiences 17
1.4.3 Streamlining Orchestration for Seamless Connectivity 17
1.4.4 Scaling New Horizons of Data Center Networks 18
1.4.5 Empowering Enterprise Connectivity 18
1.5 SDN Controllers *18*
1.6 Challenges Related to Software-Defined Networking (SDN) *19*
1.6.1 Network Security Challenges 19
1.6.2 Scalability and Performance Issues 20
1.6.3 Standardization and interoperability Challenges 20
1.6.4 Complexity and Management Issues 20
1.6.5 Challenges with Reliability and Fault Tolerance 20
1.7 Problem Statement *20*
1.8 Proposed Solution *21*
1.9 Thesis Outline *23*
2. Chapter 02 - Literature Review **24**
3. Chapter 03 - Virtualization of Network Functions **27**
3.1 Fundamentals of Network Function Virtualization (NFV) *27*
3.2 Virtual Network Functions (VNF) *27*
3.3 Framework and Services *27*
3.3.1 Infrastructure Layer 27
3.3.2 Virtualization Layer 27
3.3.3 Orchestration Layer 28
3.3.4 VNF Manager Layer 28
3.3.5 Network Services Layer 28
3.4 Advantages of Network Function Virtualization (NFV) *29*
3.4.1 Service Flexibility 29
3.4.2 Cost Efficiency 29
3.4.3 Scalability and Elasticity 30
3.4.4 Service Chaining 30
3.4.5 Network Programmability 30
3.5 Addressing the Obstacles of Network Function Virtualization (NFV) *30*
3.6 A Comparative Analysis of NFV and SDN *30*
4. Chapter 04 - Programming Protocol-Independent Packet Processors **31**
4.1 P4 Language *31*

4.1.1 Packet Parsing 31
4.1.2 Ingress Pipeline 32
4.1.3 Egress Pipeline 32
4.1.4 Packet Deparsing 32
4.2 Behavioral Model *32*
4.3 V1 Model Architecture *33*
4.3.1 Metadata Fields 34
4.3.2 P4 Code and Functionality 35
4.3.3 P4 Basic Headers 36
4.3.4 Parser 37
4.3.5 Simple Actions 38
4.3.6 Tables 39
4.4 P4Runtime *40*
4.4.1 Control Plane with P4 Runtime API *41*
4.5 P4 Advantages and Specifications *43*
5. Chapter 05 - Open Network Operating System 44
5.1 ONOS Specifications *44*
5.1.1 Centralized Management 44
5.1.2 Northbound and Southbound Interfaces 44
5.1.3 Flow Management and Traffic Engineering 44
5.1.4 Network Virtualization 44
5.1.5 Application Ecosystem 44
5.1.6 Flexibility and Programmability 45
5.1.7 Scalability and High Availability 45
5.1.8 Open-Source Community 45
5.2 Design Principles of ONOS *45*
5.2.1 Code modularity 45
5.2.2 Configurable Features 45
5.2.3 Protocol Independence 45
5.2.4 Protocol Awareness Module 46
5.2.5 System Core 46
5.2.6 Applications 46
5.3 System Components of ONOS *46*
5.3.1 Device Subsystem 47
5.3.2 Host Subsystem 47
5.3.3 Topology Subsystem 47
5.3.4 Path Service 47
5.3.5 Flow Rule Subsystem 47
5.4 ONOS Subsytem Structure *47*
5.5 Network-state Representation *48*
5.5.1 Outbound Packet 49
5.5.2 Inbound Packet 49
5.6 Device Subsystem in ONOS *49*
5.6.1 Device Manager 49
5.6.2 Device Providers 49
5.6.3 Device Store 50

5.7 Device Driver Subsystem 51
6. Chapter 06 - Environmental Setup 52
6.1 System Specifications 52
6.1.1 Manual Installation 52
6.2 ONOS Installation using Docker 53
7. Chapter 07 - Experimental Setup 56
7.1 Operating System Stratum 56
7.2 Mininet Topology 57
Chapter 08 – Results 59
8 Program leaf1 using P4Runtime 59
8.1 Static NDP Table Entries 59
8.2 P4Runtime Table Entries 60
8.3 YANG 61
8.3.1 Open Configuration 63
8.3.2 Global Network Management Interface 63
8.4 ONOS as a Control Plane 64
8.4.1 Enable Packet I/O and Double-Check Link Discovery 64
8.4.2 L2 Bridging and Host Discovery 69
8.4.3 Topology Discovery 71
8.4.4 Performance Analysis 72
Chapter 09 - Future Work 74
Chapter 10 - References 75

List of Figures

Figure 1: Ecosystem of SDN **14**
Figure 2: Three-layered SDN Framework **15**
Figure 3: Proposed SDN Tri-Layer Architecture **16**
Figure 4: Proposed Architecture **21**
Figure 5: VNF Framework **28**
Figure 6: End to End Network Service **29**
Figure 7: P4 Architecture **31**
Figure 8: P4 Packet Processing **32**
Figure 9: P4 V1 Model **33**
Figure 10: Match Action Data Flow **40**
Figure 11: Local Control Plane with P4 Runtime API **41**
Figure 12: Remote Control Plane with P4 Runtime API **42**
Figure 13: P4 Runtime Data Flow **43**
Figure 14: ONOS Stack **46**
Figure 15: ONOS Components **47**
Figure 16: OF Provider **50**
Figure 17: Stratum Controller **56**
Figure 18: Mininet Topology **58**
Figure 19: Static NDP Entries **60**
Figure 20: Rules Insertion in P4Runtime Shell **60**
Figure 21: Ping h1 to h2 .... **61**
Figure 22: Detection of Devices in ONOS CLI **67**
Figure 23: Links on Devices **68**
Figure 24: Topology Discovery **71**

List of Graphs

1	RTT Analysis for h1 to h2 Ping	72
2	TP Analysis for h1 to h2 Ping	73

Abstract

In recent years, the networking domain has experienced a significant transition, with Software-Defined Networking (SDN) emerging as an architectural innovation that provides network operators with unparalleled programmability and administrative flexibility. By decoupling network control from forwarding operations, simplifying administration, and accelerating network advancement, SDN facilitates the creation of new networking abstractions. Because of the expanding demand for fault tolerance and scalability in SDN systems, the importance of improving and refining SDN operating systems with distributed architectures is increasingly growing. However, without access to expensive testbeds, successful implementation of such systems can be difficult. Various SDN development approaches depend on full system virtualization or resource-intensive containers, which increases complexity and costs while decreasing user-friendliness. In response to these issues, this paper introduces P4T (a P4-based networking testbed) to address the challenges of creating reliable and scalable SDN operating systems to meet the demands of modern network development. The proposed testbed leverages the power of P4-based BMv2 switches and Mininet and provides a comprehensive but simplified platform to simulate SDN-based network topology. With this testbed, several BMv2 switch instances can be developed inside the same virtual environment, seamlessly orchestrating forwarding behavior. Moreover, P4T empowers users to create custom topology, easily integrate an ONOS controller for centralized control, and implement a wide range of P4-based networking applications. This facilitates detailed evaluation of network programmability and performance, thereby advancing the field of SDN network simulation.

1. Chapter 01 - Introduction

1.1 Motivation

With the emergence of Software-Defined Networking (SDN), the landscape of conventional networks has experienced substantial modifications. SDN presents three separate layers: infrastructure or physical layer, control layer, and application layer. The widely used Southbound APIs, notably the OpenFlow [33] Protocol, bridge the gap between the physical and control layers. Northbound APIs, similarly, act as the interface between the control layer and the application layer. The controller is the nucleus of the SDN network architecture, commonly compared to the network orchestrator, with the task of orchestrating and managing network traffic throughout network devices located at the physical layer. The main principle underlying this technique is the separation of network control logic (control plane) [10] from forwarding hardware (data plane), resulting in seamless integration. This technique tackles various constraints inherent in traditional networks, providing improved dependability, security, and flexibility.

However, the use of SDN has few challenges that must be carefully considered. One of them is the time necessary to adopt new protocols [25] or increase current functions that need a large time investment. To become an official standard, the protocol must be implemented in ASICs (Application-Specific Integrated Circuits), which might take many years. In the context of corporate networks, it is essential to consider their dependency on network chip makers. The advent of the P4 is notable joint efforts to address the aforementioned difficulties.

P4 (Programming Protocol-Independent Packet Processors) is created for network devices to provide versatility and efficiency, making it appropriate for a variety of purposes including NICs, routers, switches both hardware and software and other applications. P4 language is well known for its ability to enable the creation of a network stack in switching hardware. It opens up new possibilities for increased flexibility, removing the need to invest in new switches because P4 software [25] allows for simple updates.

1.2 Revolutionizing Networking

The efficacy of some technologies, for instance cloud computing, has a significant impact on network infrastructure performance. Still, the slow development of flexible IT structures creates a number of issues. To overcome these difficulties, SDN technology offers new features to the entire network structure. Administrators can encapsulate the network's fundamental framework in order to support applications including network features. The [35] Open Networking Foundation (ONF), a non-profit group, is responsible for the marketing, development, and standardization of SDN. It is characterized features like the separation of the control and data planes, as well as the ability to program within the control plane. This architectural decoupling boosts network performance and extends control to many networking levels, from packet forwarding to data link connections. It offers real-time centralized network control, policy

implementation, and dynamic network management. The expanding need for SDN has been driven by the growing demand for dynamic [36] network management, rapid service creation, and efficient resource utilization. It enables network administrators to control and manage networks programmatically, resulting in effective resource allocation, quicker service delivery, and flexibility [11] to changing network circumstances. SDN replaces traditional network topologies with centralized control planes, providing a unified view of the network and enabling network virtualization for better resource utilization, scalability, and multi-tenancy.

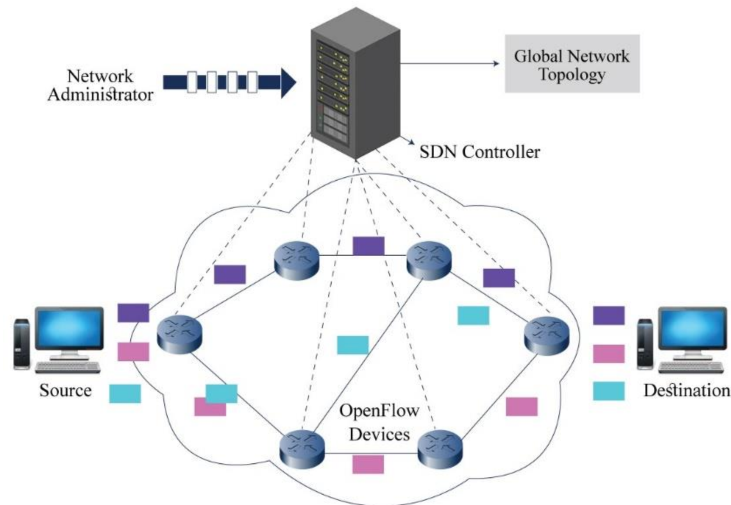


Figure 1: Ecosystem of SDN

SDN transforms network programmability by employing APIs, allowing administrators to dynamically configure devices in response to user requests. This gives network functioning a new degree of agility and resilience. The diagram in Figure 1 depicts network programmability in the context of SDN ecosystem. SDN OF data plane devices are configured by network administrators using the SDN controller. Rules are distributed to linked switches by using the centralized SDN controller. Scheduling, rerouting, and load balancing of traffic across available connections are all configuration activities that must be adapted to user expectations and performance optimization requirements.

1.3 The Dynamic Architecture of SDN

One significant component of SDN design is the separation of control and data forwarding processes, often known as dis-aggregation. [15] This architectural divide provides a substantial advantage by giving apps improved access to network status information straight from the controller. This extends beyond the capabilities of conventional networks, where network awareness is restricted. Figure 2 depicts a network administrator participating in programming operations via the SDN controller, especially targeting SDN OF data plane devices. The administrator's operations include scheduling, rerouting, and dividing traffic among available connections based on user demand and performance requirements, with the ultimate goal of

achieving optimal network performance. To be more precise, the architecture of SDN comprises three distinct layers: the application plane, the control plane, and the data plane

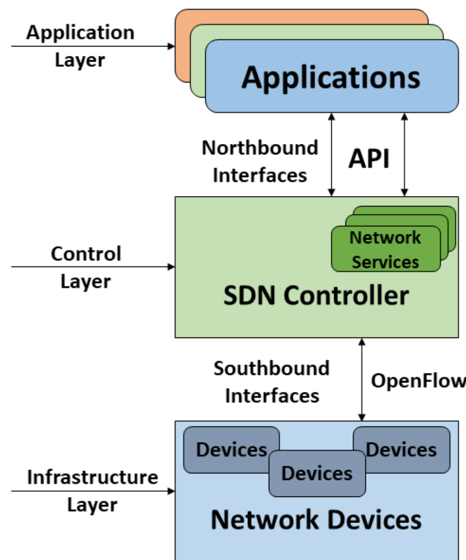


Figure 2: Three-layered SDN

1.3.1 Application Plane

The application plane or layer is responsible for delivering different network services and applications. It acts as a bridge between consumers or applications and the network infrastructure. Users can interact with the SDN controller [12] via the application layer, making requests or defining network policies. This layer provides network programmability and customization depending on individual application needs.

1.3.2 Control Plane

The control layer holds paramount importance and serves as the central point for network control and monitoring the whole network's behavior and operation. It contains the SDN controller, which acts as the network's brain, making decisions and controlling network traffic flow. The SDN controller connects with the application layer [12] within the control layer to accept instructions, rules, and network needs from applications or users. It uses this information to dynamically configure network equipment like switches and routers, as well as to specify how data packets should be delivered.

Administrators can use the control layer to respond quickly to changing network circumstances, adapt to new requirements, and optimize network performance. The control layer's programmability enables efficient traffic routing, load balancing, and network-wide policy enforcement, resulting in improved network dependability and performance.

1.3.3 Data Plane

The data plane which is also known as infrastructure layer in the SDN architecture, which forms the foundation of the network infrastructure. This layer [12] includes physical network devices including switches, routers. The infrastructure layer is in charge of actually transmitting data packets based on orders from the control layer.

Network devices at the infrastructure layer are programmable and capable of implementing the policies and settings provided by the SDN controller. These devices connect with the controller using standardized protocols like the OF Protocol, which allows for smooth integration and interoperability inside the SDN architecture. It acts as the execution layer, carrying out the control layer's commands and policies. Because the control and data planes are separated, the infrastructure layer may concentrate on quick and reliable packet forwarding while the control layer performs high-level decision-making and network administration responsibilities.

1.3.4 Proposed Tri-Layer SDN Model

Considering network flexibility and meeting the ever-changing demands of current applications, we offer a strong tri-layer framework represented in Figure 3, to remodel the existing SDN network architecture. Our proposed framework hosts the P4 target program that uses BMv2 switch to define the behavior of the data plane. P4 Runtime API communicates with the control plane via gRPC. Within the network architecture, data plane is configured through control plane for efficient packet processing, table management, and packet IN/OUT operations.

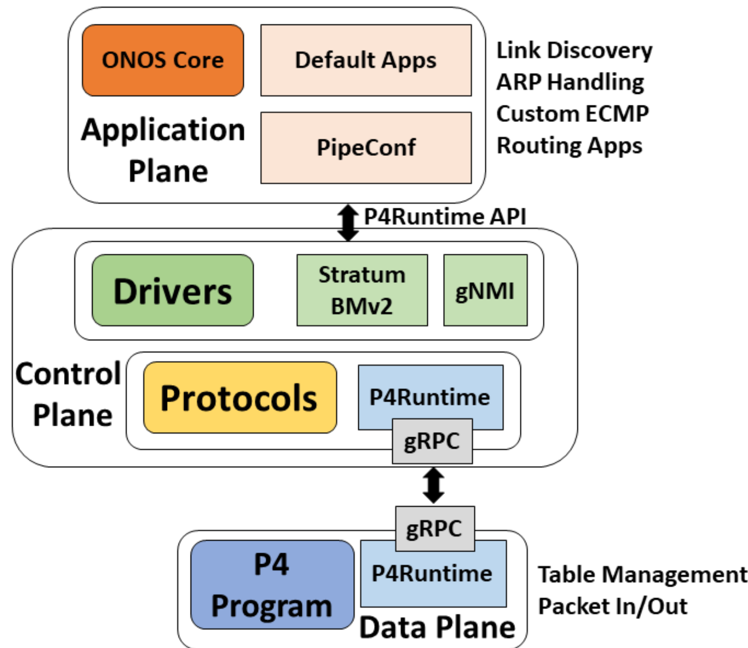


Figure 3: Proposed SDN Tri-Layer Architecture

The control plane uses an SDN controller to orchestrate rules and commands throughout the network. The ONOS SDN controller was chosen as the master conductor for creating routing tables and directing network traffic using RESTful APIs. We combine different elements within this control plane, such as Stratum BMv2 switches, GNMI, drivers, and protocols to ensure effective network administration and control. Protocols include: link discovery protocol to discover links between switches, routing protocols to exchange routing information among BGP, RIP and OSPF. The drivers ease interaction between the control plane and P4 Runtime module.

The application plane is at the top of our architecture where networking innovations comes to life. It enables user to discover a variety of networking program here, including L2 and L3 forwarding switches with customized and dynamic solutions. It uses HTTP, FTP and SMTP protocols for browsing, transferring files and sending emails. Using our testbed the user will be able to write P4 based networking applications in addition to traditional networking applications.

1.4 Applications of SDN

This section focuses on a range of real-world applications that have successfully deployed SDN, proving its practical flexibility and effectiveness.

1.4.1 Integrated Storage Solutions

Converged storage is a unified storage system that integrates computer resources and storage into a single framework. It provides a flexible architecture for designing storage-centric, server-centric, or hybrid applications. Many important data services have embraced programmable technology for their operations in the domain of SDN. Edgenet1, [41] e.g. a universally distributed edge cloud, has created a solution based on the Programmable Flow SDN Ecosystem. The OF controller, in conjunction with robust and efficient switching, is used in this environment to provide smooth and optimal network performance.

1.4.2 Empowering Visual Experiences

Ribbon Communications, doing business as Sonus Networks [41], offers a new SDN solution for video and collaboration applications in the real-time communications realm. They use a unified communications and control strategy, combining Juniper's network virtualization platform with their session border controllers. This integration considerably improves communication session capacity. The use of SDN, which assures the preservation of Quality of Service (QoS), is an exciting component of their strategy.

1.4.3 Streamlining Orchestration for Seamless Connectivity

SDN and network functions virtualization (NFV) have both made inroads into the world of mobile network operators (MNOs) [41], with many suppliers embracing them in order to build

highly resilient networks with optimal resource utilization and dynamic provisioning. Vendors can drastically cut lead times and provide better service delivery by combining SDN and NFV.

1.4.4 Scaling New Horizons of Data Center Networks

SDN systems are a possible replacement to data center networks' traditional wired architecture. University of Illinois researchers [41] have undertaken considerable research on the adoption of SDN switches to evaluate the performance of data center networks. They accomplished effective scalability of bandwidth across servers while minimizing hardware expenses by exploiting SDN. Hundreds of Pica84 [41] switches with hundreds of ports were put in their tests, guaranteeing high-speed access and allowing load balancing and bandwidth scaling to be controlled properly.

1.4.5 Empowering Enterprise Connectivity

SDN suppliers have adopted software-driven networking solutions for enterprises, which have largely replaced the old reliance on physical networking [41] infrastructure. This paradigm change to software-driven networking marks a huge improvement in corporate connection, providing a more dynamic and scalable solution than traditional physical networks.

1.5 SDN Controllers

As the primary control point of SDN, the SDN Controller serves as the network's ‘brain’. Using southbound [40] APIs, it manages the flow of traffic between the network switches and routers underneath and the applications and corporate sections above. OF and Open Virtual Switch Database (OVSDB) are two protocols that controller use to establish connection with switches and routers. It has various modules effortlessly added into the system to execute system activities. These operations include device inventory [40], device capabilities assessment, network statistics collection, and monitoring functions. The platform's flexibility enables for the insertion of additional modules to expand functionality and support advanced features.

NOX was the first SDN controller, created by Nicira [40] Networks in conjunction with OpenFlow. It is worth mentioning that ONIX, the basis for the Nicira/VMware controller built by a collaboration of NTT, Google, and Nicira Networks, is not an open-source solution. Here is a list of open source SDN controllers:

Open Network Operating System (ONOS): A scalable and distributed controller with a reactive programming style for high-performance networks.

OpenDaylight: It is a highly extendable and modular Java controller platform that supports a wide range of protocols and interfaces.

Ryu: It is a Python-based component-based software-defined networking architecture that includes a library for developing SDN applications.

Faucet: An open source SDN controller for business and college networks that is written in Python and works with OpenFlow switches.

Floodlight: An Apache-licensed Java controller with a wide range of capabilities and support for the OpenFlow protocol.

Trellis: An ONOS-based controller intended primarily for multi-tenant and scalable networks, notably in cloud and edge contexts.

OpenContrail: An open-source software-defined networking controller that focuses on network virtualization and overlay features.

Beacon: A Java-based controller that provides a lightweight and adaptable framework for SDN research and development.

Maestro: A Telefonica open source SDN controller designed to coordinate and manage network services in multi-domain systems.

FBOSS (Facebook Open Switching System): It is not a standalone controller, FBOSS is an open-source software stack used by Facebook for its data center networks that contains a network operating system and control plane components.

Although there are numerous open source controllers but in this research, ONOS controller is integrated with P4 testbed because of its adaptability in supporting a wide range of applications and services. It provides a wide range of functionality and features customized to varied network requirements, whether in data center networks, service provider settings, or edge computing situations. This adaptability enables us to take use of ONOS's special features that are relevant to our study subject. It has a reactive programming paradigm that allows dynamic network adaptation by reacting in real-time to network events and changes.

The emphasis on intent-based networking in ONOS is also a significant benefit. ONOS simplifies network administration and automates the translation of intentions into low-level network settings by abstracting network regulations and needs into higher-level intents.

1.6 Challenges Related to Software-Defined Networking (SDN)

SDN has many advantages, but it also has certain drawbacks. Some of the challenges include:

1.6.1 Network Security Challenges

Network security is a key difficulty in SDN. SDN controller vulnerabilities might expose the entire network to possible attackers. Because the control plane in SDN is centralized, it becomes a single point of failure, necessitating the implementation of rigorous [27] security measures.

Unauthorized access and control plane assaults are other issues that must be addressed to preserve network integrity and secrecy.

1.6.2 Scalability and Performance Issues

Scalability is a significant issue in managing large-scale SDN implementations. The proper management of the network's size becomes crucial as the number of switches and flows rises. The maintenance and processing of flow tables becomes more complicated, [27] necessitating the use of optimization techniques to guarantee effective forwarding and control. Furthermore, latency and delay concerns in the control plane must be minimized in order to ensure responsive network performance.

1.6.3 Standardization and interoperability Challenges

They pose difficulties in SDN implementations. Incompatibility difficulties between SDN controllers and network devices might arise due to a lack of universal standards and vendor-specific implementations. Integrating traditional network infrastructure [27] with SDN technologies necessitates careful planning in order to provide seamless compatibility and simple migration pathways.

1.6.4 Complexity and Management Issues

The intrinsic complexity of SDN poses management issues. Configuration and policy management may be difficult to handle, particularly in large-scale systems. The complexity is increased further by orchestrating [27] and coordinating across many domains and technologies. To successfully manage the network and maintain smooth operations, monitoring and debugging distributed SDN systems necessitate specialized tools and methodologies.

1.6.5 Challenges with Reliability and Fault Tolerance

Ensuring high availability and fault tolerance in SDN is critical. Reducing single points of failure in SDN designs and ensuring control plane resilience are critical issues. Implementing fault detection [27] and recovery procedures becomes critical, especially in dynamic networks with changing network architecture and traffic patterns.

1.7 Problem Statement

The lack of a specialized testbed for assessing P4-based solutions hinders not only the thorough modelling of SDN networks, but also the study of future network paradigms. Existing tools, while useful, sometimes lack the flexibility necessary to build custom topologies and deploy a diverse set of P4-based networking applications. This constraint not only limits our capacity to completely examine network programmability and performance, but it also limits innovation in the fast developing field of software-defined networking.

1.8 Proposed Solution

To address the aforementioned issue, the research has developed a specialized P4-based testbed within Mininet, hence improving the capabilities of SDN network simulation. The proposed testbed uses P4-based bmv2 switches to correctly replicate SDN-based networking topologies, offering an optimal environment for testing and experimentation. Users are able to design bespoke topologies, effortlessly incorporate an ONOS controller for centralized control, and deploy a wide range of P4-based networking applications.

Our proposed architecture, named P4T is created by using P4-based BMv2 switches to allow users to make custom network topology. Our novel approach leverages ONOS controller applied rules to configure the P4 switch pipeline as shown in Figure 4. At the core of our architecture lies the ONOS controller, serving as the central nervous system of the SDN framework to control and orchestrate network monitoring and configuration. With P4 Runtime support, it provides a variety of useful functions, including device detection, P4 pipeline provisioning, table operation for match action execution (through current ONOS APIs like Intent, Flow Objective, Flow Rule), controller processed packet handling (packet IN/OUT), and the ability to read counters.

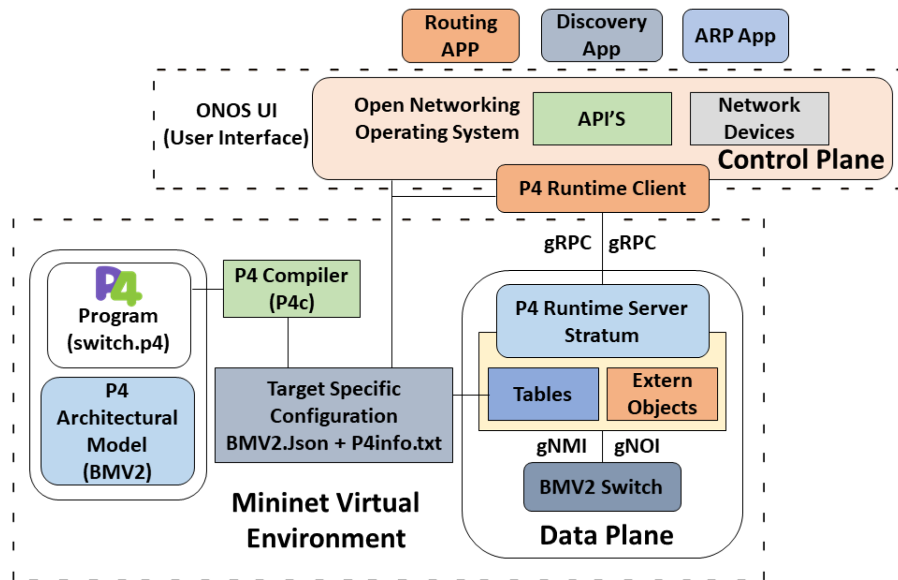


Figure 4: Proposed Architecture

The BMv2 Switch forming the foundation of our architecture is at the bottom layer and acts as the data plane for packet processing. It runs the forwarding logic of the P4 program to process incoming and outgoing network traffic. To alter the behavior of data plane, we applied pipeline modification rules controlled by ONOS. Interfaces such as gNOI and gNMI permit seamless interaction by facilitating easy communication between various layers.

We used P4C (P4 compiler) to transfer our P4 source code into the desired machine readable format. It creates a data plane configuration to execute the forwarding logic and to provide a control plane interface for managing the state of data plane objects. Extensive testing is done on P4 BMv2, as well as on Stratum, that provide access to various SDN access points, including P4 Runtime, to validate our concept. Stratum provides a Docker image built for executing network simulations using Mininet, with stratum BMv2 as the default switch. This Docker image enables testing and demonstrating features like P4 Runtime and gNMI, within a simulated network environment. We utilized ONOS' control plane capabilities to evaluate our testbed. Our Bmv2 switches are controlled and programmed using a thrift based interface, which is supplemented by a Command Line Interface (CLI). We used P4 Runtime, a standardized, open, and hardware agnostic protocol that allowed real time P4 forwarding within P4 forwarding plane.

In the data plane, we used our target BMv2 switch to create, test and debug the P4 data plane for later usage at the SDN controller. Using the conventional compilation method, the P4 source program is first installed and then compiled in the network switches through p4c, generates target specific two configuration files, P4info.txt and BMv2.Json. P4info.txt contains schema of P4info instance modified for our P4 program which provides a detailed description of the match-action tables accessible via the P4 Runtime API while the BMv2.json contains the key settings required for seamless integration and operation within the SDN framework and is used to describe specific flow table entries along with the properties such as its location, matching criteria or method, action name and parameters. All of these characteristics are dependent on the flow tables, matching fields, and actions that we have configured in our P4 code.

When BMv2 starts, it imports the JSON file created by the compilation of the P4 code, parses it and then loads the resulting configuration into the flow table of switch. To allow P4 programmability and to connect data plane with control plane, communication is established via P4 Runtime. Traditional systems do not allow for post-deployment interaction with the P4 switch, such as altering table entries i.e. adding, deleting, accessing or querying counters.

P4 Runtime overcomes this issue by the use of a client-server paradigm. A client component incorporated into the control plane and communicates with a server. This interface allows a variety of operations, including P4 program loading, pipeline information change and retrieval, and packet transmission and reception. Furthermore, it enables the application and installation of IPv4 forwarding rules within the switches that are suited to the network architecture.

ONOS controller is used to host the control plane in our P4T architecture. It allows developers to use both existing pipeline agnostic apps operating on Flow Objectives and develop new apps customized for specific data planes. Deploying an SDN controller in a P4-based network with a P4 Runtime interface unlocks the full potential of P4 functionality. Stratum is actively working on enabling a P4 Runtime interface for P4-based devices integrated with Network Operating Systems (NOS). Intelligent applications can be constructed above the ONOS controller to govern

its network activities, establishing an Application plane. The ONOS build-in topology discovery process is accomplished via following synchronized stages:

First, at each switch port, Link Layer Discovery Protocol (LLDP) packets are injected and captured to perform link discovery. LLDP is commonly used in OF networks for identifying and comprehending inter-switch interactions. Next, the system then proceeds to discover hosts by intercepting Address Resolution Protocol (ARP) packets originating from data plane ports. ARP is critical in mapping IP addresses to physical MAC addresses, allowing the network to determine the existence and location of individual hosts. Following that, table entries are systematically placed to simplify data packet routing between hosts using the ECMP principles. These table entries provide the foundation for providing effective and optimized data packet forwarding over the network, supporting smooth communication between network endpoints.

1.9 Thesis Outline

The thesis's remaining sections are organized as follows: Chapter 2 covers the literature review. Chapter 3 changes the focus to Network Functions Virtualization (NFV), which explains the notion of virtualizing network functions and the benefits it provides. It compares NFV and SDN, demonstrating their similarities and differences as well as how they may be utilized together in networking systems. Chapter 4 presents the P4 programming language, which is used to specify the behavior of network devices such as switches and routers. This section delves into P4's features and applications in network programmability. Chapter 5 of the thesis delves into ONOS, an SDN controller and includes its features, architecture, and functioning. Chapter 6 focuses on how this virtual environmental setup has been created. Chapter 7 has experimental setup that explains how SDN and P4 may be used in tandem to build and implement network protocols in a programmable and flexible way. Chapter 8 offers the results and findings, summarizing the key findings, contributions, and limitations. Chapter 9 provides prospective avenues for future study. However, Chapter 10 contains all the references.

2. Chapter 02 - Literature Review

Network topology primarily depended on proprietary devices, which came with high administration expenses which were frequently difficult to configure and maintain effectively. SDN resulted in a paradigm [9] by decoupling [24] the control and data planes to enable customization. However, it uncovers potential weaknesses like lack of support for deploying custom topology and diverse implementation of P4-based networking applications. The literature presents that mainly focuses on virtualization in programmable forwarding planes. In the field of P4-based networking testbeds, we found a significant milestone in [1]. This study contributes significantly to the field of P4 network programmability. First and foremost, it establishes the international P4 Experimental Networks (i-P4EN), providing a collaborative framework among research institutes worldwide. This architecture promotes the sharing of distributed P4 resources across worldwide research and education networks, as well as the start-up of international research partnership programs. Furthermore, the research solves a basic difficulty in P4-based networks: multi-tenancy. It adds a dynamic tenant pipeline configuration, allowing multiple tenants to share P4 switches without interruption. The adoption of a Role-based Shared Table mechanism (P4MT) is critical to achieve this goal. This approach guarantees control plane and data plane isolation, which helps to improve the security and efficiency of multi-tenant P4 networks. It excels at explaining the principles and offers for the implementation of P4MT across both software (BMv2) and hardware (Tofino-based) switches. However, the lack of open access to the code restricts repeatability and overall effect, leaving openings for further improvements. In [2], author investigated Active Queue Management (AQM) via the prism of the P4 and DPDK-based compilation. They have used several AQM approaches, both known and newer, such as RED, CoDel, and PIE, to solve the pressing issue of buffer-bloat in current access networks. Their main contribution is novel AQM assessment methodology, which allows for the practical evaluation of AQM algorithms on a testbed that simulates actual traffic situations. This framework allows for the execution and evaluation of AQM algorithms such as RED and PIE on a testbed that replicates realistic traffic scenarios. The paper [2] demonstrates three separate situations in the demo: a reference FIFO, the PIE AQM, and the RED AQM. These scenarios are tested on a testbed comprised of two computers outfitted with NICs and Docker containers that simulate traffic sources and sinks. The produced test traffic consists mostly of responsive TCP flows with varying numbers of active flows and adjustable congestion management techniques. The AQM assessment framework, which utilizes the T4P4S compiler and software switch and extends the P4-16 v1model architecture to reveal essential queue state information fields, is the paper's core innovation. However, it is vital to highlight that this [2] testbed is fixed and cannot accept varied P4 situations and applications, limiting its adaptability for larger AQM analyses. This study stands out by offering a realistic tool for AQM [2] assessment in real-world traffic situations, answering the critical need for a thorough technique for evaluating AQM algorithms. While the study exhibits excellent originality and execution, it may benefit from a more thorough comparison analysis with existing AQM assessment systems or testbeds. A more in-depth

examination of the framework's scalability and application to diverse high-speed network traffic circumstances will increase its use. Despite this, the suggested approach has enormous promise for furthering the evaluation of AQM algorithms and their influence on network performance. We resorted to the pages of [3] in our search for network automation. This [3] proposes P4click, a Next-Generation SDN automation platform that uses the P4 programming language to overcome the issues of modular data plane pipeline creation. With the introduction of P4, developer's encountered restrictions in sharing P4 logic across different data plane pipelines, limiting feature flexibility and reusability. Recognizing SDN developments and OF's importance in managing vendor-independent SDN forwarding devices, the authors noted a fundamental challenge: sharing modular P4 logic for specific models. In response, P4click, their [3] solution, presents an automation tool for streamlining data plane pipelines and simplifying network settings. This [3] study is consistent with current developments in SDN and P4-based networking, emphasizing the need of modularity and automation in handling complex data and control plane operations. It solves a major issue in P4-based network programmability, where existing techniques lack the ability to select and integrate certain characteristics into data plane pipelines. P4click [3] is implemented using a data plane repository that comprises feature modules represented by P4 code and configuration files. These modules can be retrieved from the repository individually, processed, and compiled to form a unified pipeline for a certain model and target. A toolset for deploying control and data plane applications is also included in the platform. [4] handled SDN and P4-based programmable switches in a way to improve the programmability of P4, allowing for sophisticated monitoring tasks such as heavy-hitter identification, flow cardinality estimate, network traffic entropy measurement, and volumetric DDoS assault detection. In comparison to prior research that focused on P4 modularity and automation, they provides hands-on methods to increase network visibility and anomaly detection, hence offering a full framework for SDN and P4-based network management. However, this framework is intended for a single switch, lacks custom topology capabilities, and is primarily used for testing and monitoring applications. Article [7] contribution is PISA's micro-architecture efficiency, with a special emphasis on match tables and programmable packet schedulers, which are main components of the PISA architecture. P4 describes how packets are processed by a programmable data plane, which ranges from ASICs to CPUs that implement PISA. While ASIC processing flexibility is relatively limited, and CPU performance for networking activities lags behind, current attempts have concentrated on implementing PISA on FPGAs. Major performance limits are exposed in executing specific PISA blocks on existing FPGA [7] designs using a mix of theoretical analysis and actual experimentation. These constraints are caused by route congestion, cable delays, and resource use. They identified certain network applications that are well suited to FPGA capabilities and offers architectural changes to improve FPGA performance, not just inside the networking domain but perhaps beyond. [5] Introduces a load balancer based on P4 that does not require a specialized controller. This load balancer may distribute loads independently while still requiring controller help for server health monitoring. In establishing of distributed SDN operating systems and applications

this [6] emphasizes the significance of fault tolerance and scalability while addressing limitations in existing methods. Many current approaches involve the creation of virtual testbeds via full machine virtualization or heavyweight containers, resulting in increased complexity and decreased scalability. To address these constraints, the paper proposes a more efficient strategy that makes use of Mininet's cluster mode, allowing the deployment of lightweight containers in virtual testbeds on a single computer, ad hoc clusters, or dedicated hardware testbeds. This strategy creates a flexible and scalable framework for constructing distributed SDN systems and application software by combining an open-source distributed network operating system like ONOS. While [6] allows for flexible deployment on a variety of hardware configurations but there's no specific technique mentioned of P4 application support, instead emphasizing on Mininet and ONOS as open-source components. To bridge the gap between fixed-pipeline traditional and programmable switching devices [8] introducing an open-source OS, Stratum with P4 integration to enable control of switches and their forwarding behavior. They emphasized on Stratum support for Broadcom XGS range fixed function switches to enable P4 flexibility in conventional hardware. For evaluating P4 program and network behavior this [8] introduces TestVectors which include, a spine and leaf fabric network comprised of Stratum supported white box from several vendors. While [8] paper does not specifically address custom topology, the programmability of Stratum suggests that users can build new topologies. In previous articles we didn't find user-friendly solution that can be used as P4 testbed for simulations and to measure SDN performance metrics. The main novelty of our work as compared to previous and existing is its open source nature. Moreover, it supports custom topology and multiple switches, giving a more complete and adaptable platform for network experimentation and validation that goes beyond the confines of BMv2 switches. Furthermore, our testbed includes a controller, allowing users to attach and assess P4 applications within regulated network contexts, significantly enhancing its value. This adaptability improves our testbed's realism and applicability, making it a great asset for investigating P4-based networking solutions in a larger range of real-world applications.

3. Chapter 03 - Virtualization of Network Functions

3.1 Fundamentals of Network Function Virtualization (NFV)

Network Function Virtualization (NFV) is a revolutionary networking concept. It involves decoupling network services from particular hardware devices and operating them as software in virtual machines (VMs). These virtual network functions (VNFs) include virtualized routers, firewalls, load balancers, WAN accelerators, intrusion detection and prevention systems, VPN gateways, and other capabilities. NFV uses virtualized networking features to provide hardware that is independent of infrastructure, allowing computation and network operations to deployed on non-proprietary hardware such as x86 [27] servers. It allows VMs to utilize resources efficiently by dynamically scaling and making most of any unused capacity on a single server.

3.2 Virtual Network Functions (VNF)

The aim to reduce operating and capital costs while accelerating the adoption of innovative network features motivated the development of network virtualization and VNFs. VNFs are in charge of certain network operations like firewalls and are essential components of [27] NFV. Individual VNFs or a mix of them can be used to construct a completely virtualized environment. When multiple VMs are deployed on a single hardware box, the entire machine's resources are consumed. Virtualization reduces network costs while increasing scalability and diversity by eliminating the need for costly hardware components and replaces them entirely with software.

3.3 Framework and Services

The NFV provides a conceptual framework for the creation and deployment of virtualized network functions. It describes the major components and their [21] interdependence within the NFV architecture. While each NFV implementation is unique, the framework usually comprises of the three following layers:

3.3.1 Infrastructure Layer

The infrastructure layer [27] is the NFV framework's basis and contains the physical resources necessary to support virtualization. Physical servers, storage systems, networking devices and hardware components are all included. These resources enable the hosting [21] and operation of VMs and VNFs by providing the appropriate processing, storage, and networking capabilities.

3.3.2 Virtualization Layer

The virtualization layer is responsible of abstracting the underlying hardware resources and enabling virtualization. It is made up of hypervisors [27] or virtual machine monitors (VMMs) that allow for the creation, administration, and separation of numerous VMs on a single physical

server. The hypervisors enable VNFs to operate on virtual machines, ensuring that hardware resources are used efficiently.

3.3.3 Orchestration Layer

The orchestration layer manages and automates the lifespan of VNFs and their interconnections. The NFV orchestrator organizes the deployment, scaling, and chaining of VNFs to provide specified network services. Resource allocation, [27] performance management, and service lifecycle management are also handled by the orchestration layer. It communicates with the infrastructure layer in order to supply and manage the resources required for VNF execution.

3.3.4 VNF Manager Layer

The VNF Manager layer manages the lifespan of individual VNFs. It does VNF instantiation, scaling, monitoring, healing, and termination. The VNF Manager [14] interfaces with the orchestration layer to receive VNF deployment and management instructions. It guarantees that each VNF runs properly and efficiently, and it communicates with the infrastructure layer to assign resources for VNF execution.

3.3.5 Network Services Layer

The network services layer is made up of VNFs that provide specialized network services. It contains numerous VNFs: virtual routers, [14] firewalls, load balancers, WAN accelerators, and other software-implemented network services. Each VNF serves a distinct network function [21] and may be deployed, maintained, and expanded separately inside an NFV environment. The NFV layers can be represented as a vertical stack in Figure 5, where each layer builds upon the one below it. At the bottom is the infrastructure layer, followed by the virtualization layer, orchestration layer, VNF Manager Layer, [27] and the network services layer at the top.

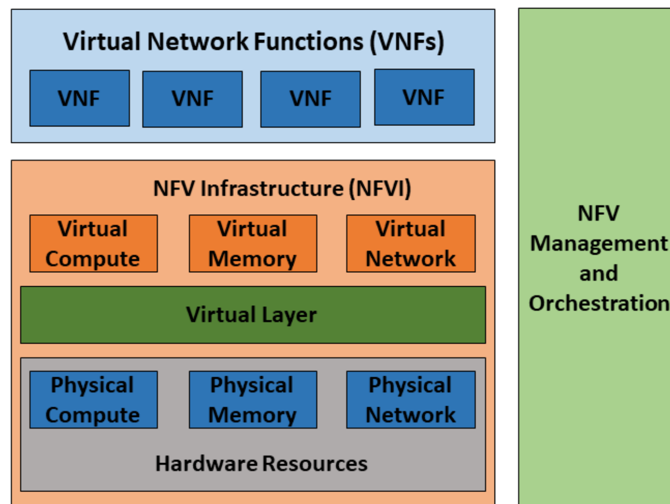


Figure 5: VNF Framework

An NF Forwarding Graph [23] is made up of NF nodes connected by logical links that might be unidirectional, bidirectional, multicast, and/or broadcast. An easy instance of a forwarding graph is a sequential succession of network services. A smartphone and a wireless network are two components throughout the entire network service. It is important to emphasize that the realm of NFV operations is confined to operator-owned resources hence power may be exercised within particular domains, excluding devices: mobile phones that fall outside of this scope.

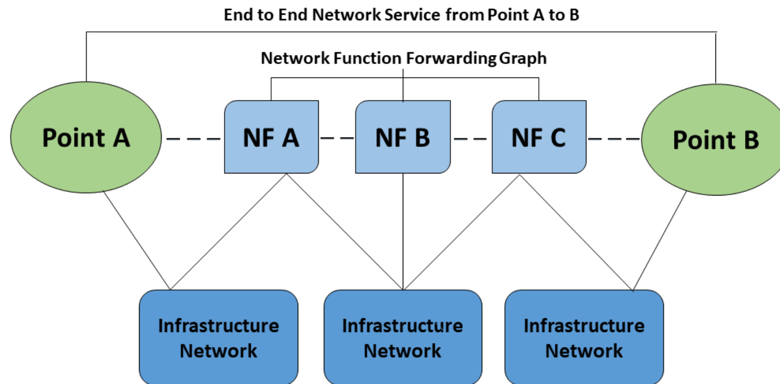


Figure 6: End to End Network Service

Figure 6 displays a complete network service. The exterior end-to-end network service is represented by End Point A, the inside NF Forwarding Graph, and End Point B. The inner NF Forwarding Graph is represented by network functions NF1, NF2, and NF3. These are linked together through logical linkages provided by the Infrastructure Network 2. The network functions are connected to the end nodes (A and B) via network infrastructure, which can be wired or wireless. The link between the end points and the network functions (NFs) is established by a logical interface, which is represented by dotted lines.

3.4 Advantages of Network Function Virtualization (NFV)

NFV's flexibility and abstraction offers various advantages, including:

3.4.1 Service Flexibility

NFV enables rapid network service deployment and scalability [28], allowing service providers to swiftly provide new services or adjust current ones to suit changing client demands.

3.4.2 Cost Efficiency

NFV decreases dependency [28] on expensive dedicated hardware by virtualizing network services, resulting in cost reductions in terms operating expenditures. By consolidating numerous services onto shared hardware infrastructure, it enables more effective resource utilization.

3.4.3 Scalability and Elasticity

NFV enables network services to be dynamically scaled in response to variable demand. In response to changing network circumstances [28] or user requirements, service providers may quickly scale up or decrease the resources assigned to a single service.

3.4.4 Service Chaining

NFV enables the formation of service chains [28], in which network services are coordinated and coupled in a predetermined order to offer end-to-end service capability. This allows for the development of sophisticated service architectures and improves service customization.

3.4.5 Network Programmability

NFV employs SDN concepts to provide centralized administration and control of virtualized network services. It enables service automation, orchestration, and policy-based administration, enhancing overall network programmability and flexibility.

3.5 Addressing the Obstacles of Network Function Virtualization (NFV)

Network Function Virtualization (NFV) has many advantages, but it also has certain problems that must be overcome for successful adoption. One problem is ensuring that virtualized network operations executed optimally and at scale, as they may involve significant processing cost as compared to dedicated hardware. To maximize performance, efficient resource allocation and management are also required. In virtualized systems, network security becomes an issue, demanding rigorous [29] safeguards to secure VNFs. Extensive frameworks and tools are also required for successful orchestration and control of VNFs, as well as for fault management. Interoperability and standardization are critical for easy integration, whereas legacy system integration and migration need careful preparation. Finally, in order to fully utilize the benefits of NFV, organizations must traverse cultural transformations and adopt new skills and procedures.

3.6 A Comparative Analysis of NFV and SDN

SDN and NFV are two independent but complimentary technologies that seek to revolutionize existing network topologies. The goal of NFV is to virtualize network functions by replacing specialized hardware with software-based VNFs. However, SDN decouples the control and data planes and centralizes network control via a logically [30] centralized controller. SDN allows programmability and dynamic network management by isolating control logic from network hardware. NFV focuses on virtualizing network functions, whereas SDN focuses on centralized control and programmability. When NFV and SDN are coupled, they produce a strong network architecture that is agile, scalable, and readily adaptable, resulting in benefits such as quick service launch, cost efficiency, centralized control, and network programmability.

4. Chapter 04 - Programming Protocol-Independent Packet Processors

4.1 P4 Language

SDN facilitates the setup and management of large-scale networks, however it frequently encounter restrictions in control protocols, limiting its potential. These protocols only cover a subset of available functionality and may not be compatible with all hardware alternatives. To resolve this challenge, a new tools identified as the Domain-Specific Language (DSL) for P4 has emerged. P4 is built to offer clear description of switch behavior, allowing network administrators to create their own software-oriented applications for programmable switches. A compiler or an interpreter [31] can be used to convert P4 code into an executable script. Figure 7 shows the architecture of a P4-based system, showcasing its components and functionalities.

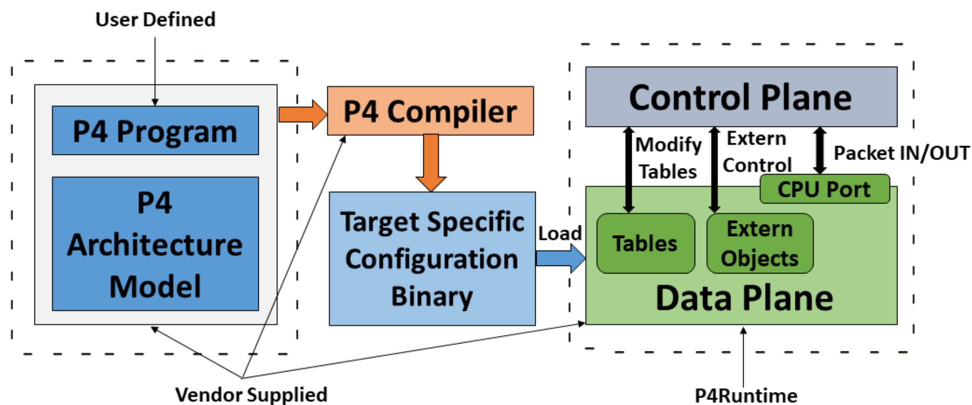


Figure 7: P4 Architecture

P4 enables network engineers and researchers to define how packets are handled and sent in network devices' data planes. It offers a high-level abstraction for determining the behavior of forwarding devices like switches and routers, regardless of the underlying hardware or protocols. Packet headers, header fields, and packet processing logic are defined when code is written in P4. Custom packet parsers, match-action pipelines, and packet modification [31] actions are all possible with P4. The constraint of OpenFlow in allowing customized protocols drove the development of P4. However, when it comes to packet processing, P4 contains certain OF benefits. P4, like OF, processes packets by taking actions depending on header field values. However, unlike OF, where the mapping is set at build time, P4 allows for dynamic mapping of packet processing operations by a control plane during runtime. The packet processing in P4 is shown in Figure 8 and it involves four major phases:

4.1.1 Packet Parsing

When a packet is received, it must be translated into a format that can be handled in the following steps. The basic program of P4 builds a predetermined state mechanism [32], which is used to parse the packet.

4.1.2 Ingress Pipeline

The packet enters the ingress pipeline at this phase, and there are no limits on executing rules depending on the packet. It is possible to do matching on several header fields, allowing the switch to decide the egress pipeline for later packet management. The P4 software [32] can obtain additional information of hardware port from which packet originated, and to where it can be resubmitted to re-enter the ingress pipeline as needed.

4.1.3 Egress Pipeline

The egress pipeline executes rules dependent on the parsed header data. Submissions to other egress pipelines before resubmissions [32] are not allowed during this time.

4.1.4 Packet Deparsing

In the last phase, the packet is deparsed depending on its present condition, preparing it for transmission over the wire. The deparser, which manages the deparsing process, is generated automatically by the parsed object. P4 is an SDN programming [32] language that allows network operators to specify packet processing and forwarding behavior in a flexible and protocol-independent manner. It enables network customization, innovation, and the growth of networking paradigms.

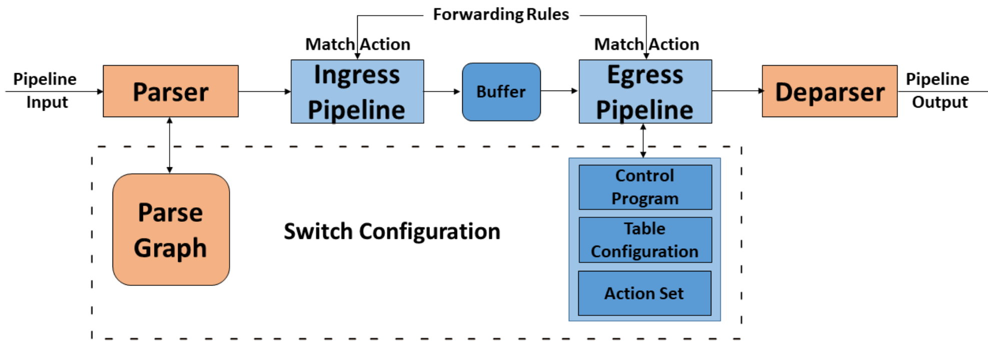


Figure 8: P4 Packet Processing

4.2 Behavioral Model

P4 behavioral model refers to a software-based simulation environment that allows P4 program to be tested and evaluated without the use of actual hardware switches. It provides a high-level abstraction of the P4-specified switch behavior and allows developers to experiment with various P4 programs. In software, the behavioral model emulates the behavior of a P4-programmable switch. It has several components that imitate various stages of packet processing, such as parsing, match-action tables, and packet changes. The model offers a virtual representation of the switch pipeline, allowing packets to be processed and sent in accordance with the P4 program rules. In order to generate an executable file from a P4 script, either a compiler or an interpreter is necessary. In this context, the P4 Language Consortium has released the P4 compiler called

p4c-behavioral. However, for various reasons, p4c-behavioral has been replaced by the bmv2. Implemented in C, bmv2 fully incorporates all the features specified in the P4 requirement. The p4c Python library offers a [34] target-independent P4 parser.

The main challenge is to build an executable that requires double compilation (P4 script to C language and from C language to binary form). Moreover availability of an ingress and egress pipeline is expected, contradicting concept that P4 script requires no hardware attributes; adjustments are required to properly enable P4.

A new behavioral model known as bmv2 [34] has been developed and used as an interpreter. The first step in running a P4 program is to compile the P4 basic script into a JSON file which then merged using appropriate P4 program to generate interpreter's input. The compilation process, including the output of JSON, is aided by p4c-bm, used to build program-specific C++ script. Tables, parser settings, checksums, ingress and egress pipelines, and the deparser are all configured using this JSON format in bmv2. In the p4lang environment, two compilers are existing for bmv2, one of which, p4c, includes a bmv2 backend.

The bmv2 backend currently supports the v1model architecture and provides limited support for the PSA architecture. P4 program developed for the v1model can be run using the simple_switch binary, whereas PSA program may be run with the binary switch psa file. However, p4c-bm is a former translator for bmv2, which is no longer being supported.

4.3 V1 Model Architecture

The simple_switch target is the primary focus of the V1Model architecture implementation in bmv2. P4.org [33] has created a software switch called bmv2 simple_switch is used to test the functioning of P4 program. The V1Model architecture, seen in Figure 9, is made up of numerous components. It comprises a P4 programmable parser and deparser that handle incoming and outgoing packet translation.

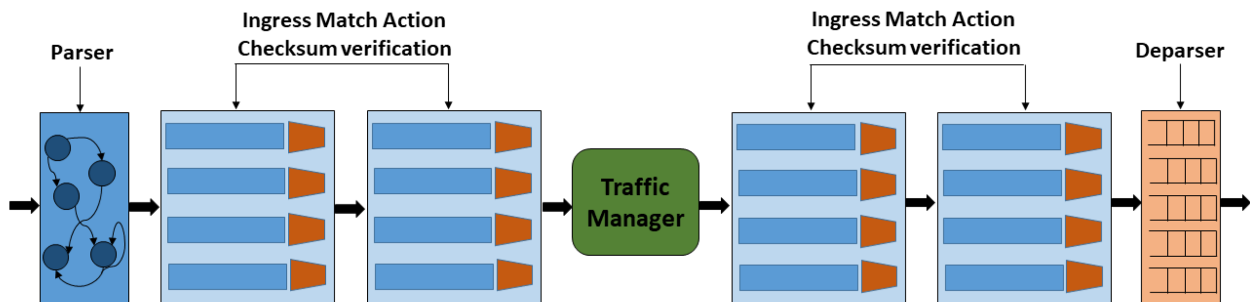


Figure 9: P4 V1 Model

In addition, match-action processing is handled by distinct ingress and egress pipelines. The design also includes blocks for validating and updating checksums on packets. A traffic management component is also present, which handles responsibilities such as packet scheduling

and possible replication across input and output ports. A collection of standard information field is used to aid packet routing [33] and processing within the bmv2 simple_switch. The P4 program specifies these information fields, which direct the packet's path via the bmv2 simple_switch.

4.3.1 Metadata Fields

A specified collection of fields that are routinely used in network packet processing and routing is referred to as standard metadata. These fields include information about source and destination addresses, [34] packet timestamps, QoS settings, flow IDs, and other pertinent packet properties.

The usage of standard metadata fields enables interoperability and compatibility among various networking components and protocols. These information fields can be read and modified by network devices such as switches and routers in order to make routing decisions, implement traffic management policies, or conduct other packet processing activities

```
struct metadata {
    bit<9> ingress_port;
    bit<9> egress_spec;
    bit<9> egress_port;
    // Other metadata fields
};
// P4 program using metadata
parser MyParser(packet_in packet, out headers hdr, inout metadata meta) {
    // Parsing logic here
}
control MyIngress(inout headers hdr, inout metadata meta, inout standard_metadata_t
standard_metadata) {
    // Accessing metadata fields
    meta.ingress_port = standard_metadata.ingress_port;
    meta.egress_spec = some_value; // Set egress_spec field
    // Other ingress control logic
}
control MyEgress(inout headers hdr, inout metadata meta, inout standard_metadata_t
standard_metadata) {
    // Accessing metadata fields
    bit<9> egress_port = meta.egress_port;
    // Egress control logic
}
control MyDeparser(packet_out packet, in headers hdr, in metadata meta) {
    // Deparsing logic here
}
```

In this code, a metadata struct with three fields is defined: ingress port, egress specification, and egress port. These fields are used to store metadata relating to packet processing. The metadata object is an output parameter in the MyParser parser, which means it may be edited and updated while parsing. The ingress_port metadata field in the MyIngress control is set to the value standard_metadata.ingress_port, which shows packet arrival port. Egress port can be specified to which packet should be transmitted using the egress specific field. The egress port field of metadata is read in the MyEgress control to identify the packet's outgoing port. Finally, the metadata object is supplied as an input parameter to the MyDeparser, providing access to its fields during deparser operations.

4.3.2 P4 Code and Functionality

The program starts by specifying the packet header format and generating header and metadata structs. As the packet traverses the architecture, these structs, together with the standard_metadata, are sent between different blocks. Throughout P4 source file, the functionality for parser and deparser, checksum verification and updating block, match and action processing at ingress and egress is defined.

```
#include <core.p4>
#include <v1model.p4>
struct metadata {
    bit<9> egress_port;
}
struct headers {
    ethernet_t ethernet;
    ipv4_t ipv4;
}
parser MyParser(packet_in packet, out headers hdr, inout metadata meta, inout standard_metadata_t smeta) {
    extract(hdr.ethernet);
    extract(hdr.ipv4);
}
control MyVerifyChecksum(in headers hdr, inout metadata meta) {
}
control MyIngress(inout headers hdr, inout metadata meta, inout standard_metadata_t std_meta) {
}
control MyEgress(inout headers hdr, inout metadata meta, inout standard_metadata_t std_meta) {
}
control MyComputeChecksum(inout headers hdr, inout metadata meta) {
}
control MyDeparser(inout headers hdr, inout metadata meta) {
}
V1Switch(
    MyParser(),
    MyVerifyChecksum(),
    MyIngress(),
    MyEgress(),
    MyComputeChecksum(),
    MyDeparser()
) main;
```

The metadata struct in the above P4 program defines the egress_port metadata field, which is used to hold the egress port value. The headers struct is used to aggregate all the headers together and contains the stated header types such as ethernet_t and ipv4_t. Using the extract method, the MyParser parser extracts the header information from the incoming packet. The MyVerifyChecksum control is in charge of validating the header checksums. In this control, necessary checksum verification techniques can be implemented. The entrance processing logic is represented by the MyIngress control. Changes or calculations to the packet headers is done here. The egress processing logic is handled by the MyEgress control. This control, like the ingress control, allows to perform any required alterations or computations. The MyComputeChecksum control is in charge of computing the header checksums. In this control, implement the necessary checksum computation procedures. The MyDeparser control defines how the packet's headers should be reassembled. Finally, the V1Switch instantiation has the controls that were created in the right order: MyParser, MyVerifyChecksum, MyIngress, MyEgress, MyComputeChecksum, and MyDeparser.

4.3.3 P4 Basic Headers

Bit-strings and integers are the two major types of fundamental data types supported by P4 program. In P4, bit-strings are unsigned integers with variable widths. They provide operations like addition, subtraction, concatenation, and slicing, which are akin to manipulating strings in Python or buses in Verilog. In P4, integers are referred to as 'ints' and offer partial support for the fundamental operations found in bit-strings. It adds the idea of variable-sized bit-strings for IPv4 protocol that has fields whose breadth is decided at runtime. The maximum allowable breadth of the bit-string [37] is represented as 'n' in such circumstances. The P4 standard includes a detailed description of the operations and features that are available for these data types. Headers are aligned to the byte level and might be in either a valid or invalid state. Methods such as isValid(), setValid(), and setInvalid() allow P4 programs to change a header's validity bit.

```
header ethernet {
    bit<48> dstAddr;
    bit<48> srcAddr;
    bit<16> etherType;
}
header ipv4 {
    bit<4> version;
    bit<4> ihl;
    bit<8> diffServ;
    bit<16> totalLen;
    bit<16> identification;
    bit<3> flags;
    bit<13> fragOffset;
    bit<8> ttl;
    bit<8> protocol;
    bit<16> hdrChecksum;
    bit<32> srcAddr;
    bit<32> dstAddr;
}
```

This code shows how to declare Ethernet and IPv4 headers in P4 script like structure of the Ethernet header, containing elements such as destination address, source address, and Ethernet type. Similarly, the IPv4 header is disclosed, together with its associated fields such as version, length, distinguished services, and total length. The 'typedef' phrase can be used to create alias names for complicated header structures, hence improving code readability and maintainability. The 'struct' data type in P4 is a flexible data structure that permits the formation of nested set of members with no aligned constraints. It allows for greater flexibility in data organization and manipulation within a P4 program.

P4 allows to combine numerous headers of the same kind into a header stack. A header stack is an array of headers that allows for efficient handling of packets having many instances of the same header type. It adds the idea of a header union, which is an alternative that contains one of multiple possible headers. A header union, for example, may include an IPv4 and an IPv6 header, presuming that packets would contain either IPv4 or IPv6, but not both at the same time.

```
    bit<16> value1;
    bit<32> value2;}
header Ethernet {
    bit<48> dstAddr;
    bit<48> srcAddr;
    bit<16> ethType;}
header IPv4 {
    bit<4> version;
    bit<4> ihl;
    bit<8> dscp;
    bit<16> totalLen; // ...}
header IPv6 {
    bit<4> version;
    bit<8> trafficClass;
    bit<20> flowLabel; // ...}
header_union IP {
    IPv4 ipv4;
    IPv6 ipv6;}
header_stack Headers {
    Ethernet eth;
    IP ipHeaders;}
metadata MetadataStruct;
// P4 program code continues...
```

This script builds a struct named 'Metadata' with two members: 'value1' of 16-bit type and 'value2' of 32-bit type. This struct is used to hold extra details about packets. Following that, defined two headers: Ethernet and IPv4. The Ethernet header contains information of destination address, source address, and Ethernet type. Version, Internet Header Length (IHL), Differentiated Services Code Point (DSCP), and overall length are all fields in IPv4 header.

A header union named 'IP' is specified to handle packets with distinct IP versions, each contains either an IPv4 or an IPv6 header. The 'Headers' header stack is introduced, which consists of an Ethernet header followed by an array of IP headers that allows packets containing many levels of IP headers, such as encapsulated packets, to be processed. The 'MetadataStruct' class is an instance of the 'Metadata' struct, which is used to hold extra metadata information about packets and enable development and manipulation of complicated data structures within a P4 program, allowing for efficient packet processing and handling of various header types and packet formats.

4.3.4 Parser

Parsers is program that convert packets into headers and information. They are implemented in the form of a state machine and has three predetermined states: start, accept, and reject. P4 programmer can construct custom states based on their needs, while it is not required to have a state for each [38] header type. In the given below parser code, MyParser is a parser that takes an input packet, extracts the headers, and updates the metadata accordingly. The parser begins in the start state, where it uses the extract command to extract the Ethernet header. Depending on the circumstance, it then moves to another header parsing state (parse_next_header_state) [42] or immediately to accept or refuse state. The transition statement selects the next state depending on circumstances presented. If the parser discovers the accept state, it takes the actions specified in

that state. Similarly, if it enters the refuse state, recording or discarding the packet, might be done. Loops are permitted in parsing state machine, allowing for the processing of repeating header structures or other parsing needs.

```
parser MyParser(packet_in packet, out headers hdr, inout metadata meta) {
    state start {
        extract(hdr.ethernet);
        transition select(next_state) {
            accept: accept_state;
            reject: reject_state;
            default: parse_next_header_state;
        }
    }
    state parse_next_header_state {
        if (condition) {
            extract(hdr.ipv4);
            transition select(next_state) {
                accept: accept_state;
                reject: reject_state;
                default: parse_another_header_state;
            }
        } else {
            transition select(next_state) {
                accept: accept_state;
                reject: reject_state;
                default: parse_another_header_state;
            }
        }
    }
    state accept_state {
        // Actions to be performed upon reaching the accept state
    }
    state reject_state {
        // Actions to be performed upon reaching the reject state
    }
}
```

4.3.5 Simple Actions

Control blocks in P4 architecture are programmable elements responsible for various packet processing activities alike C language functions, although they don't support loops. Variables, tables, and externs can all be stated within a control block. The apply block within a control block defines the block's real functionality, where various types of packet processing, match-action pipelines, and deparsers can be described as directed acyclic networks. User-defined and architecture-specific types, often involving headers and metadata, are used to establish interfaces between control blocks and other components within an architecture. In below script, control block 'MyControl' receives the headers and metadata and alters them as needed. The source and destination MAC addresses are exchanged within the apply block using the temporary variable tmp. The packet is then sent out over the same input port by changing the standard_metadata egress_spec field to the ingress_port. Temporary variable is distinctive to this control block which is not easily retrieved by any block. To provide same functionality, the temporary variable may have been stated directly within the apply statement.

```
control MyControl(inout headers hdr, inout metadata meta) {
    // Declare a temporary variable for swapping MAC addresses
    bit<48> tmp;
    apply {
        // Swap source and destination MAC addresses
        tmp = hdr.ethernet.srcAddr;
        hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
        hdr.ethernet.dstAddr = tmp;
        // Send the packet out through the same input port
        standard_metadata.egress_spec = standard_metadata.ingress_port;
        // Perform any additional packet processing or actions
    }
}
```

4.3.6 Tables

In P4, tables are used for match-action within a pipeline. Multiple aspects are involved in the definition of a table, including the criteria for matching, the type of matching to be performed, a list of possible actions, and, if necessary, extra attributes such as entry count and a default action. Each table has many items known as rules. Each rule has a unique key for matching, an associated action to do when a match occurs by passing optional data to the action. A longest prefix match should be done by routing table on the destination IP. In P4, routing table definition can look like this:

```
table ipv4_lpm {
    key = {
        ipv4.dstAddr: lpm; // Longest prefix match on destination IP address
    }
    actions = {
        ipv4_forward; // Update destination MAC, configure egress port, decrement TTL
        drop; // Drop the packet
        NoAction; // No action
    }
    size = 1024; // Allocate enough room for 1024 entries
    default_action = NoAction; // Default action if no match is found
}
```

The key defines the IPv4 destination address and employs the lpm (longest prefix match) match type. The actions section specifies the potential actions that this table do: ipv4_forward, drop, and NoAction. The table has adequate space for 1024 items, and the default_action is set to NoAction if no match is discovered. The data plane, as described by P4, is in charge of structuring the forwarding table and define the match-able fields and determine the specific actions to be called. Selected actions are executed after table lookup. Control plane, on the other hand, control of populating table entries. Manual configuration by network operators, automated discovery, and routing protocol computations are examples of control methods. The deparser feature, which requires the rebuilding of headers into a correctly formed packet, can be implemented using a control block. Broadcast of the packet_out extern is used to validate specified header field into the packet at end, assuring its inclusion only if it contains valid data.

```
control deparser {
    apply {
        packet.emit(ethernet);
        packet.emit(ipv4);
        if (ipv4_options.isValid()) {
            packet.emit(ipv4_options);
        }
        packet.emit(tcp);
        packet.emit(payload);
    }
}
```

The deparser control block is defined in the code to reassemble the headers into identifiable packet. The control block begins by using packet.emit(ethernet) to emit the ethernet header. The ipv4 header is then emitted with packet.emit(ipv4). Using packet.emit(ipv4_options), if the ipv4_options header is valid, it is emitted. The tcp header is then emitted using packet.emit(tcp). Using packet.emit(payload), the payload is emitted. All this process is shown in Figure 10. The deparser control block guarantees that the headers are appropriately constructed in the required sequence to build a full packet for transmission by utilizing the emit method.

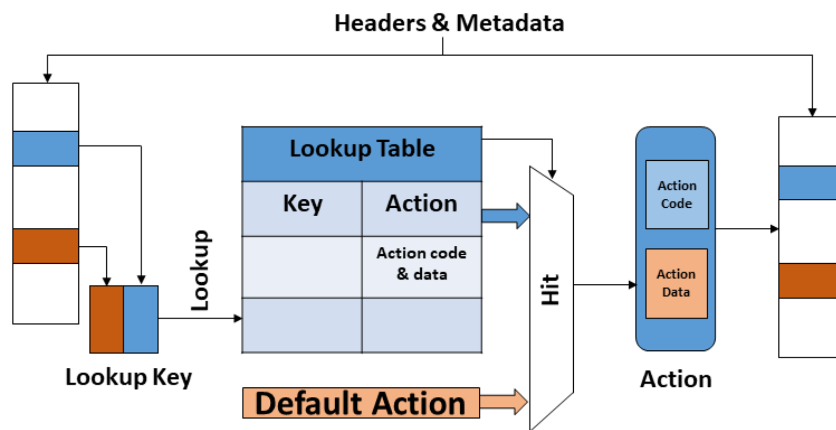


Figure 10: Match Action Data Flow

4.4 P4Runtime

P4 Runtime presents a novel method for managing forwarding planes of switches, routers, and firewalls. P4 Runtime, in contrast to typical closed and fixed APIs, has various advantages and allows control over both fixed-function and programmable switches, including ASIC-based [43] devices and software switches operating on x86 servers. In P4 Runtime, the framework remains self-contained, allowing a diverse set of switches to be controlled using the same API. Its API is automatically updated to meet novel protocols added to routing plane, avoiding need for control plane restarts or reboots. It has no restrictions on the location of the control plane, which on x86 [43] servers can be either a system operating on local switch or a remote control plane. Because of their limited scope and lack of extension, closed and fixed APIs have been a source of worry [43]. These proprietary APIs are usually customized for individual switch chips and are seldom updated.

To address these concerns, efforts have been undertaken to replace restricted APIs with open interfaces [43]. OpenFlow, which debuted a decade ago and allows remote control planes to handle switches from many manufacturers using a standardized API, is one such example. However, OF presented its own set of difficulties. It was primarily built for certain use cases and fixed-function switches, and it lacked flexibility and displayed ambiguous behavior.

AI is a solution that overcomes some of the issues associated with closed APIs, focused on networks where the control plane is located within the switch. SAI [22] like OF, allows control of switches depending on different switch ASICs. However, as it matures, SAI gets more sophisticated, making extensions difficult and remote control of switches confusing.

P4 Runtime, on the other hand, provides a full solution to the aforementioned difficulties [43]. It is a flexible and open framework that can handle any switch ASIC, allowing various networks to use the same API while including different protocols and capabilities. By specifying the forwarding behavior in P4, the P4 Runtime [43] enables for simple expansion over time, enabling the simulation of both OFand SAI behaviors.

4.4.1 Control Plane with P4 Runtime API

Within the same switch, P4 Runtime is used to operate switches from control plane that can be operating as local or remote. Necessary schema for the P4 Runtime API is built by utilizing the P4 code ‘switch.p4’ to specify the pipeline of switch and leveraging P4c, allowing addition and deletion of entries in the forwarding table at runtime. This method guarantees effective control of switches using the P4 Runtime API.

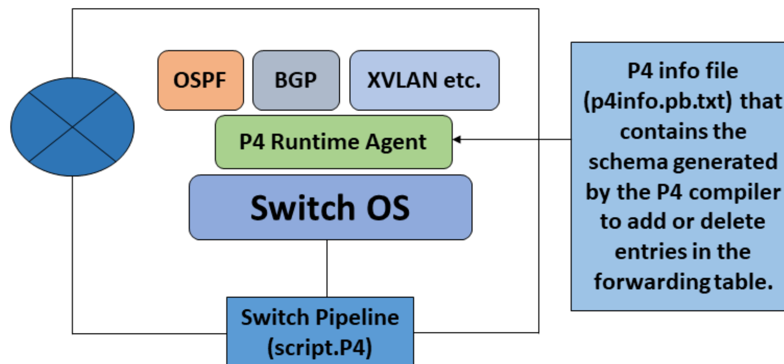


Figure 11: Local Control Plane with P4 Runtime

The SDN Controller is a logically centralized entity tasked with orchestrating the whole network in accordance with the administrator regulations. We used the ONOS controller in our scenario due to its specific benefits over other SDN controllers. ONOS distinguishes itself by powerful features such as seamless scalability, high availability, and dynamic network resource management. Figure 11 depicts a scenario in which control plane operating as local uses API of P4 Runtime that allows to manipulate any switch that is defined in the P4 code. A developer can

control switches of fixed-function nature, by writing a P4 script that accurately defines the behavior of the switch in the P4lang. P4c recognizes the components that require control of tables lookup defined inside the P4 script, for which entries must be inserted or removed. Using the P4 Runtime API, this automated procedure simplifies switch configuration.

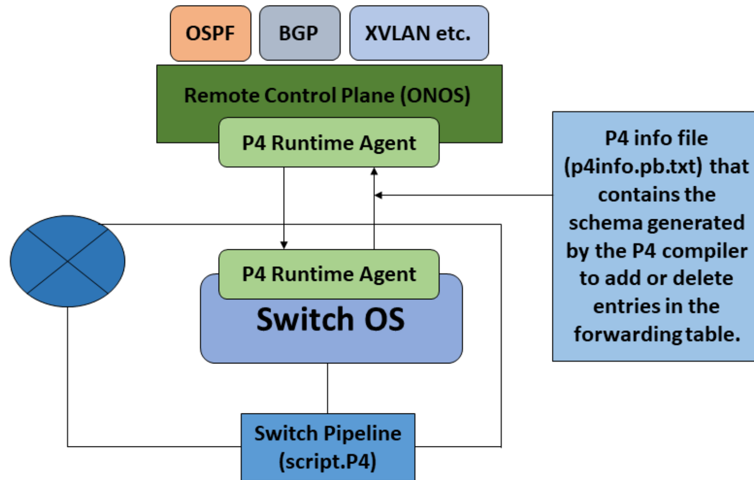


Figure 12: Remote Control Plane with P4 Runtime

In the configuration of Figure 12, the remote control plane connects to network switches via the P4 Runtime API, acting as an intermediate level. This setup allows administrators of networks to manage and alter switch functionality from a single centralized location, which is very useful in substantial networking installations covering various geographic areas. It allows for global changes, efficient routing, and responses that adapt to changing traffic patterns, all while preserving centralized and unified network administration.

Basically, the P4 Runtime API controls BMv2 switch in Figure 12, whereas a P4 implementation specifies the switch's pipeline, defining the way packets are processed and forwarding across the network's infrastructure. A P4 compiler (P4c) provides a required schemas for the P4 Runtime API to enable real-time changes to the routing table, allowing the creation and elimination of entries on as needed. This P4 file essentially serves as the switch's operational guide, allows controllers and network managers to dynamically adjust the switch's behavior. They can respond quickly to changing network conditions, flows of traffic, or particular needs by adding or removing forwarding table entries. P4 Runtime API, working together with a control plane operating remotely plays vital role in implementing the SDN goal, providing increased versatility and effectiveness in network management.

An IPv4 longest-prefix-match (LPM) table is included with the P4 program. Entries in this table will require insertion and deletion during runtime, notably concerning the 8-bit prefix (seen in the center of Figure 13). The P4 compiler provides a complete schema (seen on the right side of Figure 13) that acts as the control plane's blueprint.

The control plane use the protobuf schema to represent the exact 8-bit prefix that must be appended to the database. Using the P4 Runtime API, this schema allows the control plane to communicate the needed table adjustments to the table. P4 Runtime API supports the creation of new tables that are managed programmatically. A programmable switch is configured with IPv6 prefix table or a custom-made table as per network requirements. When additional tables are added, the P4c updates the schema (protobuf message) appropriately. As a result, the control plane control and manage these newly introduced tables through the P4 Runtime API [26].

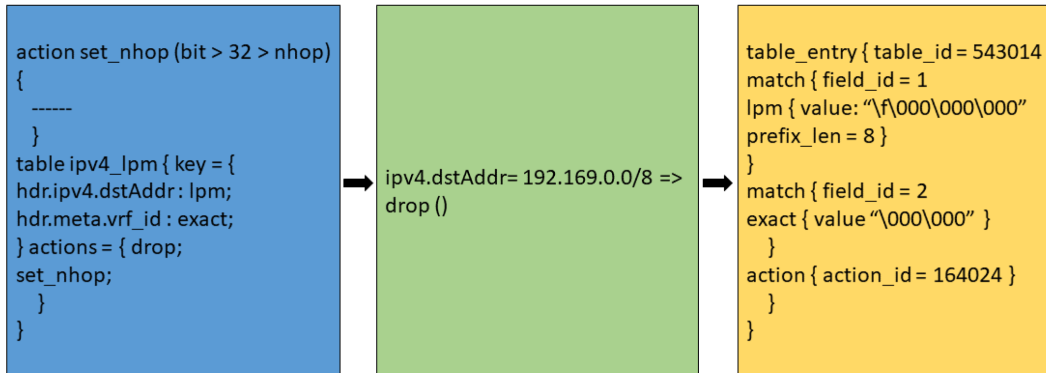


Figure 13: P4 Runtime Data Flow

4.5 P4 Advantages and Specifications

Because of its protocol independence, P4 is vital for SDN, allowing the introduction of additional protocols and customization. It allows it to provide unique network services and optimize traffic. Because P4 is hardware-independent, program may be transferred across devices, increasing deployment flexibility. The most recent P4_16 standard includes advanced language features such as support for complicated data structures and control flow. P4 is a useful tool for flexible, programmable, and adaptive SDN networks because it uses a match-action approach that allows rule-based packet processing and target-specific implementations for multiple hardware platforms.

5. Chapter 05 - Open Network Operating System

The Open Network Operating System (ONOS) is a popular open-source SDN controller when it comes to next-generation SDN/NFV solutions. It is based on OSGi technology, handles different sub-projects and provides extensive support for network configuration and real-time control. The use of ONOS [16] eliminates the requirement to run the control protocols (switching and routing) within the architecture of network. ONOS serves as an intelligent cloud controller, allowing the development of new network applications without affecting the underlying data plane technology. It has a scalable design to enable scalability while meeting the rigorous needs of commercial carrier settings.

5.1 ONOS Specifications

ONOS provides a fault-tolerant architecture based modularity principles that easily supports large-scale SDN deployments, supporting networks of varied sizes, from modest corporate setups to gigantic carrier-grade [44] settings. Some of the important aspects are as follows:

5.1.1 Centralized Management

ONOS provides a centralized control plane [13], allows network managers to see the whole network. It offers efficient network resource management, and dynamic network provisioning.

5.1.2 Northbound and Southbound Interfaces

ONOS offers a variety of [13] northbound interfaces, enabling the creation and integration of customized network applications and services. It also supports different southbound protocols, ensuring interoperability with a diverse set of SDN switches and devices.

5.1.3 Flow Management and Traffic Engineering

ONOS provides comprehensive flow management features, allowing for fine-grained [13] control over network device forwarding behavior. It provides traffic engineering processes such as network path optimization, load balancing, and quality of service (QoS) provisioning.

5.1.4 Network Virtualization

ONOS enables the establishment of several [13] logical networks over a shared physical infrastructure, facilitating network virtualization for efficient multi-tenancy.

5.1.5 Application Ecosystem

ONOS has a thriving and active community that creates and supports a diverse set of network applications. These programs supplement ONOS's capabilities by adding features such as

network monitoring, security, and network analytics. ONOS offers following benefits that make it a popular choice for SDN deployments:

5.1.6 Flexibility and Programmability

ONOS provides a highly [13] programmable environment, allowing network operators to customize and adjust their networks' behavior. Its modular architecture and extendable foundation make it possible to create and integrate new network services and protocols.

5.1.7 Scalability and High Availability

ONOS is built for large-scale networks with heavy traffic volumes. Its distributed design guarantees scalability and high availability, reducing the effect of failures and ensuring network stability.

5.1.8 Open-Source Community

ONOS has the support of a thriving open-source [13] community of developers, academics, and network operators. Because of this community-driven approach, ONOS is a stable and future-proof alternative for SDN installations, with ongoing development, bug fixes, and feature updates.

5.2 Design Principles of ONOS

The early design aims for ONOS centered on fulfilling the following [17] objectives:

5.2.1 Code modularity

Allowing for the creation of additional functionalities as standalone modules.

5.2.2 Configurable Features

The ability to dynamically load and unload features during startup or runtime.

5.2.3 Protocol Independence

It refers to the ability of programs to be independent of certain protocol libraries and implementations.

ONOS source code is organized hierarchically, making sub-project administration easier [17]. Every sub-project possess individual directory along with a pom.xml file, which inherits parent POM file's common dependencies and settings. This modular form enables for the construction of independent sub-projects. The root directory contains the top-level POM files that are responsible for constructing the whole project and its modules.

Karaf is used as ONOS Service Gateway initiate technology that include the ability to construct a secure API interface using standard JAX-RS API, support for centralized custom settings secure shell that include both local and remote control login enable by an extensible CLI, and the ability to capture logs at various levels. ONOS has a layered design that includes the following components, regardless of the underlying [17] protocol:

5.2.4 Protocol Awareness Module

This module interacts with the network and is does protocol-specific actions.

5.2.5 System Core

Tracks and delivers network status information, offering a standard foundation for network management.

5.2.6 Applications

ONOS has protocol awareness module that communicates network and offers information about network states, and apps that utilize and take actions depending on the data offered by core. As depicted in Figure 14, the core establishes communication with network centric modules through a southbound (provider) API and interfaces with applications via a northbound (consumer) API. The SB API provides a protocol-agnostic means of transmitting network status data to the Core, which then engages with network devices through network modules. NB API [17] gives abstractions defining network properties to apps, allowing them to define their desired actions.

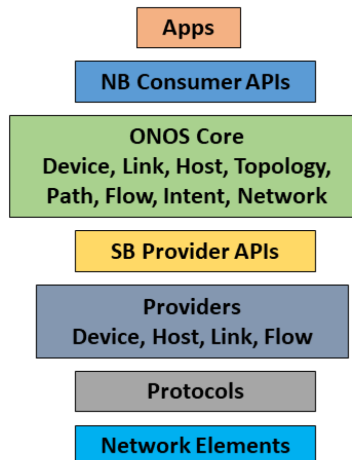


Figure 14: ONOS Stack

5.3 System Components of ONOS

A subsystem is a component that collectively comprise a service, whereas a service is a functional unit made up of numerous components that span different levels, producing a software

stack that allows for development [17] of vertical slices. ONOS subsystem is shown in Figure 15 with its key attributes outlined as follows:

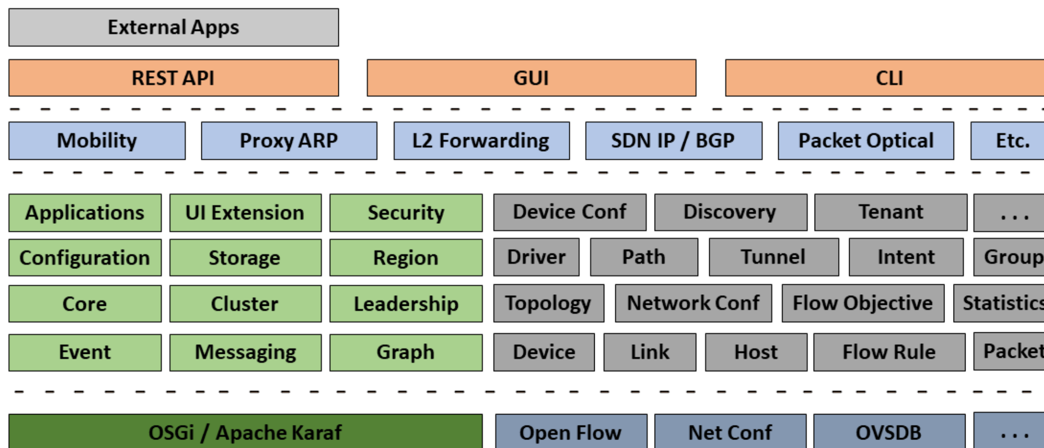


Figure 15: ONOS Components

5.3.1 Device Subsystem

Manages infrastructure device inventories managed by the [17] Link Subsystem.

5.3.2 Host Subsystem

Keeps track of end-station hosts [17] and their network locations.

5.3.3 Topology Subsystem

This subsystem is used to manage time-ordered snapshots of network graph views.

5.3.4 Path Service

Devices path or end-station hosts are computed using the most current topology graph snapshot.

5.3.5 Flow Rule Subsystem

It delivers flow metrics for the inventory of match/action flow rules deployed on infrastructure devices however packet subsystem [17] allows apps to receive incoming packets from network devices and transmit outgoing packets into the network via one or more network devices.

5.4 ONOS Subsytem Structure

In ONOS subsystem structure the Provider component [19] is located at the bottom of the stack. It connects with the underlie device using a protocol specific library and interacts with the core via the ProviderService interface. Each Provider has its own ProviderId, which serves as an

external identification for the Provider family. Multiple Providers can be attached to the subsystem, each of which is classed as major or auxiliary. The primary Provider owns the entities linked with the service, whereas the ancillary Provider uses this information, giving precedence to primary Provider.

The Manager component is responsible for receiving information from Providers and sending it to applications and other services. The NB and AdminService interface, SB ProviderRegistry [19] and ProviderService interface are among its interfaces.

The Store, which is linked to the Manager in the Core, employs specialized procedures to index, persist, and synchronize the data received from Providers. This guarantees that information is resilient and consistent across various ONOS instances.

Through the AdminService and Service interfaces, applications consume and process information acquired from the Manager. They perform tasks of presenting network architecture [19] via a web browser and customizing network traffic pathways. Each application, like Providers, is assigned a unique ApplicationId, allowing ONOS to monitor the context associated with that specific application.

ONOS sends information in two basic forms: events and descriptions. These units, once constructed, do not alter and are related with certain network parts and concepts. Descriptions provide information about items through the SB API, whilst events act as change notifications spread across Managers and Stores.

The Store generates events based on the Manager's input. They are constructed and then distributed to interested listeners through the StoreDelegate interface. The StoreDelegate guides the event, and the EventDeliveryService ensures that it reaches just the appropriate listeners. The Manager contains both of these components, with the StoreDelegate supplying the implementation class to the Store. Event listeners are components that implement the EventListener interface. EventListener child interfaces are classified based on the type of Event subclass they handle.

5.5 Network-state Representation

The control plane is used to handle and obtain information of [18] network state. It collect this data and make it available to applications. ONOS uses network detection and configuration tools to build a protocol-agnostic topology that takes use of the benefits of both methodologies. It maintains network element and state representations [18] that are independent of specific protocols, allowing for smooth translation between multiple representations. Network directives are specified at the application level as flow rules composed of match criteria [18] and action treatment pairs. The incoming network traffic and packets to be added in network are equivalent to the Packet IN/OUT notions defined by OpenFlow.

5.5.1 Outbound Packet

A protocol-independent representation of a synthetic packet designed for network transmission. It contains information about the packet's destination.

5.5.2 Inbound Packet

A protocol-independent representation of a network device's packet transmitted to the controller. Packet IN is made available to providers and applications for host monitoring and connection detection and allows reactive packet processing. There is a dependence connection between items in ONOS. Ports [18] cannot exist in the absence of a Device, and Links cannot exist in the absence of Ports since they act as endpoints. As a result, in ONOS' network representations, devices are regarded as a basic element. ONOS uses following terminology:

Subject: A reference to an item that will be configured via the subsystem like A DeviceId, is used to represent a network device.

Config: A combination of changeable parameters presented for a certain item. BasicDeviceConfig, for example, lets you to establish or change a device's type and southbound driver.

Key: A string name assigned to a topic that serves as the key for the JSON field holding the configuration variables. Device configurations can be searched using the key ‘devices’ in the field. A configuration key is a string name supplied to a configuration class that acts as both the configuration class identity and the JSON field key. The ‘basic’ key describes the device's universal setups. A configuration operator is responsible of harmonizing [18 multiple sources of network setup data related to a certain object. Its duty is to consolidate and manage configuration data from many sources to ensure consistency and correct system application.

5.6 Device Subsystem in ONOS

The Device subsystem in ONOS does the task of identifying and monitoring the network's devices. It allows operators and applications to operate these devices, and many of ONOS basic sub-systems rely on the Device and Port model [18] objects, created and managed by the Device subsystem. The Device subsystem is made up of the following elements:

5.6.1 Device Manager

It talks with numerous Providers via the Device Provider Service interface and with multiple listeners via the Device Service interface.

5.6.2 Device Providers

This connect with the network using their own network protocol libraries or methods.

5.6.3 Device Store

It manages Device model objects and creates Device Events. The OF Device Provider, which supports interaction with OF devices, is one of the Device Providers utilized by ONOS. Major representations translated between the two levels are shown in Figure 16, with the core tier presenting different network components and attributes as [18] protocol-agnostic model objects and provider stage representing them as protocol-specific objects.

OF Device Provider and driver components are part of the ONOS OF subsystem. It makes use of the [18] Java protocol bindings produced by Loxi [45] to implement the OF protocol's controller-side behavior.

Figure 16 displays the OF subsystem's southbound design. The OF Controller coordinates with OF functions and produce OF events to which providers can subscribe. Providers can implement a variety of listeners, including [18] OF Switch Listener, OF Event Listener, and Packet Listener, which handle switch events, OF messages, and incoming traffic packets, respectively.

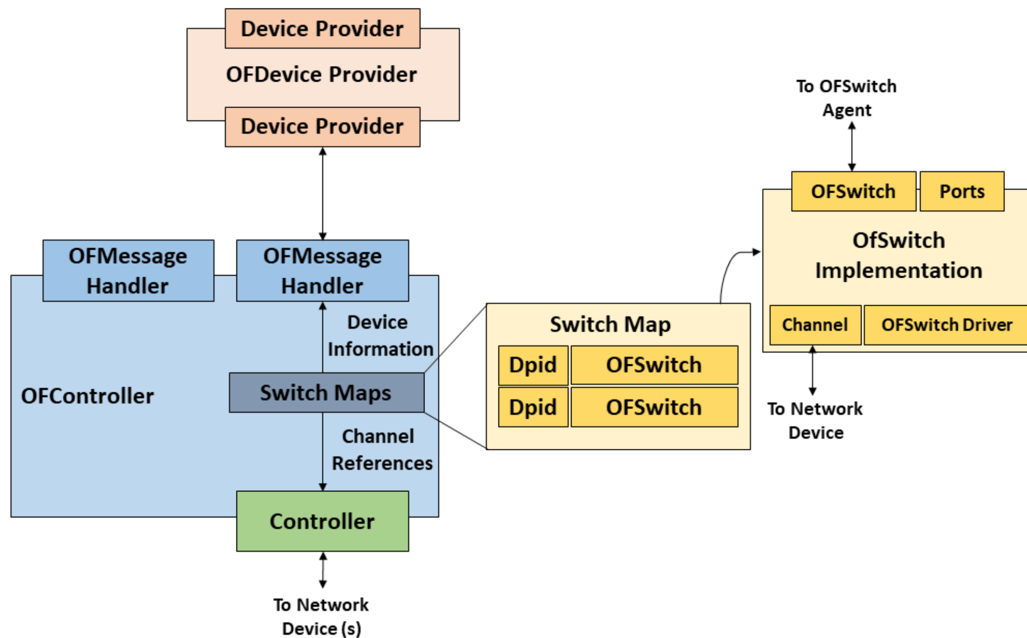


Figure 16: OF Provider

The OF Controller creates and maintains communication channels for each Switch object. The Controller makes connections, while the OF Switch Agent monitors the condition of each linked switch. The Switch object serves as the OF subsystem's portrayal of a network device. It contains port information, [18] device data, a unique identity, and, on the other end, a channel reference to the actual connected device. It has two interfaces that face in opposite directions: OF Switch and OF Switch Driver. The OF Switch interface is oriented northward towards Providers, whereas the [18] OF Switch Driver interface is oriented southward towards the channel and the controller.

5.7 Device Driver Subsystem

The Device subsystem in ONOS is responsible for isolating [18] device-specific code and preventing it from propagating throughout the system. A Driver in the Device subsystem represents a certain device family or a specific device. It has the following characteristics: a distinct name, a set of Behavior classes that it supports, the ability to inherit behaviors from another Driver, and the capacity to be abstract. ONOS provides the Default Driver class to implement the Driver interface.

Driver Providers are entities that can offer device drivers and the behaviors that go with them. The Driver Admin Service is in charge of controlling device drivers indirectly by managing driver suppliers and has functions such as [18] getProviders(), registerProvider(), and unregisterProvider(). The Driver Service is the principal service used by ONOS apps and other subsystems to identify appropriate drivers for a device. It allows [18] to search for drivers by name, device maker, hardware version, software version, supported Behavior, and device ID.

Driver Data is a container storing information about a device gathered from previous encounters. It has a parent Driver and Behaviors for describing a device [18].

6. Chapter 06 - Environmental Setup

This chapter gives guideline to the experimental setup that involves following setups:

6.1 System Specifications

We build our SDN experimental setup and application development stack by combining free and open-source components. Our main computing resource was a Huawei server outfitted with a powerful Xeon processor. We used 32 GB of RAM at our disposal to execute memory-intensive operations and allocated to two virtual machines (VMs) developed for our testing. The Mininet and BMv2 components were hosted in a single VM1 with 16 GB of RAM. The VM2 was given 10 GB of RAM and was used to install ONOS instances. The server included SAS storage with a 2TB storage capacity for data storage. We also assigned a 200 GB virtual hard disk drive (VHDD) for specialized storage and virtualization requirements. To develop and administer our network emulations, we used Mininet version 2.3.1. The Linux distribution we used was Ubuntu 20.04.5. As our major computing resource, we used an Intel(R) Xeon(R) Silver 4210 CPU, equipping the system with 6 virtual CPUs (vCPUs) to efficiently spread processing workloads. Our virtualization infrastructure was built on VMware ESXi version 7.0 U2, a solid platform that provided us with advanced virtualization capabilities, boosting the efficiency of our tests even further. This extensive experimental setup gave us the computational capacity and resources we needed to conduct our research efficiently.

6.1.1 Manual Installation

The following components are required for manual Docker installation:

- Docker v1.13.0 and later (with docker-compose)
- Python 3
- Bash-like Unix shell
- Wireshark (optional)

We used the following commands to update the system's package repository and installed the Docker Compose plugin:

```
sudo apt-get update

sudo apt-get install docker-compose-plugin
```

We used the repository of ngsdn-tutotiral from github and cloned it in our home directory and upgraded the required dependencies:

```
sudo make deps
```

This repository is organized as follows:

- p4src/ P4 implementation
- yang/ Yang model used further
- app/ custom ONOS app Java implementation
- mininet/ Mininet script to emulate a 2x2 leaf-spine fabric topology of stratum_bmv2 devices
- util/ Utility scripts
- ptf/ P4 data plane unit tests based on Packet Test Framework (PTF)

6.2 ONOS Installation using Docker

Installation of ONOS using Docker, as well as instructions on how to test a simple topology using mininet is described here. To install Docker, we logged in to the mininet VM with a user account that has superuser permissions and executed the provided commands to fetch the package lists from the repositories.

```
sudo apt-get update

sudo apt-get -y install docker.io
```

By creating a symbolic link between the installed Docker IO package files and the /usr/local/bin/docker directory, the Docker CLI can be executed by simply typing 'docker' into the Linux command line.

```
sudo ln -sf /usr/bin/docker.io /usr/local/bin/docker
```

By using the following command we downloaded the onos image:

```
sudo docker pull onosproject/onos
```

To run a single instance of ONOS used:

```
sudo docker run -t -d -p 8181:8181 -p 8101:8101 -p 5005:5005 -p 830:830 --name onos
onosproject/onos
```

The preceding command is set up with the subsequent options:

Using -t will assign a pseudo-tty to the container.

The -d flag will execute the container in the foreground.

The option -p <CONTAINER_PORT>:<HOST_PORT> maps a CONTAINER_PORT to a HOST_PORT. Some of the specific ports that ONOS employs are as follows:

- Port 8181 for the REST API and GUI
- Port 8101 for accessing the ONOS CLI
- Port 9876 for intra-cluster communication between target machines

- Port 6653 for OpenFlow
- Port 6640 for OVSDB
- Port 830 for NETCONF
- Port 5005 for debugging, which can be utilized for attaching a Java debugger.

The execution of the preceding command, exposed the ONOS CLI, GUI, NETCONF, and Debugger ports. To operate a released version, we included the: Version_Number:

```
sudo docker run -t -d -p 8181:8181 -p 8101:8101 -p 5005:5005 -p 830:830 --name onos
onosproject/onos:2.1.0
```

```
sudo docker ps
```

Once the container has been launched, we navigate to the ONOS UI by accessing the following URL via Mininet: http://localhost:8181/onos/ui.

```
wget -O - http://10.3.12.139:8181/onos/ui > /dev/null
```

This command should be run from the Mininet terminal. To display the ONOS UI via a web browser, determine Mininet's IP address by running:

```
ip addr | grep eth0
```

In our virtual environment, the IP address assigned to P4 Mininet is 10.3.12.140. Consequently, the ONOS UI can be accessed using the following URL: http://10.3.12.139:8181/onos/ui. Once there, it will be necessary to log in to the system, which requires the default username ‘onos’ and password ‘rocks’. Upon successful login, the dashboard will display current topology of system.

As ONOS is operating inside a container, it is necessary to utilize SSH to access the ONOS instance. This can be achieved by executing the command "ssh -p 8101 karaf@<ONOS_IP>", where <ONOS_IP> should be replaced with the IP address of the ONOS instance. The IP address can be obtained from the topology page of the ONOS UI; in our case, the sole device listed on the page has the IP address 10.3.12.140, so it is connected using:

```
ssh -p 8101 karaf@10.3.12.139
```

After connecting, the user will be prompted to enter a password, which is ‘karaf’. Once connected, the controller can be configured. By executing the command ‘apps –s’, a list of all the applications currently installed on the controller can be obtained. To activate the OpenFlow application, run ‘app activate org.onosproject.openflow’. Additionally, it is necessary to install the ‘onos-apps-fwd’ feature by executing ‘feature:install onos-apps-fwd’.

To display only the active applications, executed the command ‘apps -s –a’. This will provide a list of the currently activated applications. The applications can also be managed from the ONOS

UI, specifically from the Applications page. From there, it is possible to activate or deactivate an application as required.

To start a minimal topology on Mininet, execute the command:

```
sudo mn --controller remote,ip=10.3.12.140, port=6653
```

This will create a topology that includes three hosts (h1, h2 and h3), four BMv2 switches (spine 1 & 2, Leaf 1 & 2), and is connected to the ONOS controller at 10.3.12.139. By running the ‘pingall’ command from the Mininet command-line interface of our VM1, the verification of ONOS controller is done. This command tests the connectivity between the two hosts (h1 and h2) in the topology by sending ICMP echo requests in both directions.

7. Chapter 07 - Experimental Setup

7.1 Operating System Stratum

Stratum is an operating system that was created particularly for software-defined networks (SDNs). It is an open-source, silicon-independent solution aimed at creating a production-ready white box switch distribution. P4Runtime and OpenConfig are two sophisticated SDN interfaces exposed by Stratum [19] that improve the flexibility and programmability of forwarding devices and behaviours.

One of Stratum's primary benefits is its ability to provide a wide variety of SDN features, such as control, configuration, and operational interfaces, over the network's full lifespan. Stratum allows smooth integration with multiple SDN systems by integrating the newest SDN NB interfaces like as P4, P4Runtime, [19] gNMI, and gNOI.

It is vital to note that control protocols are not included by default in Stratum. It is instead intended to support external network operating systems to coexist with NOS functionalities [19] utilizing the same embedded switch. Based on the unique network needs, this design method offers for greater flexibility and adaptation.

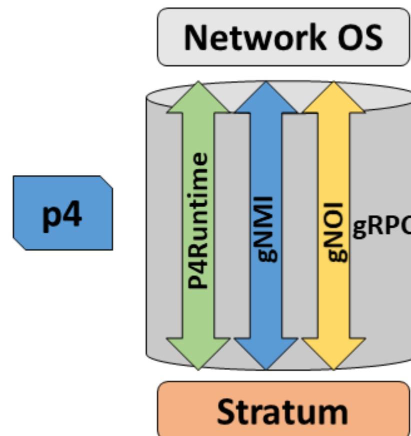


Figure 17: Stratum Controller

Stratum provides a Docker image that enables the execution of a network simulated using mininet, with stratum_bmv2 serving as the default switch. The image enables testing and demonstration of a variety of functions, including P4 runtime and gNMI etc.

This Docker image was used to generate the network architecture shown in Figure 17 to demonstrate the possibilities of Stratum. It allows users to test and evaluate technologies like P4 runtime and gNMI while exploiting ONOS's control plane capabilities. Users may simply set up and explore the functionality given by Stratum in a controlled environment by using the Stratum Docker image, enabling for fast testing and development of SDN applications and protocols.

The switch.p4 program, designed for the bmv2 simple target switch, undergoes compilation using p4c. To compile our switch.p4 program, Open Networking Docker image for p4c is utilized.

```
sudo make p4-build
```

The compiler generates the following files, with the primary output saved in p4src/build/bmv2.json (-o).

It describes a configuration for the BMv2 simple target switch in JSON format, which the simple switch uses to process incoming packets based on the P4 program. A P4Info file is also generated in p4src/build/p4info.txt (--p4runtime-files), which provides a Protobuf Text-formatted version of a P4Info schema for the P4 application.

7.2 Mininet Topology

To start an emulated network of stratum_bmv2 switches i.e. the topology, used the following command in our VM2 that has IP 10.2.12.140:

```
sudo make start
```

We performed this to start two Docker containers, one for Mininet and one for ONOS. To ensure that the container starts up smoothly, used the following command to inspect the Mininet log.

```
sudo make mn-log or docker logs -f mininet
```

In docker-compose.yml, the parameters for starting the mininet container are supplied. The container has been set up to run the topology script provided in mininet/topo-v6.py (press Ctrl-C to exit from the mininet CLI). When this command is executed, it start displaying the logs from the point where the container was initially started. To view the latest logs, without the older ones, use - - tail option followed by a number to specify the number of recent log lines).

```
Docker logs --tail 25 mininet
```

When the mininet container is started, following files pertaining operation of each stratum bmv2 instance is generated in the temporary directory.

- tmp/leaf1/stratum_bmv2.log
- tmp/leaf1/chassis-config.txt
- tmp/leaf1/write-reqs.txt

Our proposed architecture features a distinctive 2x2 fabric topology comprises of four BMv2 switches: spine 1, 2 and leaf 1, 2 interconnected with host: h1, h2 and h3 attached to leaf 1 as visually depicted in Figure 18. These hosts are connected directly through a single interface in

leaf switches, orchestrated and controlled by ONOS and programmed using P4 Runtime shell that links to a P4 Runtime server to perform P4 Runtime commands.

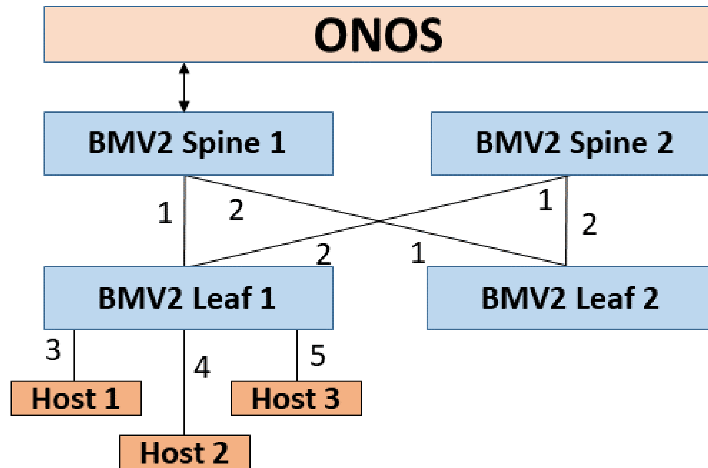


Figure 18: Mininet Topology

To support this virtual setup, we deployed two Docker containers, one to serve as the host for the Mininet topology and the other to utilise the ONOS operating system.

Our topology is created within the P4 and Mininet environment, using our VM2. For compilation of our P4 script for BMv2 target switch, we used P4c which includes a specialized back-end design specifically called 'P4c-BM2-SS.' To accelerate compilation, we used an Open Networking Docker image, which includes all of the necessary tools and dependencies for compiling the switch.p4 file. In our custom topology, we added the controller with the IP address '10.3.12.139' associated with our ONOS VM1. This allowed us to simulate and administer the custom network architecture with Mininet. This was performed using the following command:

```
def main():
Added Remote Controller in our Custom Topology
    net = Mininet(topo=TutorialTopo(), controller=RemoteController('onos', ip='10.3.12.139'))
Start the Mininet network
    net.start()
```

Chapter 08 – Results

To evaluate our P4T testbed for network solutions, we did a number of experiments and their findings are discussed in the following subsections: 'Bridge Connectivity Test' displays connection results, 'P4 Pipeline Configuration via ONOS' discusses pipeline impacts and 'Global Network Management Interface' and 'Topology Discovery' explain their separate functions.

8 Program leaf1 using P4Runtime

We used P4Runtime Shell that give support in connecting to a P4Runtime server in order to perform P4Runtime commands. Build or downloaded the Docker image manually with in our VM1, 10.3.12.139, where we placed ONOS instances:

```
$ git clone https://github.com/p4lang/p4runtime-shell
$ cd p4runtime-shell
$ docker build -t p4lang/p4runtime-sh .
```

We used the P4 Runtime shell to program leaf1. When we connected to a P4 Runtime server, it executed P4 Runtime commands, allowing us to conduct tasks: generating, reading, updating, and removing flow table entries. The P4 Runtime shell has two modes of operation: with or without a P4 pipeline configuration. The shell initially uses the P4 Runtime SetPipelineConfiguration RPC to send the given pipeline configuration to switch. Subsequently, it attempts to acquire switch's presently configured P4Info file, use to improve command readability by permitting the usage of P4Info names, enabling auto-completion, and validating command accuracy and completeness. An election ID is provided while connecting to a P4 Runtime server to enable the pipeline configuration and table entries. To route pipeline configuration acquired from our P4 code and to connect leaf 1 with P4 Runtime shell running on our ONOS VM1, following command is used:

```
$ util/p4rt-sh --grpc-addr 10.3.12.140:50001 --config p4src/build/p4info.txt,p4src/build/bmv2.json
--election-id 0,1
```

Our mininet container is executed remotely, therefore we used the '--grpc-addr of our Mininet and P4 VM2, 10.3.12.140:50001' to make this possible. Notably, TCP port 50001 corresponds to leaf 1, hosted gRPC server.

8.1 Static NDP Table Entries

We began the process of pinging two IPv6 hosts on the same subnet by having the hosts determine their respective MAC addresses using the Neighbor Discovery Protocol (NDP). We had h1 send an NDP Neighbor Solicitation (NS) message to discover h2's MAC address in the case when we attempted to ping h1 from h2, h2 from h1 and h1 from h3. Following receipt of the NDP NS message, h2 responded immediately with an NDP Neighbor Advertisement (NA) with its own MAC address. With both hosts now knowing each other's MAC addresses, we

exchanged ping packets. We repeated the same procedure for h1 to h3 ping. In Mininet P4 VM2 (10.3.12.140) we, inserted three static NDP entries in our hosts by following command:

```
ubuntu@ubuntu-vm: ~/
*** Attaching to Mininet CLI...
*** To detach press Ctrl-D (Mininet will keep running)
mininet> h1 ip -6 neigh replace 2001:1:1::B lladdr 00:00:00:00:00:1B dev h1-eth0
mininet> h2 ip -6 neigh replace 2001:1:1::A lladdr 00:00:00:00:00:1A dev h2-eth0
mininet> h3 ip -6 neigh replace 2001:1:1::C lladdr 00:00:00:00:00:1C dev h3-eth0
```

Figure 19: Static NDP Entries

This ping worked after inserting any P4Runtime table entry to forward these packets.

8.2 P4Runtime Table Entries

In our ONOS controller VM1 IP 10.3.12.139, we used P4Runtime shell to create and insert three table entries on l2_exact_table in leaf1. The sequence of table entries in P4Runtime shell are:

```
IPython: /p4runtime-sh
*** Welcome to the IPython shell for P4Runtime ***
P4Runtime sh >>> te = table_entry['IngressPipeImpl.l2_exact_table'](action='Ingr
            ...: essPipeImpl.set_egress_port')
            ...: te.match['hdr.ethernet.dst_addr'] = '00:00:00:00:00:1A'
            ...: te.action['port_num'] = '3'
            ...: te.insert()
            ...:
            ...: te = table_entry['IngressPipeImpl.l2_exact_table'](action='Ingr
            ...: essPipeImpl.set_egress_port')
            ...: te.match['hdr.ethernet.dst_addr'] = '00:00:00:00:00:1B'
            ...: te.action['port_num'] = '4'
            ...: te.insert()
            ...:
            ...: te = table_entry['IngressPipeImpl.l2_exact_table'](action='Ingr
            ...: essPipeImpl.set_egress_port')
            ...: te.match['hdr.ethernet.dst_addr'] = '00:00:00:00:00:1C'
            ...: te.action['port_num'] = '5'
            ...: te.insert()
```

Figure 20: Rules Insertion in P4Runtime Shell

P4Runtime shell internally transforms the value provided in the P4Info to a Protobuf based on the information in the P4Info, created a byte string. But sometimes legacy server doesn’t convert and rejects binary strings formatted using the canonical representation with following error:

P4RuntimeWriteException: Error(s) during Write:

*At index 0:INVALID_ARGUMENT, ‘Invalid bytestring format’

Return to the byte-padded format by entering the following command in the shell before inserting the above table entries:

P4Runtime sh >>> global_options["canonical_bytestrings"] = False

The match action is done with the destination MAC address to forward entries between the hosts. These entries direct the network device to route Ethernet frames depending on their destination addresses to certain ports. Each entry associates a specific Ethernet destination address with a specific port number, ensuring that incoming traffic is sent to the relevant host. After inserting these entries, when we went back to our Mininet VM2 referred to bridge connectivity test, the ping from h1 to h2 worked as shown in Figure 21.

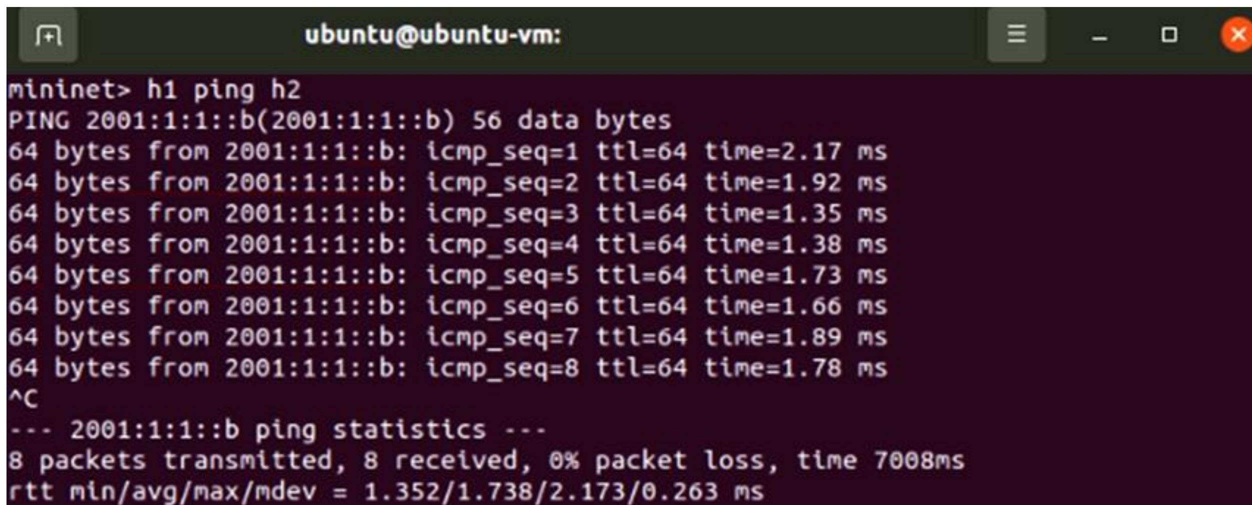


Figure 21: Ping h1 to h2

8.3 YANG

In network management configuration, we utilized a standard protocol gNMI (gRPC Network Management Interface) to work with YANG (Yet Another Next Generation) based data models. As a language, YANG provides protocol freedom, allowing for easy conversion into various encoding formats such as XML or JSON. Two key considerations come into play in the context of a YANG model:

To begin, the **Data Tree Organization** serves as the foundation. It defines the structure of the data tree, revealing important information about the pathways and attributes of leaf data types. This structure is the foundation for data representation and manipulation.

Second, the **Semantics of Leaf Nodes** must be considered. The description field of the model explains the meaning and purpose of particular leaf nodes, resulting in a clear comprehension of the data's intended usage. A YANG module is a self-contained unit that represents the smallest item that YANG tools can compile. Each module contains following components:

- **Boilerplate:** The module's specified namespace, a reference prefix for cross-module utilization, a description of its purpose, and version/revision history are all included in this section.
- **Identities and Derived Types:** The module includes identities and derived types that increase the model's capabilities and expressiveness.
- **Modular Groupings:** These groupings encourage model modularity and reusability, resulting in a more ordered and efficient structure.
- **Top-Level Container:** A top-level container is specified at the module's core to determine the structure of the data node tree, molding the overall representation of data. The example script is as follows:

```
// A module is a self-contained tree of nodes
module demo-port {
// YANG Boilerplate
yang-version "1";
namespace "https://opennetworking.org/yang/demo";
 prefix "demo-port";
description "Demo model for managing ports";
revision "2019-09-10" {
description "Initial version";
reference "1.0.0";
}
```

YANG defines several built-in types including binary, bits, boolean, decimal64, empty, enumeration, identityref, int8, int16, int32, int64, string, uint8, uint16, uint32, uint64, decimal64. An identity is a globally distinct and abstract entity that lacks data type categorization. These identities are important in identifying elements with clear and explicit semantics, and they have the possibility for hierarchical organization.

Derived types, on the other hand, provide a mechanism for enforcing constraints on built-in data types or other derived types, and are commonly constructed with the 'ypedef' construct. Let us use an example to demonstrate this:

```
// Identities and Typedefs
identity SPEED {
description "base type for port speeds";
}
identity SPEED_10GB {
base SPEED;
description "10 Gbps port speed";
}
typedef port-number {
type uint16 {
range 1..32;
}
description "New type for port number that ensure
the number is between 1 and 32, inclusive";
}
```

A grouping represents a reusable collection of nodes, encompassing containers and leaves, which can be incorporated within a container. It's important to note that a grouping, when defined or imported, doesn't inherently introduce any nodes to the module. In contrast, a leaf denotes a node holding a value, which may be of a built-in or derived type, and it doesn't have any child nodes.

```
container ports {
description "The root container for port configuration and state";
list port {
key "port-number";
description "List of ports on a switch";
leaf port-number {
type port-number;
description "Port number (maps to the front panel port of a switch);
also the key for the port list";}
// each individual will have the elements defined in the grouping
container config {
description "Configuration data for a port";
uses port-config; // reference to grouping above}
container state {
config false; // makes child nodes read-only
description "Read-only state for a port";
uses port-state; // reference to grouping above}
}
}
```

A container is a node that contains a collection of child nodes. Every module has a top-level or root container that serves as the foundation for its structure. A list, on the other hand, is a node that has numerous children of the same type. A key characteristic distinguishes each element in a list. Containers labeled 'configuration false' represent state data, which is read-only from the client's perspective. These containers are frequently used to communicate status information or statistics data.

8.3.1 Open Configuration

Open Configuration is a network provider consortium with the common goal of advancing computer networks into a dynamic and programmable architecture by embracing SDN principles such as declarative pattern, model determined administration, and operational procedures [18].

The fundamental goal of Open Configuration is to create a consistent collection of vendor-agnostic data models written in YANG. These models are built with practical needs drawn from use cases and feedback from numerous network operators, assuring their relevance and application in real-world network operations.

8.3.2 Global Network Management Interface

The OpenConfig initiative includes gNMI, which stands for gRPC Network Management Interface. It functions as a defined protocol stated in protocol buffers, created primarily to handle YANG-based data models and ease interactions via a specialized Network Management Interface. The primary operations include: Set Request and Response, Get Request and Response, and Subscription. Mininet VM2 is used to extract and read all configuration data from the 'leaft 1' Stratum switch. Stratum responds with openconfiguration.Device, the highest-level

message defined in openconfiguration.proto. The data stored in binary format according to the protobuf message structure is represented by this response. While the binary value itself is not human-readable, we used this to convert the protobuf message between binary and textual representations, making it more interpretative by executing following commands in our ONOS VM1:

```
// Retrieve the entire configuration
util/gnmi-cli --grpc-addr 10.3.12.140:50001 get /
// Retrieve configuration for leaf 1 Ethernet 3
util/gnmi-cli --grpc-addr 10.3.12.140:50001 get \
  /interfaces/interface[name=leaf1-eth3]/config
// Sample incoming unicast packet counters
util/gnmi-cli --grpc-addr 10.3.12.140:50001:50001 \
  --interval 1000 sub-sample \
  /interfaces/interface[name=leaf1-eth3]/state/counters/in-unicast-pkts
// Subscribe to changes in the operational status
util/gnmi-cli --grpc-addr 10.3.12.140:50001:50001 \
  sub-onchange \
  /interfaces/interface[name=leaf1-eth3]/state/oper-status
```

The gNMI CLI is used to interact with a gRPC (gNMI) service operating at 10.3.12.140:50001. The first command (get /) obtains the whole configuration of network. The second command (get /interfaces/interface[name=leaf1-eth3]/config) asks the leaf 1 Ethernet 3 network interface configuration information. The third command (sub-sample) takes 1000 (m)s samples of counter information for arriving uni-cast packets on leaf 1 Ethernet 3 port. The fourth command (sub-on change) subscribes to changes in the leaf1 eth 3 interface's operational status, getting notifications anytime the status changes. These commands make it easier to retrieve and monitor network configuration and status data using the gNMI protocol.

8.4 ONOS as a Control Plane

All the activities perform in this section are executed in controller VM1 that has IP 10.3.12.139. In this section, we'll look at how to use pre-existing ONOS services for link and host discovery in conjunction with P4. These services depend on the switches' capacity to transmit data plane packets to the controller and receive packets from it, facilitated through the use of packet IN and OUT mechanisms. Some basic code changes are required to provide this feature with the switch.p4 program. Changes to the pipe configuration Java implementation are also necessary in order for ONOS's built-in programs to use packet-in and packet-out features through P4Runtime. The process of integration has two parts:

8.4.1 Enable Packet I/O and Double-Check Link Discovery

We modified the P4 program to allow the controller to receive incoming packets and deliver packets to the switches. Test the ONOS link discovery feature by using the packet-in capability to receive link discovery packets from the switches. The switch.p4 file incorporates features designed to transmit custom metadata within P4Runtime Packet IN and OUT messages. The

@controller_header P4 annotation is used to define and annotate two special headers in our switch.p4 program. These headers have special functions:

- The initial switch ingress port information for a packet-in message is carried by the cpu_in_header_t header.
- A packet-out message's cpu_out_header_t header indicates the desired output port.

The P4Runtime agent expects the cpu_in_header_t head to be the first in the frame when it receives a packet from the switch CPU port in Stratum. It reads the P4Info file's controller packet metadata section to determine the number of bits to strip at the start of the frame and populates the relevant metadata field in the Packet IN message. This provides information such as the packet's entry port. Likewise, in the case of Stratum receiving a P4Runtime Packet OUT message, it proceeds to serialize and add a cpu_out_header_t header at the beginning of the frame. This header is generated based on the data derived from the metadata fields in the Packet OUT message. The frame is then conveyed to the pipeline parser for further processing. The following capabilities are already supported by the switch.p4 code:

When the ingress port corresponds to the CPU port, the system begins parsing the cpu_out_header. As a result, in the deparser, the cpu_in_header is designated as the primary header. This configuration also includes the creation of an Access Control List (ACL) table with ternary match fields, as well as an accompanying action designed to deliver or clone packets destined for the CPU port. This operation is responsible for generating packet INs. These characteristics allow the P4 program to handle packet input and output communication between the controller and the switches, streamlining control plane operations and improving network management capabilities. The Ingress Pipe Implementation block may be changed as follows to offer comprehensive packet-in/out capability with the aforementioned modification:

```
control IngressPipeImpl(inout parsed_headers_t hdr, inout local_metadata_t local_metadata, inout
standard_metadata_t standard_metadata) {
    apply {
        if (hdr.cpu_out.isValid()) {
            standard_metadata.egress_spec = hdr.cpu_out.egress_port;
            hdr.cpu_out.setInvalid();
            exit;
        }
        bool do_l3_l2 = true;
    }
}
```

The modified code snippet keeps the structure and logic intact and includes the necessary changes to set the egress port based on the header cpu out. If this header is valid, it sets the standard metadata egress specification to the egress port value in the cpu out header, marks this header as invalid using hdr.cpu_out.setInvalid(), and exits the control block using exit. The EgressPipeImpl logic checks to see if the packet should be sent to CPU port, for example, if an ingress on the ACL table with the action send or cpu clone matches.

```
control EgressPipeImpl(inout parsed_headers_t hdr, inout local_metadata_t local_metadata, inout
standard_metadata_t standard_metadata) {
    apply {
        if (standard_metadata.egress_port == CPU_PORT) {
            hdr.cpu_in.setValid();
            hdr.cpu_in.ingress_port = standard_metadata.ingress_port;
            exit;
        }
        // Remaining logic goes here
    }
}
```

The modified code snippet keeps the structure and logic intact and includes the necessary chang es to handle the packet egress port for the CPU port. If the standard_metadata.egress_port matches the CPU port, it sets the hdr.cpu_in header as valid using hdr.cpu_in.setValid() and assigns the ingress_port value from standard_metadata to hdr.cpu_in.ingress_port. Then, it exits the control block using exit.

gui2: ONOS online user interface, available at http://ONOS_IP>:8181/onos/ui drivers.bmv2: P4Runtime, gNMI, and gNOI-based BMv2/Stratum drivers. We followed these steps to construct the ONOS app, including the pipeconf:

```
$ make app-build
```

This will generate a binary file called SNAPSHOT.oar, which is used to install the application in the presently running ONOS instance. After activating the app, we observed the following entries in the ONOS log (using 'onos-log'), indicating the successful registration of the pipeconf and the beginning of various app components.

```
INFO  [PiPipeconfManager] New pipeconf registered: org.onosproject.ngsdn-tutorial (fingerprint=...)
INFO  [MainComponent] Started
```

Alternatively, to display the list of registered pipeconfs, execute the 'onos-cli' command to:

```
onos> pipeconfs
```

The task of instructing ONOS on how to create contact with the four linked switches and establish control involves initializing the system with a file called mininet/netcfg.json, which contains information of gRPC address and port information associated with each Stratum device, the designated ONOS driver (stratum-bmv2) for each device, and the specific pipeconf choice for each device, as defined in our PipeconfigurationLoader.java. We opened a terminal window and entered the following command to push the netcfg.json file to ONOS that triggered discovery and P4 switches configuration.

```
$ make netcfg
```

To validate previously push network configuration we open ONOS CLI and enter the command:

```
onos> netcfg
```

To load the app into ONOS and activate it, we used the command:

```
$ make app-reload
```

The program has components, specifically 'lldpprovider' and 'hostprovider' which are in charge of LLDP link discovery and host discovery, respectively. A substantial portion of it involved the development of a new software required for integrating ONOS with the given P4 application. This app focuses on pipe configuration implementation and includes three key files: 'PipeconfLoader.java' which is responsible for registering the pipe configuration during app initialization; 'InterpreterImpl.java' which acts as a pipeline interpreter in terms of driver behavior and 'PipelinerImpl.java' which is responsible for implementing the Pipe liner driver behavior. These files play important role for guaranteeing ONOS's flawless integration with the P4 program, as well as the proper functioning and coordination of pipeline interpretation and execution.

The translation of ONOS packet I/O representations into a format suitable with the P4 pipeline is an essential aspect of the Pipe line Interpreter ONOS driver behavior. For ONOS built-in apps to properly establish the output port of a packet OUT and access the first ingress port of a packet IN, services such as link and host discovery must be present. This method involves the submission of network settings in JSON format to ONOS in order to begin device and link discovery. We executed the following command to validate the successful detection of all devices and connections using the ONOS CLI:

```
onos> devices -s
```

The output contains information about the IDs, availability, role, type, and driver of found devices as shown in Figure 22.

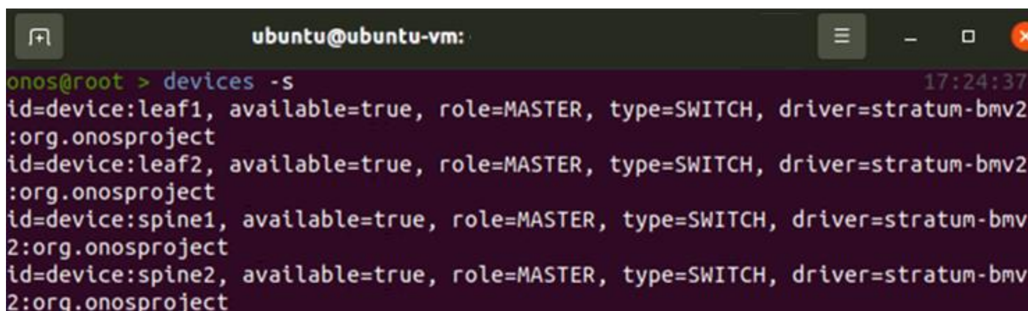


Figure 22: Detection of Devices in ONOS CLI

Each line represents a detected device and contains information such as the device ID, availability status (true or false), and role (e.g., MASTER), and device type (e.g., SWITCH), and driver (e.g., stratum-bmv2:org.onosproject.ngsdn-tutorial).

```
onos> links
```

This command is used to display information about the discovered links between devices. Here is the output of it:

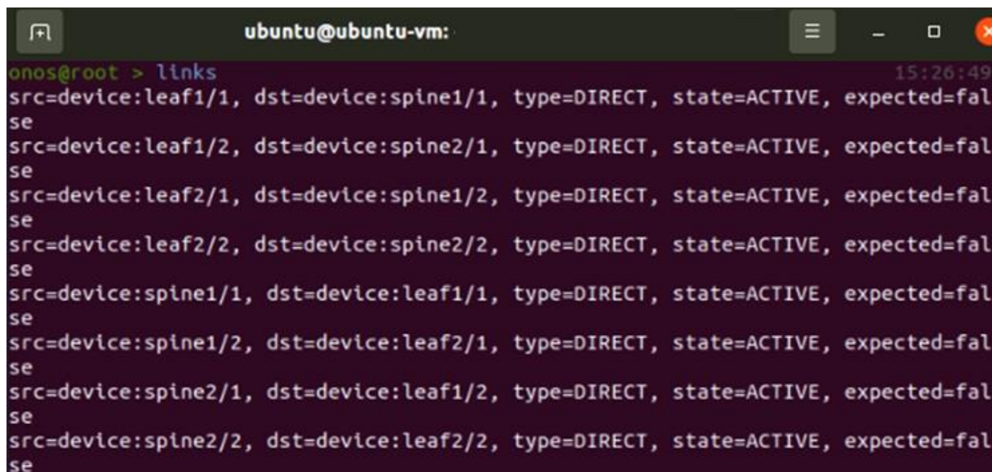


Figure 23: Links on Devices

Each line represents a link between two devices and provides details such as the source device (src), destination device (dst), link type (type), link state (state), and expected status (expected).

Each device have five flow rules. To display all flow rules placed on device leaf1, we typed:

```
onos> flows -s any device:leaf1
```

The output result contains details about the flow rules implemented on device: leaf 1, the device ID, the number of flow rules (flowRuleCount), and detailed information about each flow rule. These flow rules are generated by the built-in applications host and lldp provider depending on the flow objectives. Pipe configuration transforms these flow objectives into flow rules via its implementation of the Pipe liner behavior (PipelinerImpelmentation.java). These flow rules define specific criteria for matching packets using identified header fields in ONOS, such as Ethernet and ICMPV6 type, etc.

The host provider app enables host discovery by capturing ARP packets and NDP packets. These packets are then cloned to the controller. Similarly, the lldp provider app generates flow objectives to intercept LLDP and BDDP (Broadcast Domain Discovery Protocol) packets. These packets are periodically sent to all device ports as P4Runtime packet OUTs, enabling auto discovery of link. Every flow rule in the system uniformly utilizes the P4 action 'clone_to_cpu(),' which in turn invokes a v1model-specific primitive for configuring the clone session ID (as defined in switch.p4).

The pipe liner of the pipe configuration constructs a CLONE group to generate P4Runtime packet IN messages for matching packets. This group is internally converted into a P4Runtime Clone Session Entry that translates the Clone Session ID of CPU to a set of ports, in this instance the CPU port. To view all ONOS groups installed on leaf 1:

```
onos> groups any device:leaf1
```

ONOS gathers link data by collecting port counts for each device on a regular basis. Internally, ONOS reads info about port, including counters, via gNMI. Enabling packet IN/OUT capabilities in the pipeline interpreter facilitates not just link discovery but also enables the built-in host provider programme for host discovery. For the new app to populate the P4 pipeline's bridging tables, a host discovery service is necessary. The bridging tables operate by directing packets according to the Ethernet destination address. The host provider application functions by overseeing incoming ARP and NDP packets within the switch, deducing the host's connection point based on the information carried in the packet's message. Subsequently, various ONOS applications, including the recently developed one, can actively monitor host-related events, facilitating access to essential data such as IP addresses, MAC addresses, and host locations.

8.4.2 L2 Bridging and Host Discovery

We added host discovery capabilities to the P4 program, allowing the controller to understand and manage the hosts connected to the switches. In the P4 program, enabled L2 bridging to allow packet forwarding across hosts connected to different switches based on their MAC addresses. A new component app named L2BridgingComponent.java is created to implement L2 bridging capabilities. This app provided the logic and configurations required to allow L2 bridging.

The bridging feature is implemented by the L2BridgingComponent.java app, which define forwarding rules depending on Ethernet destination addresses. It populate the bridging tables in the P4 pipeline with information collected from the host provider app. Based on the Ethernet destination address, the program process incoming packets and select the suitable output port. It then forward the packets in the appropriate order to allow L2 bridging within the network.

Internal Device and Host Listener are dual event listeners defined at the end of the L2 Bridging Component class in our code. These listeners deal with device and host events, respectively. When a device event occurs, such as the connecting of a new switch, the Internal Device Listener is activated. This listener invokes the setUpDevice() function, in control for generating multicast groups for all host facing ports and adding flow rules that point to these groups into the l2_ternary_table.

The Internal Host Listener, on the other hand, is activated when a host event occurs like discovery of a new host. The learnHost() function is called by this listener and is responsible for actions such as adding unicast L2 entries depending on the recently identified host location. The L2 Bridging Component application dynamically respond to device and host events by using these event listeners and the appropriate methods, ensuring that multicast groups, flow rules, and L2 entries are properly set up and updated to support L2 bridging capabilities inside the network.

The aforementioned methods are also performed after component activation to ensure support for reloading the app implementation. At the moment of activation, the activate() method and setUpAllDevices() function are called to handle all known devices and hosts inside ONOS.

The 'setUpAllDevices()' function is in responsible for configuring each device after it has been activated. This includes operations like building multicast groups for ports facing hosts and adding flow rules to the l2_ternary_table to steer traffic to these groups.

To keep things simple, the broadcast realm is limited to a single device. This constraint implies that packet replication is only permitted among ports on the same leaf switch, excluding multicast group ports linked to spine switches.

Implementation of the L2 Bridging Component.Java assumes that all hosts in a subnet are linked to the same leaf switch and that two IPv6 subnets cannot be setup on separate leaves. L2 bridging is only permitted between hosts connected to the same leaf switch. To examine all installed flow rules on device leaf 1, executed the following command:

```
onos> flows -s any device:leaf1
```

It displayed the flow rules installed on device leaf1. The flow rules are associated with the L2BridgingComponent and stored in the l2_ternary_table. There are two flow rules listed:

Rule: It has a priority of 10 and a selector that matches the Ethernet destination address 0x333300000000 with a mask of 0xffff00000000. The treatment applied is set_multicast_group(gid=0xff), indicating that packets matching this rule will be directed to the multicast group with group ID 0xff.

Rule: This rule also has a priority of 10 and a selector that matches the Ethernet destination address 0xffffffffffff with a mask of 0xffffffffffff. The treatment applied is the same as in Rule 4, directing packets to the multicast group with group ID 0xff.

These flow rules are in charge of providing L2 bridging capabilities and allowing multicast forwarding for the Ethernet destination addresses given. The groups command is used to display multicast groups as well. To display groups on leaf1:

```
onos> groups any device:leaf1
```

In the displayed output, the groups command shows the groups configured on device leaf1. There are two groups listed:

Group ID 0x63: This group is of type CLONE and is associated with the org.onosproject.core app. It has one bucket (bucket ID 1) with an action of Output:Controller, indicating that packets replicated to this group will be sent to the controller.

Group ID 0xff: This group is of type ALL and is associated with the org.onosproject.ngsdn-tutorial app. It has four buckets (bucket IDs 1-4) with actions OUTPUT:3, OUTPUT:4, OUTPUT:5, and OUTPUT:6. These actions specify the output ports for multicast forwarding, indicating that packets sent to this group will be forwarded to ports 3, 4, 5, and 6. These groups are used in the L2 bridging implementation to handle packet replication and multicast forwarding.

The generated app establishes a new group in ONOS named the ALL group. P4Runtime Multicast Group Entry corresponds to ALL groups in ONOS to broadcast NDP NS packets to all ports linked with hosts.

8.4.3 Topology Discovery

Topology discovery involves identifying connections of hosts and BMv2 switches in our 2x2 fabric topology. Hosts are defined in Linux containers whereas switches operate as switch daemons that interact with SDN controller via OF protocol. The links in Figure 24 indicate the network physical and logical connectivity. The ONOS UI is used to discover our topology. It collects and develops a full picture of the network's structure as network parts communicate and share information. ONOS-managed apps are responsible for Ethernet bridging at the leaf and IP routing throughout the spine network. They subscribe to packet IN and port status changes in P4 Runtime and gNMI, and then establish the appropriate flow rules and action groups.

The host provider app enables host discovery by capturing ARP and NDP packets and cloning to controller. Similarly, the LLDP provider application develops flow objectives for capturing LLDP and Broadcast Domain Discovery Protocol (BDDP) packets, facilitating automatic link discovery. Enabling packet IN/OUT feature within the pipeline interpreter not only simplifies link discovery but also improves performance and additionally allows the built-in host provider program to execute host discovery which is essential for populating P4 pipeline tables, used to route packets based on their Ethernet destination addresses.

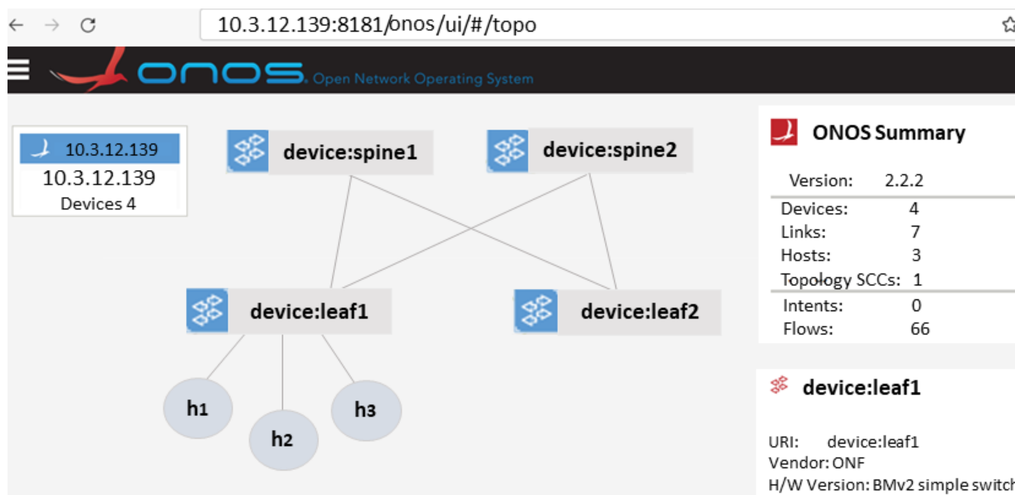


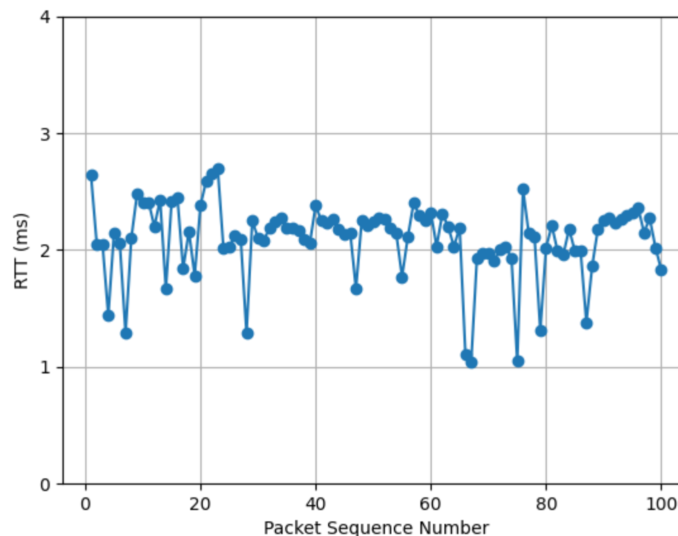
Figure 24: Topology Discovery

The host provider application works by monitoring ARP and NDP incoming packets at the switch end and discovering the connection of host location from message information in the packet. Other ONOS apps then monitor the events related to host and extract either IP or MAC addresses and host locations.

To take advantage of services provided by link and host discovery, ONOS built-in application uses the output port of a packet OUT operation and accesses the original ingress port of a packet IN operation. The network configuration JSON submitted to ONOS in order to begin device and link discovery. The Pipe line Interpreter ONOS driver behaviour is utilized to translate the ONOS packet IN/OUT into the format compatible to P4 pipeline.

8.4.4 Performance Analysis

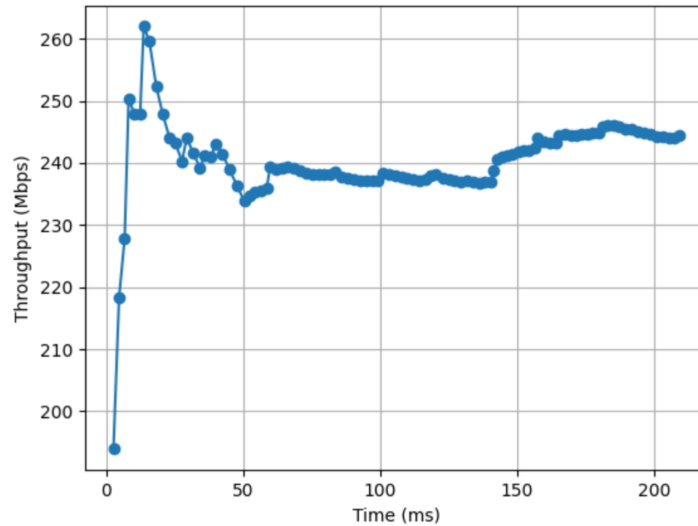
As part of our thesis, we also did a performance analysis of Round Trip Time and Throughput. The graph in Fig. 10 displays the RTT of network packets delivered from host h1 to h2. It is the amount of time a packets takes to transit from source to destination and return. The x-axis has packet sequence number while y-axis has RTT in milliseconds. Data points are represented by circular symbols connected by lines. RTT is initially high due to route establishment or propagation delay but drops after around 10s. Some spikes, particularly the one at 15s shows network condition change owing to congestion caused by higher traffic volume. Following then, the RTT falls and stabilizes at about 1.4ms around 20s indicating a restoration to network stability with minimal fluctuations which meets the requirement of our application.



Graph 1: RTT Analysis for h1 to h2 Ping

The throughput graph in Fig. 11 shows the performance of the network link over time, with time on the x-axis and TP in megabits/sec on the y-axis. Initially, it has low TP indicating that network communication is barely getting started. Then it shows a rapid increases due to varying

levels of congestion and packet routing ultimately stabilizing at 200 Mbps. This is characteristic of underutilized network links, where low starting traffic progressively increases throughput until the link reaches its maximum capacity. Throughput is affected by network link bandwidth, traffic type, and congestion. The consistent throughput of 200 Mbps in this scenario reflects the network link's 200 Mbps bandwidth, indicates the effective performance of the network link.



Graph 2: TP Analysis of h1 to h2 Ping

Chapter 09 - Future Work

In P4T research, we examined SDN and NFV existing solutions and introduced a novel idea of testbed centered on data plane programming by utilizing the P4 programming language and integrating an ONOS controller. In addition, we showed various experiments of combining ONOS, an SDN controller, with P4, which provides enterprises with a more efficient approach to network architecture.

In the future, we will use P4T to investigate real time video packet classification to assess and optimize network performance for video streaming applications. This testbed will enable us to delve deeper into P4-based networking, explore SDN integration with cutting-edge technologies such as 5G and IoT, and further enhance network security procedures. We hope to stimulate innovation and produce scalable, efficient, and secure network solutions that match with the increasing demands of our digital era by employing P4T.

Chapter 10 - References

[1] Buck Chungy, Chien-Chao Tseng, Jim Hao Cheny, Joe Mambretti, "P4MT: Multi-Tenant Support Prototype for International P4 Testbed," ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), 18 November 2019.

[2] Sandor Laki, Peter Voros, and Ferenc Fejes, "Towards an AQM Evaluation Testbed with P4 and DPDK," Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos, August 2019, 19 August 2019, Pages 148–150.

[3] Eder Ollora Zaballa, David Franco, Michael S. Berger, and Marivi Higuero, "A perspective on P4-based data and control plane modularity for network automation," EuroP4'20: Proceedings of the 3rd P4 Workshop in Europe, 01 December 2020, Pages 59-61.

[4] Damu Ding, Marco Savi, Federico Pederzolli, and Domenico Siracusa, "Design and Development of Network Monitoring Strategies in P4-enabled Programmable Switches," NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium, 25 April 2022, Pages 1–6.

[5] Kulkarni Manasa, Goswami Bhargavi, and Paulose Joy, "P4 based Load Balancing Strategies for Large Scale Software-Defined Networks," IEEE 2022 Second International Conference on Advances in Electrical, Computing, Communication, and Sustainable Technologies (ICAECT), April 2022.

[6] Bob Lantz, Brian O'Connor, "A Mininet-based Virtual Testbed for Distributed SDN Development," ACM SIGCOMM Computer Communication Review, Vol. 45, No. 4, pp 365–366, 17 Aug 2015.

[7] Thomas Luinaud; Thibaut Stimpfling; Jeferson Santiago da Silva; Yvon Savaria; J.M. Pierre Langlois, "Bridging the Gap: FPGAs as Programmable Switches," IEEE 21st International Conference on High-Performance Switching and Routing (HPSR), 22 May 2020.

[8] Brian O'Connor, Yi Tseng, Maximilian Pudelko, Carmelo Cascone, Abhilash Endurthi, You Wang, Alireza Ghaffarkhah, Devjit Gopalpur, Tom Everman, Tomek Madejski, Jim Wanderer, and Amin Vahdat, "Using P4 on Fixed Pipeline and Programmable Stratum Switches," 2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), November 2019, pages = 1-2.

[9] Sreekanth Sasidharan and Saurav Kanti Chandra, "Defining Future SDN-based Network Management Systems: Characterization and Approach," IEEE Fifth International Conference on Computing, Communications, and Networking Technologies (ICCCNT), November 2014, pages = 1-5.

[10] Jose Miguel-Alonso, "A Research Review of OpenFlow for Datacenter Networking," Journal: IEEE Access, volume = 11, pages = 770-786, December 2022.

[11] T. Aditya and A. David Donald and G. Thippanna and M. Mohsina Kousar and T. Murali, "NFV and SDN: A New Era of Network Agility and Flexibility," Journal = International Journal of Advanced Research in Science, Communication, and Technology (IJARSCT), volume = 3, number = 2, March 2023.

[12] Hend Abdelgader Eissa and Kenz A. Bozed and Hadil Younis, "Software Defined Networking," 19th International Conference on Sciences and Techniques of Automatic Control and Computer Engineering (STA), May 2019.

[13] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, Guru Parulkar, “ONOS: Towards an Open Distributed SDN OS,” ACM Conferences, COMM Proceedings, 22 Aug 2014.

[14] Luis F. Gonzalez, Ivan Vidal, Francisco Valera, Borja Nogales, Victor Sanchez-Aguero, and Diego R. Lopez, “Transport-Layer Limitations for NFV Orchestration in Resource-Constrained Aerial Networks,” Journal, Sensors, Volume 19, Issue 23, 28 November 2019.

[15] Wenfeng Xia, Yonggang Wen, Senior Member, IEEE, Chuan Heng Foh, Senior Member, Dusit Niyato, and Haiyong Xie, “A Survey on Software-Defined Networking,” IEEE Communication Surveys & Tutorials, Vol. 17, No. 1, First Quarter 2015.

[16] ONOS - Open Networking Operating System: https://opennetworking.org/onos/

[17] Programmer Sought - Article: https://programmersought.com/article/76604747557/

[18] ONOS Project Wiki: https://wiki.onosproject.org/

[19] Stratum - Open Networking Foundation: https://opennetworking.org/stratum/

[20] Yi-s-gNMI-tool on GitHub: https://github.com/Yi-Tseng/Yi-s-gNMI-tool

[21] Network Functions Virtualization (NFV) - TechTarget:
https://www.techtarget.com/searchnetworking/definition/network-functions-virtualization-NFV

[22] Open Compute Project - Switch Abstraction Interface (SAI):
https://github.com/opencomputeproject/SAI

[23] Beginner's Guide to Network Service and VNF Forwarding Graph in NFV:
https://telcocloudbridge.com/blog/beginners-guide-to-network-service-and-vnf-forwarding-graph-in-nfv/

[24] Huawei Enterprise Documentation:
https://support.huawei.com/enterprise/en/doc/EDOC1100202532

[25] P4 Programming - Plvision Blog: https://plvision.eu/blog/sdn/p4-programming-future-sdn

[26] Protocol Buffers (Protobuf) on GitHub: https://github.com/protocolbuffers/protobuf

[27] White Box Solution - 6 Examples of NFV Use Cases: https://www.whiteboxsolution.com/blog/6-examples-of-nfv-use-cases/

[28] Intraway Blog - NFV Benefits: https://www.intraway.com/blog/nfv-benefits

[29] RCR Wireless News - Challenges Facing NFV: https://www.rcrwireless.com/20170803/fundamentals/five-challenges-facing-nfv-tag27-tag99

[30] Benzinga - SDN and NFV Market Analysis: https://www.benzinga.com/pressreleases/23/09/34202750/sdn-and-nfv-market-recent-innovations-and-upcoming-trends-analysis-by-2029

[31] P4 – Programming Future SDN: https://plvision.eu/blog/sdn/p4-programming-future-sdn

[32] P4 - Programming Protocol-Independent Packet Processors: https://p4.org/p4-spec/docs/PNA.html#:~:text=P4%20is%20a%20domain%2Dspecific,programmable%20blocks%20within%20that%20pipeline

[33] Volansys - P4 Runtime and the Future of SDN: https://www.volansys.com/blog/p4-runtime-future-of-sdn

[34] P4 Behavioral Model on GitHub: https://github.com/p4lang/behavioral-model

[35] Software-Defined Networking - Open Networking Foundation: https://opennetworking.org/sdn-resources/whitepapers/software-defined-networking-the-new-norm-for-networks/#:~:text=The%20ONF%20is%20a%20non,planes%20of%20supported%20network%20devices.

[36] Stefanini - Understanding Software-Defined Networking: https://stefanini.com/en/insights/news/what-is-sdn-understand-the-concept-of-software-defined-networking

[37] P4 Forum - P4 Architecture: https://forum.p4.org/t/p4-architecture/246/2

[38] P4 Language Specification (P4-16) (PDF): https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.pdf

[42] P4 Switch on GitHub: https://github.com/p4lang/switch

[40] O'Reilly - SDN: Software-Defined Networks: https://www.oreilly.com/library/view/sdn-software-defined/9781449342425/ch04.html

[41] RouterFreak - SDN Use Cases: https://www.routerfreak.com/software-defined-network-use-cases-from-the-real-world/

[44] TechTarget - ONOS Definition: https://www.techtarget.com/searchnetworking/definition/ONOS-Open-Network-Operating-System

[45] OpenFlowJ-Loxi on GitHub: https://github.com/floodlight/loxigen/wiki/OpenFlowJ-Loxi