A DEPENDENCY ANALYZING SYSTEM FOR COMMUNICATION ACTIVITIES IN NETWORK CONSTRUCTION EXERCISES USING TREMA

HLWAM MAINT HTET

UNIVERSITY OF COMPUTER STUDIES, YANGON

JULY, 2024

A Dependency Analyzing System for Communication Activities in Network Construction Exercises using Trema

Hlwam Maint Htet

University of Computer Studies, Yangon

A thesis submitted to the University of Computer Studies, Yangon in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

July, 2024

Statement of Originality

I hereby certify that the work embodied in this thesis is the result of original research and has not been submitted for a higher degree to any other University or Institution.

…..……………………………   ………….......…………………………

Date   Hlwam Maint Htet

ACKNOWLEDGEMENTS

First and foremost, I am deeply grateful to the Ministry of Science and Technology for granting me the opportunity to pursue my Ph.D. by providing the support that enabled me to conduct my research at the University of Computer Studies, Yangon, Myanmar.

I would like to extend my heartfelt thanks to Dr. Mie Mie Khin, the Rector of the University of Computer Studies, Yangon, for allowing me to develop this thesis and for her general support during my study period.

Special thanks are due to Dr. Mie Mie Thet Thwin, the former Rector of the University of Computer Studies, Yangon, for her overall support throughout my Ph.D. coursework, seminars, and thesis.

I am particularly grateful to my supervisor, Dr. Amy Tun, Professor at the Faculty of Computer Systems and Technologies, University of Computer Studies, Yangon, for her patience, motivation, constant encouragement, and invaluable guidance and suggestions throughout the process of writing both my papers and this thesis.

I owe a debt of gratitude to my former supervisor, Dr. Khin Than Mya, Professor at the University of Computer Studies, Yangon, for her guidance, advice, and encouragement throughout my Ph.D. studies.

I sincerely would like to express my deepest gratitude to my former co-supervisor and advisor, Dr. Yuichiro Tateiwa, Associate Professor at Nagoya Institute of Technologies, for providing the foundation of this research, as well as for his helpful guidance and suggestions throughout our research collaboration.

My sincere gratitude and special thanks go to Dr. Si Si Mar Win, Course-Coordinator of the Ph.D. 11th batch, for her excellent guidance, care, and support throughout my Ph.D. journey.

I would also like to express my respectful gratitude to Dr. Tin Thein Thwel, Professor and Former Course-Coordinator of the Ph.D. 11th batch at the University of Computer Studies, for her support and encouragement during various stages of my research.

My heartfelt thanks and respect are extended to Dr. Sabai Phyu, Professor and Former Course-Coordinator of the Ph.D. 11th batch at the University of Computer Studies, Yangon, for her general guidance, kindness, and encouragement throughout my Ph.D. studies. She also provided patient teaching during my coursework.

Special thanks go to Professor Daw Aye Aye Khine, Head of the English Department, for her patience and valuable suggestions, which greatly improved my thesis writing.

I am grateful to U Min Htut, Deputy Minister of the Ministry of Planning and Finance, and former Director General of the Internal Revenue Department, for his permission to attend the Ph.D. course in line with my duties.

I would also like to thank my superior, U Tin Maung Thant, Deputy Director General of the Internal Revenue Department and former Director of the IT Directorate, for his kind support, encouragement, and permission to pursue my research throughout this journey.

My appreciation extends to all my Ph.D. 11th batch colleagues for their kindness, motivation, knowledge sharing, constant care, and the unforgettable memories we created over the past five years.

Lastly, my deepest gratitude and love go to my parents and family members for their eternal love, kindness, patience, physical and mental support, and constant encouragement throughout my life. Without their efforts, none of this would have been possible.

ABSTRACT

Nowadays, networking and virtualization technology has been developing in momentum. Software Defined Networking (SDN) has been popular for research and innovation. Universities and research labs are the basic points for innovation because innovation by academia and research organizations can accelerate the rate of change in industries. SDN construction exercises have been developed in e-Learning. Software-Defined Networking (SDN) is a networking approach that decouples the control plane from the data plane, allowing centralized network management. It remains popular in the research field for its benefits that researchers continue to explore various aspects such as: network security, traffic management, network virtualization, edge computing, machine learning and so forth. SDN's flexibility and programmability keep it relevant for emerging technologies and innovative network solutions.

When performing network construction exercises, novice learners cannot understand the behavior of their network and fail to satisfy the requirements for the network reachability of communication data. In this system, learners construct SDN network construction exercises by using Trema and OpenFlow Protocol is used for communication between controllers and switches. Here, some learners cannot find their bugs from their settings due to the reasons such as ping cannot find delivery routes including switches, switches have no function to log rules used for choosing output ports for packets, and Trema cannot find execution statements used for setting rules to switches. To satisfy these problems, learners need help and the system will provide analysis results for learners in visual way so that they can narrow down executed statements that cause incorrect communication. This dissertation presents a Dependency Analyzing System for Communication Activities in Network Construction Exercises using Trema. It includes four main modules: constructing Software Defined Network (SDN) Construction Exercises Using Trema, collecting data packets from constructed virtual network, collecting executed statements in controller program, and giving the analysis results to learners so that they can narrow down their visualizing packet location and executing statement information in chronological order.

TABLES OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF EQUATIONS

CHAPTER 1

INTRODUCTION

With the rapid growth of science and technology, virtualization technology has been developed in momentum and it is still a useful and efficient technology in networking era. SDN is a kind of virtualization technology [36]. With the spread of SDN, network construction exercises using SDN have been developed in some educational institutions. The experience of basic network construction is useful for not only network administrators but also for network application programmers and network system designers. In this work, when the learners construct SDN with Trema as exercises, novice learners cannot understand the behavior of their networks and cannot find their bugs from their settings. Therefore, the system can provide analysis results that help learners to narrow down missettings in SDN construction exercises.

SDN, or Software-Defined Networking, refers to a paradigm in network management and operations where the control plane is decoupled from the data plane. This separation allows network administrators to dynamically manage network traffic and resources through software-based controllers rather than relying on traditional hardware-based network devices (like routers and switches) with embedded control planes. The key concepts of SDN include:

Centralized Control: SDN centralizes network intelligence in a software-based controller, which makes decisions about how data traffic should be forwarded across the network.

Programmability: SDN enables programmable network behavior through APIs (Application Programming Interfaces), allowing network administrators and developers to automate network management tasks and implement policies more efficiently.

Virtualization: SDN facilitates network virtualization, where multiple virtual networks can run on the same physical infrastructure, enhancing flexibility and resource utilization [19].

Open Standards: SDN often leverages open standards and protocols (like OpenFlow) to ensure interoperability between different vendors' hardware and software components.

Dynamic Provisioning: With SDN, networks can be dynamically provisioned and adjusted based on changing traffic patterns or application requirements, improving scalability and responsiveness.

SDN has gained popularity due to its potential to reduce network operational costs, improve agility in deploying new services, and enhance overall network performance and security through centralized management and automation.

1.1 Problem Definition

Trema [50]is a framework designed for developing OpenFlow controllers using Ruby and C. Open vSwitch is a virtual switch that adheres to the OpenFlow standard. In the SDN construction exercises discussed in this study, students use Trema to implement controller programs and set up software-defined networks on their PCs, utilizing these controllers and Open vSwitches. The exercises include configuring data plane networks, creating shell scripts to build these networks, designing complete network topologies, and ensuring requirements for reachability and communication routes are met. Students' networks must fulfill these requirements. However, students often struggle to solve the problems because they fail to identify and correct errors in their controller programs.

There are causal links between packet transmissions in the data plane, transmissions in the control plane, and the execution of statements in the controller program. For instance, when a switch in the data plane receives a packet, it queries the controller on how to handle the packet. The controller receives this query from the control plane and processes it by executing specific statements.

Students are expected to debug their controller programs by analyzing these causal connections. However, some learners are unable to identify misconfigurations in their controllers due to the following reasons-

1. ping cannot find delivery routes including switches (Learners cannot narrow down switches that cause incorrect communication)
2. Switches have no function to log rules (PacketOut and flow tables) used for choosing output ports for packets (Learners cannot identify rules that cause incorrect communication)

3. Trema cannot find execution statements (for sending PacketOut and FlowMod) used for setting rules to switches. (Learners cannot narrow down execution statements that cause incorrect communication)

1.2 Terminology

To obtain the student’s network construction exercise and analyze them and generate event visualizer, the following general terminologies are required.

1. SDN
2. OpenFlow
3. Trema Open Flow Controller Framework
4. PlantUML

1.3 Motivation

The motivation for this research in building network construction exercises stems from the need to address several key challenges in the field of network engineering education. As networks become more complex and software-defined networking (SDN) gains prominence, there is a pressing need for effective training tools that can help learners grasp the intricacies of network behavior and dependencies. While building software defined network construction exercises in Universities and research labs, some learners who learn SDN network construction exercises as beginners fail to debug their controller programs while building with the controller Trema. So, a support tool is proposed that collects information of execution statements, in the controller programs and packets in their networks, finding relations between them, generates communication events logs and then visualize with sequence diagrams. Then, learners can see errors in their program and correct them by analyzing the resulted diagrams. Specifically, this research aims to:

Enhance Learning Outcomes: Provide novice learners with practical, hands-on experience in constructing and managing SDN networks, thereby improving their understanding and skills.

Address Knowledge Gaps: Fill the gaps in current educational resources by offering comprehensive exercises that cover both theoretical and practical aspects of network construction.

Improve Problem-Solving Skills: Develop exercises that encourage critical thinking and problem-solving, enabling learners to identify and address network behavior issues effectively.

Promote Innovation: Encourage the development and testing of new network architectures and algorithms in a controlled, educational environment.

Support Collaboration: Facilitate collaborative learning and knowledge sharing among students, fostering a community of practice in network engineering.

1.4 Objectives

The main objective for this research is to help learners while they construct SDN construction exercises by analyzing dependency in communication activities to narrow down misettings and the other objectives of this research area are as follows;

1 To implement a function to compute the dependency between communication routes and flow tables and executed controllers

2 To implement a function that visualizes the dependency so that learners can understand it easily

3 To evaluate the performance (time, CPU, Memory) while the system computes the dependency

4 To evaluate the effectiveness that the system helps learners find bugs

1.5 Contributions

The contributions of this research are as follows:

1. Finding relations between input packets and their output ports and OpenFlow messages by imitating switch actions to treat OpenFlow messages and choose output ports that helps learners to identify rules that cause incorrect communication.
2. Estimating delivery routes based on collected packets and the relations of item 1 that helps learners to narrow down switches that cause incorrect communication.
3. Finding relations between executed statements and sent OpenFlow messages by logging controller and capturing the messages that helps learners to

narrow down executed statements that cause incorrect communication and showing the analysis result with the sequence diagrams.

1.6 Organization

This dissertation is organized into seven chapters.

- Chapter 1 introduces the thesis by outlining the motivation, problem statements, objectives, key focuses, and contributions of the research.
- Chapter 2 reviews challenges and elements related to network behavior issues and communication activity dependencies, especially as novice learners build SDN networks, based on existing literature.
- Chapter 3 explores the theoretical background of software-defined networking, OpenFlow Controllers, the Trema OpenFlow controller used in this study, and network behavioral issues in creating SDN networks.
- Chapter 4 discusses the architecture of the proposed system and the algorithms developed to analyze communication activity dependencies.
- Chapter 5 delves into the design and implementation of the proposed system.
- Chapter 6 details the evaluation of the experimental results.
- Chapter 7 concludes the research by summarizing the findings and suggesting future research directions.

CHAPTER 2

LITERATURE REVIEWS

This chapter provides an overview of the relevant to the SDN network construction, OpenFlow controller framework and learning status of novice learners. The advent of Software-Defined Networking (SDN) has revolutionized the networking landscape, offering unparalleled flexibility, scalability, and control. Central to the educational and research domains is the development and analysis of SDN networks, particularly through construction exercises. This literature review delves into the concept of a Dependency Analyzing System for Communication Activities in SDN Network Construction Exercises, exploring existing research, methodologies, and gaps in the literature.

2.1 Software Defined Networking (SDN)

SDN decouples the network control plane from the data plane, enabling centralized management of network resources. The control plane makes decisions about where traffic is sent, while the data plane forwards traffic to the selected destination. This separation simplifies network management and allows for more dynamic and programmable networks.

Siamak Azodolmolky's paper "Software Defined Networking with OpenFlow" [36] is a pivotal work that delves into the transformative impact of SDN and OpenFlow on modern networking paradigms. The paper provides a comprehensive overview of SDN, emphasizing its decoupling of the control plane from the data plane, which allows for more flexible and efficient network management. OpenFlow, as a prominent protocol in the SDN ecosystem, is highlighted for its role in enabling this separation, allowing network administrators to program the behavior of the data plane directly through a standardized interface. Azodolmolky elaborates on how OpenFlow facilitates fine-grained traffic control, dynamic policy enforcement, and real-time network reconfiguration, which are essential for addressing the evolving demands of contemporary networks.

The literature review within the paper explores various implementations and use cases of SDN and OpenFlow across different environments, including data centers, enterprise networks, and wide-area networks. It discusses the benefits of SDN with OpenFlow, such as improved network agility, reduced operational costs, and enhanced scalability. Furthermore, Azodolmolky examines the challenges associated with SDN adoption, including issues related to scalability, security, and interoperability. The review synthesizes findings from multiple studies, presenting a balanced view of the potential and limitations of SDN with OpenFlow. Overall, the paper serves as a crucial resource for researchers and practitioners interested in leveraging SDN and OpenFlow to innovate and optimize network infrastructure.

2.1.1 Existing Tools and Frameworks for SDN in Education

Several tools and frameworks are commonly used in SDN education, including Mininet, POX [43], and Floodlight. Mininet, [30] for instance, allows users to create a virtual network environment on their computers, providing a realistic and cost-effective way to experiment with network topologies and protocols. POX and Floodlight are open-source SDN controllers that offer simple interfaces for developing and deploying network applications.

Compared to these tools, Trema provides a unique advantage by supporting both Ruby and C, making it accessible to a broader range of users with different programming skills. Additionally, Trema's modular design allows for the easy integration of new functionalities, which is crucial for educational purposes where flexibility and extensibility are important

2.1.2 Importance of SDN in Education and Research

Universities and research institutions are pivotal in advancing SDN technologies. They provide a fertile ground for innovation, enabling the exploration of new architectures, protocols, and applications. Research labs and academic institutions often employ SDN construction exercises to teach networking concepts and conduct experimental research.

2.1.3 Network Construction Tools and Techniques for Learners

Network construction tools and techniques are essential for learners to understand the intricacies of computer networks. These tools provide practical, hands-on experience that complements theoretical learning. Here’s an overview of some popular tools and techniques:

1. **Netkit**:Netkit is an open-source tool that uses User-Mode Linux (UML) to create virtual network environments on personal computers. It allows students to simulate complex network topologies, making it an ideal platform for hands-on learning without the need for physical hardware (Ricciato et al., 2008; Mongiello et al., 2013) [32]. Netkit’s simplicity and extensive documentation makeit accessible for beginners and useful in educational settings (Netkit, n.d.).
2. **Mininet**: Mininet is another widely used tool for network emulation, particularly in the study of Software-Defined Networking (SDN). It allows users to create a virtual network on a single machine, where they can run real code, applications, and services. Mininet supports rapid prototyping and is often used in academic research and teaching (Lantz et al., 2010; Handigol et al., 2012) [11].
3. **GNS3**: Graphical Network Simulator-3 (GNS3) [29] is a popular tool that provides a graphical interface for emulating complex networks. It supports a wide range of network devices and configurations, making it suitable for both beginners and advanced users. GNS3 is often used for certification training (e.g., Cisco CCNA) and in educational labs (Antonakakis et al., 2014; Martin et al., 2018) [2].
4. **Packet Tracer**: Cisco Packet Tracer is a network simulation tool developed by Cisco. It is widely used in networking courses and provides a user-friendly interface for building, configuring, and troubleshooting networks. Packet Tracer is especially useful for beginners learning Cisco networking concepts and preparing for Cisco certification exams (Cisco, n.d.) [6].

Here are some techniques for learners in building SDN network construction exercise problems;

1. **Hands-on Labs:** Hands-on labs are essential for applying theoretical knowledge to practical scenarios. Virtual labs using tools like Netkit, Mininet, and GNS3 allow learners to experiment with network configurations, troubleshoot issues, and understand the behavior of different network protocols (Forte et al., 2009; Cardenas et al., 2010) [9].
2. **Step-by-Step Tutorials**: Step-by-step tutorials guide learners through the process of setting up and managing network topologies. These tutorials often

include detailed instructions, screenshots, and explanations of key concepts, making complex tasks more approachable (Perkins & Pfleeger, 2005).

3. **Simulation and Emulation**: Simulation tools like Netkit and Mininet enable learners to emulate real-world network environments. These tools help in understanding network behavior, performance analysis, and testing new configurations without the risk of disrupting actual networks (Handigol et al., 2012; Lantz et al., 2010) [9].
4. **Collaborative Learning**: Collaborative learning involves students working together on network projects, sharing knowledge, and solving problems as a team. Tools that support collaborative features, such as shared virtual labs, enhance the learning experience by fostering peer-to-peer interaction and collective problem-solving (Smith et al., 2009).
5. **Automated Feedback and Hint Systems**: Intelligent systems that provide real-time feedback and hints can significantly aid learning. These systems analyze student interactions with network simulations, identify common errors, and offer constructive guidance, thereby improving understanding and reducing frustration (Jones et al., 2018; Wang et al., 2021) [22]. In [58],” Yuichiro Tateiwa et al, addressed the challenges faced by novices in mastering the complexities of network construction, particularly in the context of Software-Defined Networking (SDN). The paper explored the development of an intelligent system designed to provide real-time, context-specific hints to learners during network construction exercises. Their system utilized an analysis of the students’ actions and the states of the network to identify common mistakes and generate helpful hints, thereby facilitating a more guided and supportive learning environment. The approach aligned with contemporary educational theories that emphasize the importance of immediate feedback and adaptive learning environments in enhancing student engagement and understanding (Davis et al., 2020; Smith & Brown, 2019) [51]. The system’s integration with tools like Trema and Mininet offered a practical and scalable solution for educational institutions, allowing for the widespread adoption of SDN technologies in curricula. By addressing the steep learning curve associated with network construction, Tateiwa’s system not on improves learning outcomes but also encourages a more exploratory and confident

approach to mastering SDN concepts (Jones et al., 2018; Wang et al., 2021) [51]. Their approach highlights the potential for intelligent tutoring systems to transform technical education by providing personalized and adaptive learning experiences.

2.1.4 The Role of Trema in Network Construction Exercises

Trema plays a significant role in SDN education by enabling students to implement controller programs and build software-defined networks on their own PCs. This hands-on approach helps students gain a deeper understanding of SDN concepts and principles. In typical network construction exercises, students use Trema to develop controllers that manage virtual switches, such as Open vSwitch, and configure data plane networks to meet specific requirements.

These exercises often involve setting up network topologies, writing shell scripts for network configuration, and ensuring the networks meet certain reachability and communication criteria. However, students frequently encounter difficulties in debugging their controller programs due to the complex interactions between the control and data planes (Open Networking Foundation) [14].

2.1.5 Communication Activities in SDN

Communication activities in SDN involve interactions between the control plane and the data plane, as well as between different network elements. These interactions are crucial for the proper functioning of the network, ensuring that data is routed efficiently and securely. Understanding these dependencies is essential for optimizing network performance and reliability.

The paper "Determining Learning Status in SDN Construction Exercises" by Takashi Yokoyama [47] addressed the challenge of assessing students' learning progress in Software-Defined Networking (SDN) construction exercises. This research is important for educators in the field of networking, as it seeks to provide a method for evaluating whether students have understood and can apply SDN concepts effectively. Traditional methods of assessing students' understanding in practical networking exercises often rely on manual grading and subjective evaluation, which can be time-consuming and inconsistent. Yokoyama's research aimed to introduce an automated, objective method for determining students' learning status in SDN construction exercises. That paper presented an automated tool designed to assess students' progress and understanding in SDN exercises. That tool leverages data collected from students' interactions with the SDN environment to evaluate their performance. It made a significant contribution to the field of network education by introducing an automated tool for assessing students' progress in SDN exercises. The research addressed the limitations of traditional assessment methods and provides a scalable, objective, and efficient solution for evaluating student performance.

“Automatic Test Packet Generation" by Hongyi Zeng et al [13]. focused on the challenges and methodologies for generating test packets automatically to diagnose and troubleshoot network faults. Their research proposed a system that aims to ensure network reliability and performance by automating the generation and deployment of test packets, which is crucial for detecting and diagnosing network issues. Traditional methods of network troubleshooting often rely on manual packet crafting and monitoring, which can be time-consuming and error-prone. As networks grow and complexity, the need for automated tools to handle network diagnostics becomes increasingly important. The concept of test packet generation (TPG) has been explored in various forms, but Zeng et al.'s approach seeks to advance the field by introducing automation into the process.

2.2 Debugging and Dependency Analysis in SDN

One of the significant challenges in SDN is debugging controller programs and understanding the dependencies between different network components. Dependency analysis is crucial in identifying the cause-and-effect relationships between packet transmissions in the data plane and the control plane's responses. Existing approaches for dependency analysis in SDN often involve sophisticated tools and techniques that may be difficult for beginners to grasp.

The complexity of debugging SDN applications can be a major hurdle for students, who may struggle to identify and correct errors in their controller programs. A robust dependency analyzing system can help by providing insights into the interactions between different network elements, thereby simplifying the debugging process (Wikipedia) (SpringerLink).

Dependency analysis involves examining the relationships and interactions between different components of a system. In the context of SDN, it can help identify bottlenecks, potential points of failure, and opportunities for optimization. Dependency analysis tools can provide insights into the complex web of interactions within an SDN environment, aiding in the design and troubleshooting of networks.

Dependency analysis is a critical aspect of many fields, including software engineering, network management, and system design. It involves understanding and managing dependencies among components to ensure system reliability, maintain-ability, and performance. In complex systems like Software-Defined Networking (SDN), effective dependency analysis is crucial for troubleshooting, optimizing performance, and ensuring robust operations. The paper "A Practice-Driven Systematic Review of Dependency Analysis Solutions" [50] provides an extensive review of various dependency analysis solutions across multiple domains, with a particular emphasis on how these solutions are applied in practice. The review aims to understand the state of the art, identify common challenges, and highlight areas for future research. They suggested several areas for future research:

1. **Scalability Improvements**: Developing more scalable dependency analysis techniques to handle large and complex systems.
2. **Dynamic Dependency Analysis**: Enhancing methods for capturing and analyzing dynamic dependencies that occur at runtime.

3. **Integration with DevOps**: Integrating dependency analysis tools with DevOps practices to support continuous integration and delivery.
4. **Advanced Analytics:** Leveraging advanced analytics and machine learning to improve the accuracy and efficiency of dependency analysis. Overall, the review highlighted the importance of dependency analysis in ensuring the reliability and performance of complex systems and provides a roadmap for future research in this critical area.

2.2.1 Existing Systems and Tools for Dependency Analysis

Several tools and frameworks have been developed for dependency analysis in networking. These include:

1. **Network Topology Discovery Tools**: Tools like Nmap and Zenmap provide basic network topology discovery and visualization capabilities, which are essential for understanding the structure of an SDN network.
2. **Flow Analyzers:** Tools such as Wireshark and OpenFlow Visualizer help in analyzing the flow of packets through the network, providing insights into the communication patterns and dependencies [15].
3. **Simulation and Emulation Platforms**: Platforms like Mininet allow researchers to create virtual SDN networks, enabling the simulation of different network configurations and the analysis of their dependencies.

2.2.2 Challenges and Gaps in Dependency Analysis of SDN

While existing tools provide valuable insights, there are several challenges and gaps in the current state of dependency analysis in SDN:

1. **Scalability**: Many tools struggle to handle large-scale networks, which can limit their applicability in real-world scenarios.
2. **Real-time Analysis**: Real-time dependency analysis is critical for dynamic SDN environments but remains a challenging task due to the complexity and volume of data involved.
3. **Integration**: There is a need for better integration between different tools and platforms to provide a more comprehensive analysis of SDN networks.

Dependency analysis in Software-Defined Networking (SDN) is crucial for understanding the interdependencies among various network components, services, and applications. Despite its importance, there are several challenges and gaps that need to be addressed to achieve effective dependency analysis in SDN environments. Here are some of the key challenges and gaps are explained.

2.2.2.1 Complexity of Network Topologies

1. **Dynamic Nature of SDN**: SDN environments are highly dynamic, with frequent changes in network topology, policies, and configurations. This dynamism makes it challenging to accurately model and analyze dependencies.
2. **Large-Scale Networks**: The scale of modern networks, especially in large enterprises or data centers, adds to the complexity. Managing and analyzing dependencies across thousands of devices and flows is a daunting task.

2.2.2.2 Data Collection and Integration

1. **Diverse Data Sources**: Dependency analysis requires data from multiple sources such as network devices, controllers, and applications. Integrating this diverse data into a coherent framework is challenging.
2. **Real-Time Data**: To be effective, dependency analysis needs real-time or near- real-time data. Collecting and processing such data at scale without introducing significant latency is difficult.

2.2.2.3 Heterogeneity of Network Components

1. **Variety of Devices**: SDN networks often consist of heterogeneous devices from different vendors, each with its own management interfaces and data formats. This heterogeneity complicates dependency analysis.
2. **Compatibility Issues**: Ensuring compatibility and interoperability among various network components and the SDN controller is essential but challenging.

2.2.2.4 Policy and Configuration Changes

1. **Frequent Updates**: Network policies and configurations in SDN environments can change frequently. Tracking these changes and understanding their impact on dependencies is complex.
2. **Conflict Resolution**: Policy conflicts can arise in multi-tenant or multi-domain SDN environments, making it difficult to analyze and resolve dependencies accurately.

2.2.2.5 Security and Privacy Concerns

1. **Sensitive Data**: Dependency analysis may involve handling sensitive data, such as network configurations, flow information, and security policies. Ensuring the privacy and security of this data is critical.
2. **Attack Surface**: The centralized nature of SDN can introduce new security vulnerabilities. Understanding the dependencies among network components is essential to identify potential attack vectors.

2.2.2.6 Scalability Issues

1. **Resource Constraints**: Performing dependency analysis at scale can be resource-intensive, requiring significant computational and storage resources.
2. **Performance Overheads**: The analysis process should not introduce significant performance overheads that could impact network operations.

2.2.2.7 Lack of Standardization

1. **Proprietary Solutions**: Many SDN solutions are proprietary, leading to a lack of standardization in how dependency information is represented and analyzed.
2. **Interoperability Challenges**: The lack of standardization can also lead to interoperability challenges when integrating different SDN components and tools.

2.2.2.8 Tooling and Automation

1. **Limited Tools**: There is a limited number of tools available for comprehensive dependency analysis in SDN environments. Existing tools may not cover all aspects of dependency analysis.
2. **Automation Challenges**: Automating the dependency analysis process is challenging, especially in dynamic and large-scale SDN environments.

2.2.2.9 Visualization and Interpretation

1. **Complex Dependency Graphs**: Visualizing complex dependency graphs in a way that is easy to understand and interpret is a significant challenge.
2. **Actionable Insights**: Translating the results of dependency analysis into actionable insights that network operators can use to optimize and secure the network is not straightforward.

2.2.2.10 Resilience and Fault Tolerance

1. **Failure Impact Analysis**: Understanding how failures in one part of the network affect other parts is crucial for building resilient SDN networks. This requires comprehensive dependency analysis.
2. **Recovery Mechanisms**: Designing effective recovery mechanisms that consider the dependencies among network components is challenging.

2.2.3 Potential Solutions and Future Directions

1. Standardization Efforts: Developing standardized protocols and formats for representing and exchanging dependency information in SDN environments.
2. Advanced Analytics: Leveraging advanced analytics, machine learning, and artificial intelligence to improve the accuracy and efficiency of dependency analysis.
3. Real-Time Monitoring: Implementing real-time monitoring solutions that can provide up-to-date dependency information without significant performance overheads.

4. Interoperable Tools: Developing interoperable tools that can work across different SDN platforms and integrate with existing network management solutions.
5. Enhanced Visualization: Creating intuitive visualization tools that can present complex dependency graphs in an easily interpretable manner.

Addressing these challenges and gaps is essential for realizing the full potential of SDN and ensuring efficient, secure, and resilient network operations.

2.2.4 A Dependency Analyzing System Using Trema

The proposed dependency analyzing system using Trema aims to address the challenges faced by students in debugging their controller programs. By providing detailed insights into the causal connections between packet transmissions and controller responses, the system can help students identify and correct errors more efficiently.

This system would include features such as real-time monitoring of network traffic, logging of control and data plane interactions, and visualization tools to map out dependencies and interactions. By integrating these functionalities, the system can enhance the learning experience and improve the effectiveness of network construction exercises [14].

2.3 OpenFlow Switch Specification and Openflow Controllers

The Open Networking Foundation's (ONF) specification for OpenFlow and switch architectures is a cornerstone document in the realm of Software-Defined Networking (SDN). OpenFlow, as delineated by the ONF, provides a standardized protocol that allows the separation of the control plane from the data plane in network devices. This separation enables more flexible and programmable network management, which is crucial for the dynamic requirements of modern networking environments. The specification outlines how OpenFlow controllers communicate with network devices (switches and routers) to dictate the flow of data packets based on pre-defined policies. This protocol allows for granular control over network traffic, facilitating advanced features such as load balancing, security policy enforcement, and efficient traffic engineering [17].

The switch specification part of the ONF's document details the architectural requirements and functionalities that compliant switches must support to integrate seamlessly within an OpenFlow-enabled network. These specifications include support for multiple flow tables, group tables, and a wide range of match fields and actions, which enhance the switch's capability to handle complex traffic patterns. The ONF's comprehensive guidelines ensure interoperability among devices from different vendors, fostering a more competitive and innovative ecosystem. This standardization is critical for the widespread adoption of SDN, as it reduces the complexity and cost associated with deploying and managing network infrastructure. The ONF's work on OpenFlow and switch specifications has significantly contributed to the advancement and practical implementation of SDN technologies, enabling more agile, scalable, and efficient networks.

OpenFlow controllers play a crucial role in Software-Defined Networking (SDN) by serving as the brain of the network. They centralize the control logic and manage the flow tables of network devices (switches and routers) using the OpenFlow protocol. This centralization allows for a more flexible, programmable, and efficient network management approach. The controllers communicate with network devices to implement policies that control traffic flow, security measures, and quality of service parameters. By decoupling the control plane from the data plane, OpenFlow controllers enable dynamic and real-time network configuration adjustments, leading to more responsive and adaptive networks.

There are several types of OpenFlow controllers, each offering different features and capabilities. For example, the NOX controller, one of the first OpenFlow controllers, provides a platform for rapid development and deployment of new network applications. ONOS (Open Network Operating System) is designed for scalability and high availability, making it suitable for large-scale carrier and cloud networks. The OpenDaylight Project [31], a collaborative open-source project, offers a flexible and modular controller framework that supports various network protocols and southbound interfaces beyond OpenFlow. These controllers are integral in enabling diverse SDN applications, from data center management to network virtualization and beyond [3].

The architecture of OpenFlow controllers typically includes several key components: the northbound API, the southbound API, and the core controller functions. The northbound API allows for communication between the controller and higher-level applications, enabling network automation and orchestration. The southbound API is used to communicate with the underlying network devices, implementing the flow rules and policies defined by the controller. Core controller functions include topology discovery, path computation, and policy enforcement, which are essential for maintaining network performance and reliability. This architecture ensures that controllers can effectively manage complex and dynamic network environments.

Security and scalability are major considerations in the design and deployment of OpenFlow controllers. Ensuring secure communication between the controller and network devices is paramount, often achieved through the use of TLS encryption and authentication mechanisms. Scalability is addressed through various techniques, such as clustering multiple controller instances to distribute the load and provide redundancy. Advanced controllers like ONOS and OpenDaylight incorporate these features to support large-scale deployments, ensuring that the network can grow and adapt without compromising performance or security. Overall, OpenFlow controllers are essential components that drive the capabilities and benefits of SDN, enabling more efficient, secure, and flexible network management

2.4 Chapter Summary

The proposed Dependency Analyzing System represents a significant advancement in SDN and openflow controllers. By addressing the limitations of current tools, it aims to enhance the understanding and optimization of SDN networks. Future research should focus on refining these capabilities and exploring new applications in educational and industrial contexts.

CHAPTER 3

THEORETICAL BACKGROUND

This chapter is dedicated to provide the necessary background for research that constructs network exercises and analyzes communication activities in Software Defined Networking (SDN) for learners in networking. It explains with a theoretical background of SDN, describing each layer of the SDN architecture. As OpenFlow is integral to SDN, the chapter also outlines the structure and functions of OpenFlow switches like Open vSwitch, along with the OpenFlow protocol. Lastly, it offers a concise overview of SDN and OpenFlow controllers, with a particular emphasis on the Trema Controller framework.

3.1 The Architecture of Software Defined Networking

Software Defined Networking (SDN) architecture is a modern approach to designing and managing networks that aims to make networks more flexible, agile, and easier to manage through software control. Here are the key components of SDN architecture:

1. **Infrastructure Layer**: This consists of physical network devices, including switches, routers, and access points.
2. **Control Plane**: Unlike traditional networks where the control plane is integrated into each device, in SDN, it is centralized and separated from the physical devices, managed by software controllers.
3. **Data Plane**: Also known as the forwarding plane, this layer is responsible for the actual forwarding of data packets according to instructions from the control plane.
4. **SDN Controller**: The central component that communicates with the control plane of network devices using protocols like OpenFlow. It provides a unified network view, managing traffic flows, optimizing performance, and enforcing policies.
5. **Southbound Interface**: This interface connects the SDN controller to the network devices, allowing the controller to program their traffic-handling behavior.

6. **Northbound Interface**: Connects the SDN controller to applications or orchestration systems, enabling automation, provisioning, and policy-based management.
7. **Application Layer**: Applications on top of the SDN architecture use the centralized control provided by the SDN controller to deliver network services, enforce security policies, perform traffic engineering, and more.

SDN architecture offers advantages such as enhanced network programmability, simplified management and automation, greater agility in response to changing network conditions, and improved support for new services and applications. It is suitable for use in data centers, wide-area networks, and enterprise and campus networks.

The Open Networking Foundation (ONF), funded by companies such as Deutsche Telekom, Google, Microsoft, Facebook, Verizon, and Yahoo, aims to develop and standardize the OpenFlow (OF) protocol to advance networking. SDN has attracted significant attention from enterprises, service providers, and industry associations. As an emerging architecture, SDN meets the demands of high bandwidth and dynamic modern applications, offering an adaptive, cost-effective, dynamic, and manageable solution.

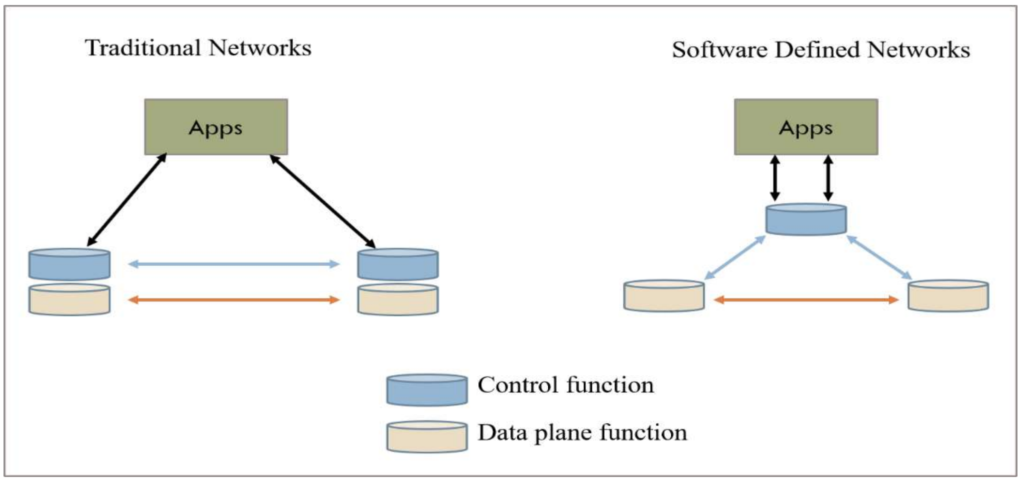


Figure 3. 1 Traditional Network Architecture VS SDN Architecture

In SDN architecture, the control and data functions are decoupled from network devices such as switches and routers. This architecture features global centralized control and promotes innovation through network programmability. Conversely, in most large enterprise networks, control and data functions are integrated within network devices, creating challenges for network operators when adjusting infrastructure and configuring numerous end devices, virtual machines, and virtual networks. Figure 3.1 contrasts traditional internet and SDN architecture, illustrating how the data plane layer (network devices) is simplified into straightforward forwarding elements while the control layer (controller) is logically managed. The data plane layer includes network devices (programmable switches) that can be implemented in hardware or software, and these switches support the OpenFlow protocol for communication and configuration with the controller. The benefits of decoupling control and data plane functions in SDN architecture include:

1. **Centralized Provisioning**: In SDN, centralized provisioning involves managing and configuring the network through a centralized control plane, unlike traditional networking's distributed control plane. Benefits include:
    - **Simplified Management**: A centralized controller simplifies network management, allowing administrators to handle the entire network from one point.
    - **Dynamic and Automated Configuration**: Real-time network configuration adjustments by the centralized controller lead to better performance and resource utilization.
    - **Improved Network Visibility**: The controller's global network view is essential for effective monitoring, troubleshooting, and optimization.
    - **Enhanced Security**: Centralized control ensures consistent security policy enforcement across the network, improving security.
2. **Reduced Operating Costs**: SDN reduces operating costs by improving efficiency, automation, and network management: Automation decreases manual intervention, lowering labor costs. Efficient resource use reduces hardware and energy expenses.
3. **Scalability**: Centralized provisioning in SDN enhances scalability, although large networks may need multiple SDN controllers due to the practical limits of managing devices.
4. **Security**: SDN controllers support centralized security management, addressing challenges posed by virtualization and allowing administrators to manage network security more effectively.

5. **Direct Programmability**: Network managers can program network operations directly, using abstract control over forwarding elements to adjust traffic flows dynamically. This enables custom application use for managing, configuring, securing, and optimizing network resources independently of proprietary software.
6. **Openness**: Every data plane element (such as OpenFlow-enabled switches or routers) has a unified programming interface for the OpenFlow controller to collect network status, regardless of vendor.

The SDN architecture mainly consists of the following three layers: the application layer, control layer, and data plane layer as shown in Figure 3.2.

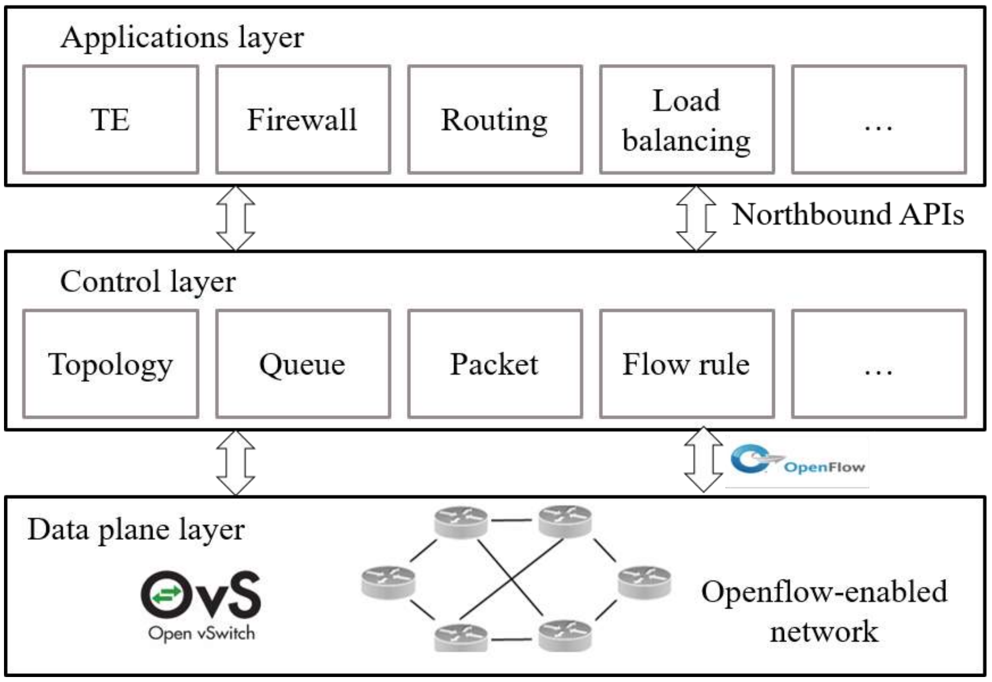


Figure 3. 2 Software Defined Networks Architecture.

The SDN applications are programmed to support all kinds of network services such as traffic engineering, load balancing, firewall, routing, and monitoring. The control layer is a core layer of the SDN architecture that extracts the data plane layer information and communicates to the application layer with an abstract view of the network topology, including statistics and events. The application and control layers communicate by using northbound APIs. The data plane layer consists of network nodes which can forward and processing of the data path. Communications between the data plane and control layers use a standardized protocol called OpenFlow. The SDN Controller defines the data flows that take place in the SDN Data Plane. When the flow is entered to the network, the flow must first take permission from the controller [38]. The controller decides whether the communication is permissible or not according to the network policy. If the flow is permitted, the controller decides an appropriate route for the permitted flow and adds flow entry for the permitted flow in each switch along the path. The SDN controller is responsible for these complex tasks and switches simply manage flow tables and focus on forwarding function.

3.2 Infrastructure Layer (or) Data Plane Layer

The data plane layer would be the physical layer over which network virtualization lays down through the controller. This layer consists of various networking equipment which may be OpenFlow-enabled or OpenFlow-complaint network devices (routers or switches).

Table 3. 1 Example of OpenFlow-Complaint Switches

Vendor	Series
Arista	Arista extensible modular Operating System (EOS), Arista 7124FX application switch.
Cisco	Cisco cat6k, catalyst 3750, 6500 series
Cinea	Cinea Core director running firmware version 6.1.1
HP	HP procurve series-5400 xzl, 8200 zl, 6200yl, 3500yl
Juniper	Juniper MX-240, T-640
NEC	NEC IP8800
Toroki	Toroki Lightswitch 4810
Dell	Dell z9000 and S4810
Quanta	Quanta LB4G4
Open vSwitch	Software switch, Latest version 1.10.0

The OpenFlow enabled switches are either based on the OpenFlow protocol or compatible with it. In the data plane layer, traffic may enter or exit through logical or physical ports by forwarding or processing functions. Management of forwarding functions performed by an SDN controller or other mechanisms that orchestrated in conjunction with the SDN controller. An OpenFlow enabled switch may be a hardware device or software program which are capable of processing and forwarding of the data path. The examples of OpenFlow-complaint switches are shown in Table 3.1.

3.2.1 Open vSwitch

Open vSwitch (OVS) [42] is a multilayer software switch aimed at providing a software switch platform with standard management interfaces and the capability for programmatic extension and control of forwarding functions. Ideal for virtual switch operations in virtual machine (VM) environments, OVS offers standard control and visibility interfaces to the virtual networking layer and supports distribution across multiple physical servers. It is compatible with various Linux-based virtualization technologies such as VirtualBox and Xen/XenServer. Written in platform-independent C, OVS can be easily ported to other environments. The switch can function entirely in user space without the need for a kernel module, making user-space implementation easier to port than a kernel-based switch. OVS in user space can also interface with Linux or DPDK devices. OVS contains the following distributions:

1. **ovsdb-server (database server):** ovsdb-server provides remote procedure call (RPC) interfaces to one or more OVS databases and supports JSON-RPC client connections over Unix domain sockets and TCP/IP. It is a lightweight configuration database server that holds information for bridges, interfaces, tunnel definitions, OVSDB managers, and an OpenFlow controller address. It also allows ovs-vswitchd to query its configuration.
2. **ovs-vswitchd (daemon):** It is the core part of the OVS and it manages any number of OVS switches on the local machine. The daemon communicates with SDN controllers, ovsdb-server, kernel module, and hosting system by using OpenFlow, OVSDB protocol, netlink, and netdev interface respectively.
3. **ovs-dpctl:** A tool for configuring the switch kernel module.
4. **ovs-vsctl:** A utility for updating and querying the configuration of ovs-vswitchd.
5. **ovs-appctl:** A utility that sends commands to running OVS daemons.

3.2.2 OpenFlow Switches Specifications

OpenFlow-compliant switches come in two main types: OpenFlow-only and OpenFlow-hybrid. OpenFlow-only switches are exclusively managed by the OpenFlow pipeline and do not support other protocols. In contrast, OpenFlow-hybrid switches offer both OpenFlow and traditional network functionalities [38]. An OpenFlow switch typically includes one or more flow tables, one or more OpenFlow channels to external controllers, a group table, and a meter table, and shown in Figure 3.3.

- ***Ports***: Packets traverse through the network interface known as OpenFlow ports, facilitating communication between OpenFlow processing and the broader network. OpenFlow switches interconnect using these ports. Typically, OpenFlow ports are categorized into three types: physical, logical, and reserved ports.

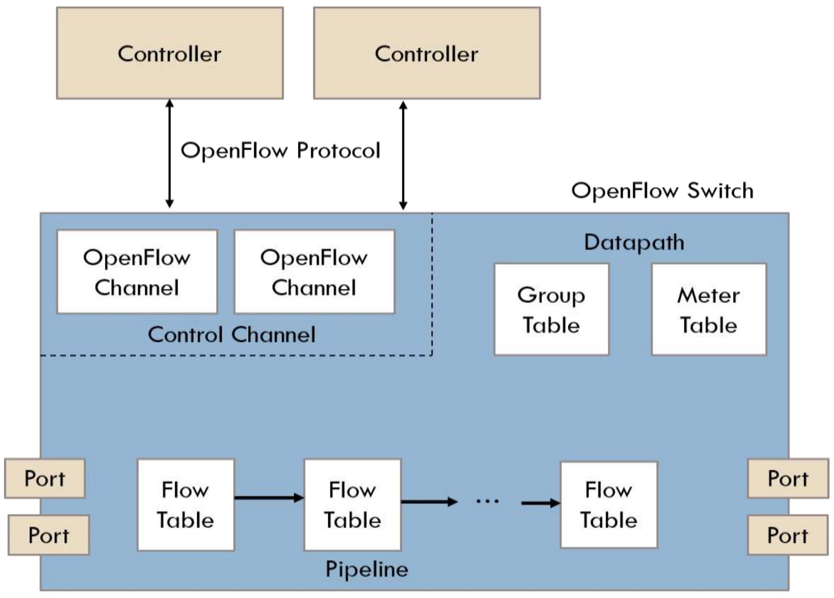


Figure 3. 3 Main Components of an OpenFlow Switch

- ***Flow Table***: A flow table is integral to OpenFlow switches, managing packet forwarding through network paths. Each flow table consists of flow entries encompassing A flow table consists of flow entries and each flow entry consists of:
    - ***Match fields***: consists of ingress port, packet header, and metadata to match against packets.
    - ***Priority*****:** matching precedence of the flow entry.
    - ***Counters***: to update for matching packets.
    - ***Instructions***: modify the pipeline processing or action set.
    - ***Timeouts***: maximum amount of time before the flow is expired.
    - ***Cookie***: Used to provide flow modification, deletion, and statistics by the controller.

- ***Group Table***: OpenFlow networks employ group tables for managing multicast, broadcast, and load balancing functionalities. Each group entry contains a group identifier, type, counters, and action buckets.
- ***Meter Table***: The meter table houses meter entries that define per-flow meters critical for Quality of Service (QoS) operations such as rate-limiting and DiffServ. Meters oversee and regulate packet rates assigned to them, directly impacting flow entries. Each meter entry is identified by a meter identifier, meter bands specifying actions, and counters.
- ***OpenFlow Channel*:** Serving as the communication link between OpenFlow switches and controllers, the OpenFlow channel facilitates switch configuration, event handling, and packet transmission. OpenFlow channel messages adhere to the OpenFlow protocol and can be secured with TLS encryption or transmitted directly via TCP.

The controller can add, delete, and update the flow tables entries in an OpenFlow switch via OpenFlow protocol.

3.2.3 Pipeline Processing of OpenFlow Switches

In OpenFlow switches, packets are handled by the OpenFlow pipeline. Packets are received on an ingress port and processed by the pipeline, which may forward them to an output port. Pipeline processing consists of two stages: ingress and egress, as depicted in Figure 3.4. For flow tables numbered from 0 to n, pipeline processing always starts at the ingress processing of flow table 0. The numbers assigned to ingress flow tables must be lower than those assigned to egress flow tables.

Initially, a packet is matched against the first ingress table, and other tables may be used depending on the result of this first match. If the ingress processing outcome is to forward the packet to an output port, the OpenFlow switch will begin egress processing for that output port. Egress processing is optional; hence, a switch might not provide or configure any egress tables. If no valid table is configured at the first egress table, the packet may be handled by the output port or forwarded out of the switch. If there is a valid configured table at the first egress table, the packet must match against the flow entries of that table, and other tables may be used depending on the result of this match.

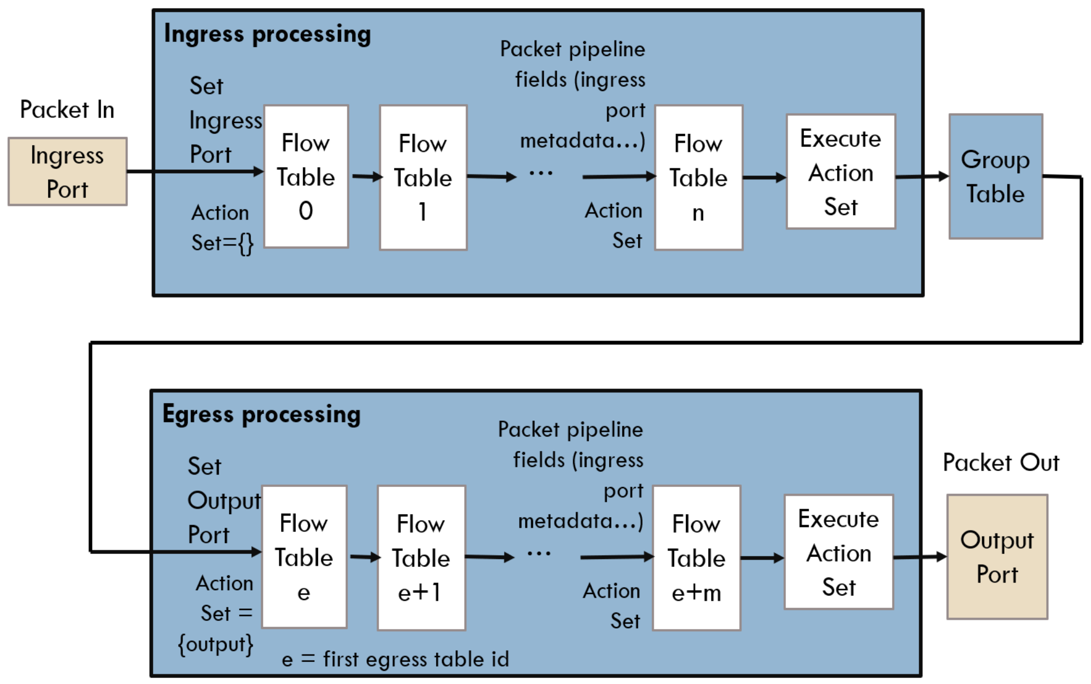


Figure 3. 4 Openflow Pipeline Architecture

3.2.4 Matching Flow Table in OpenFlow Switches

OpenFlow switches employ flow tables where each entry is uniquely identified by its match fields and priority. These match fields, encompassing ingress ports, packet headers, and metadata, define the criteria for selecting a specific flow entry within the table. Each flow entry includes counters to monitor packet usage and a set of instructions specifying actions to be executed.

Figure 3.4 visualizes the flow matching structure in OpenFlow, illustrating how packets interact with flow tables during processing. Upon reaching a flow table, packets are scrutinized against its flow entries to identify a suitable match. Upon successful matching, the associated instructions of that flow entry are triggered. These instructions, such as Apply-actions, Clear-actions, Write-actions, Write-metadata, and GoTo-table, determine subsequent actions. If the instruction is GoTo-table, the packet may proceed to another flow table for further processing. In the absence of a GoTo-table instruction, processing within the current flow table concludes, and the packet is handled based on the specified actions.

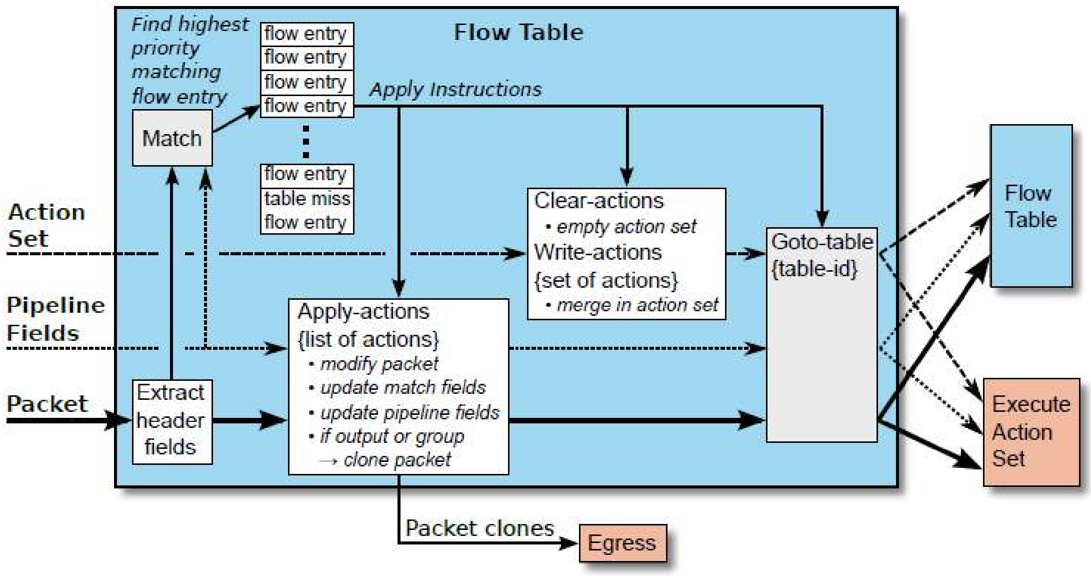


Figure 3.5 Matching and Instruction Extraction in a Flow Table

As in Figure 3.5, Every flow entry includes a series of instructions that are carried out when a packet matches the entry. These instructions lead to modifications in the packet, action set, and/or pipeline processing [38].

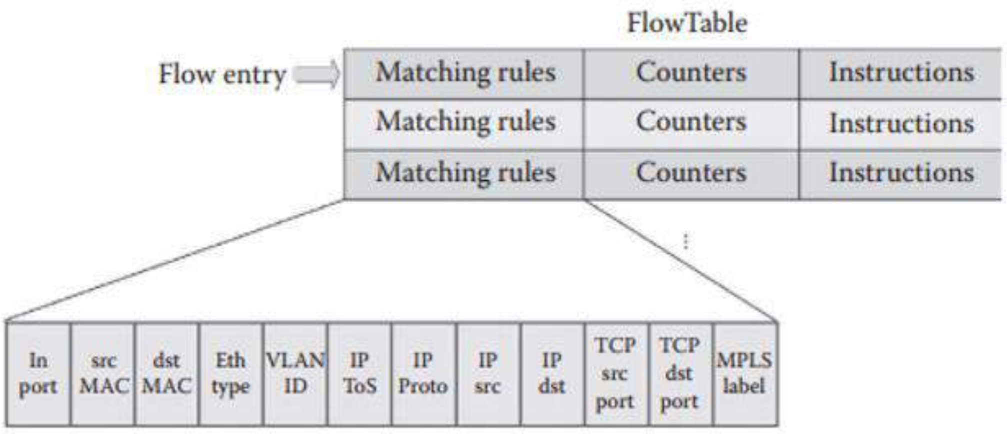


Figure 3. 6 Flow Matching Structure of OpenFlow

If a flow entry is not matched, known as a table miss, the handling of packets depends on the configuration of the flow table. The instruction set for table miss scenarios dictates how unmatched packets are processed. These instructions may include dropping packets, redirecting to another flow table, or sending packet-in messages back to the OpenFlow Controller through the control channel.

3.3 Protocol Options for Southbound Interface

The control layer communicates the data plane layer by using Southbound APIs (Application Programming Interfaces). The controller uses these APIs to dynamically change forwarding rules that installed in the data plane devices such as switches and routers [50]. There are some examples of southbound APIs that are used for managing network devices in SDN deployment: **NETCONF** (standardized by IETF), **Opflex** (supported by Cisco), **OF-Config** (supported by the Open Network Foundation (ONF)), **OpenFlow** and so on. To support hybrid networks or to utilize traditional networks with software-defined manner, some routing protocols (i.e. OSPF, ISIS, and BGP) have been developed as southbound interfaces in some OpenFlow controller. Currently, the most popular southbound API is OpenFlow.

3.3.1 The Concept of OpenFlow Protocol

OpenFlow is a standardized protocol facilitating communication between OpenFlow switches and controllers. It serves as a programmable network protocol with an open standard-based interface, enabling various vendors to manage and support network traffic. Through OpenFlow, SDN controllers can configure data plane devices like OpenFlow switches by installing packet forwarding rules. Switches, in turn, communicate events and notifications to controllers via the OpenFlow protocol.

At initialization, switches configure their SDN controller's IP address and TCP port number. They establish a secure TLS session for communication. The controller sends an OFPT FEATURES REQUEST message to each switch to gather configuration details such as port numbers and MAC addresses, essential for network management. OpenFlow messages fall into three main types:

1. ***Controller to switch messages***: These messages, initiated by the controller, control or monitor switch states:
    - **Features:** Establishes the OpenFlow channel by requesting switch capabilities, with the switch responding via a feature replies message.
    - **Configuration**: Allows the controller to set and query switch configurations.
    - **Modify-State (FLOW_MOD):** Used to add, modify, or delete flow or group entries.

- **Read-State**: Retrieves various switch information like current configurations and port statistics.
- **Packet-Out**: Sends packets from the controller to switches.
- **Barrier:** Ensures message dependencies are met and receives notifications for completed operations.
- **Role-Request**: Sets the role of the OpenFlow channel.
- **Asynchronous-Configuration**: Defines additional filters for asynchronous messages on the OpenFlow channel.

2. ***Asynchronous***: These types of messages are applied to change the switch state and update the controller with the network events changes. These messages are initiated by switches. These messages are:
    - ***Packet-in***: Transfer the control of a packet to the controller. It may be table-miss flow entry, TTL checking or packet-in events.
    - ***Flow-Removed***: Inform the controller about the flow has been removed because of the controller's flow delete request or the switch's flow expiry process.
    - ***Port-Status***: Inform the controller about the status of the port.
    - ***Error***: The switch enables to notify the problems to controllers using error messages.
3. ***Symmetric:*** These types of messages are initiated by either the controller or the switch and sent without solicitation. Five symmetric messages have been represented as a part of the OpenFlow protocol:
    - ***Hello***: Hello messages or keep-alive messages exchanged between switch and controller upon connection startup.
    - ***Echo*:** To verify the liveness of connection, the controller and switch used echo request/reply messages.
    - ***Experimente*r**: To supports additional functionality within OpenFlow message type space or an area for the features of future OpenFlow versions.

3.4 Control Layer of SDN

In the context of Software-Defined Networking (SDN), the control layer assumes a pivotal role in the centralized management and orchestration of network resources and policies. It effectively separates the control plane from the data plane, optimizing network operations. By abstracting data plane details, the control layer provides a comprehensive view of the network topology, including key statistics and events. Communication between the application layer and the control layer is facilitated through northbound APIs. Notable open-source controllers utilized widely today include OpenDayLight [31], ONOS, NOX, FloodLight, Ryu, Trema, and other similar platforms. In this paper, we won't introduce one by one but we describe Trema Openflow controller which used in the test experiments. Table 3.1 shows some of the feature comparisons of popular open-source SDN controllers.

Table 3. 2 Features Comparison of Popular SDN Controllers

Controller	Centralized/ Distributed	Implementation	Developers	Application Domain
NOX	Centralized	Python	Nicira Networks	Campus
POX	Centralized	Python	Nicira Networks	Campus
Ryu	Centralized	Python	NTT	Campus
Trema	Centralized	C and Ruby	NEC	Research, Education, Prototyping of Custom Network Services
FloodLight	Centralized	Java	Big Switch Networks	Campus
OpenDayLight	Distributed	Java	The Linux Foundation	Datacenter
ONOS	Distributed	Java	Open Networks Foundation	Datacenter, WAN and transport

3.4.1 Trema OpenFlow Controller

Trema is an open-source framework designed for developing OpenFlow controllers. It is known for its ease of use, modularity, and performance, making it suitable for research, education, and development of network applications. It was developed with the languages C and Ruby where Trema's core is written in C, providing high performance for low-level operations and efficient packet processing and Trema offers a high-level API in Ruby, which simplifies the development process. Ruby scripts can be used to write network applications, leveraging the flexibility and ease of scripting. Trema was developed by NEC Corporation, a company known for its contributions to networking and telecommunications technologies.

Trema is open-source and has a community of contributors who help maintain and enhance the framework. This community support ensures that Trema remains up-to-date with the latest developments in SDN and OpenFlow technologies. It is widely used in academic and research environments for SDN experimentation and education. Its simplicity and flexibility make it an excellent tool for teaching and learning about SDN and OpenFlow. Developers can use Trema to rapidly prototype and test new network applications. Its high-level Ruby API allows for quick development cycles and easy modification of network logic. Trema can be used to simulate and test OpenFlow networks, making it valuable for developers and researchers who need to validate their SDN applications and configurations.

The application use cases are:

4. **Network Function Virtualization (NFV):** Prototyping and testing NFV applications.
5. **Custom Network Services**: Developing custom network services such as load balancers, firewalls, and monitoring tools.
6. **Educational Tool**: Teaching SDN and OpenFlow programming in academic settings.

The key features of Trema are;

1. **Ease of Use**: The Ruby API makes it easy for developers to create and manage OpenFlow applications.
2. **Modularity**: Trema's modular architecture allows for easy extension and customization.
3. **High Performance**: The C core ensures efficient packet processing and low-level operations.
4. **Comprehensive Documentation**: Trema offers extensive documentation and examples to help developers get started quickly.

Trema can be installed from its GitHub repository, where detailed instructions are provided. The framework includes various examples and templates to help new users begin developing their own OpenFlow applications.

Trema is a powerful and flexible OpenFlow controller framework suitable for a variety of use cases, from academic research to network application development. Its combination of high performance and ease of use makes it an attractive option for those looking to explore and implement SDN solutions.

These resources provide detailed information on Trema's capabilities, installation, and usage, making it easy for developers and researchers to leverage this framework in their SDN projects. Figure (3.7) shows the architecture of Trema Controller Framework as in [12].

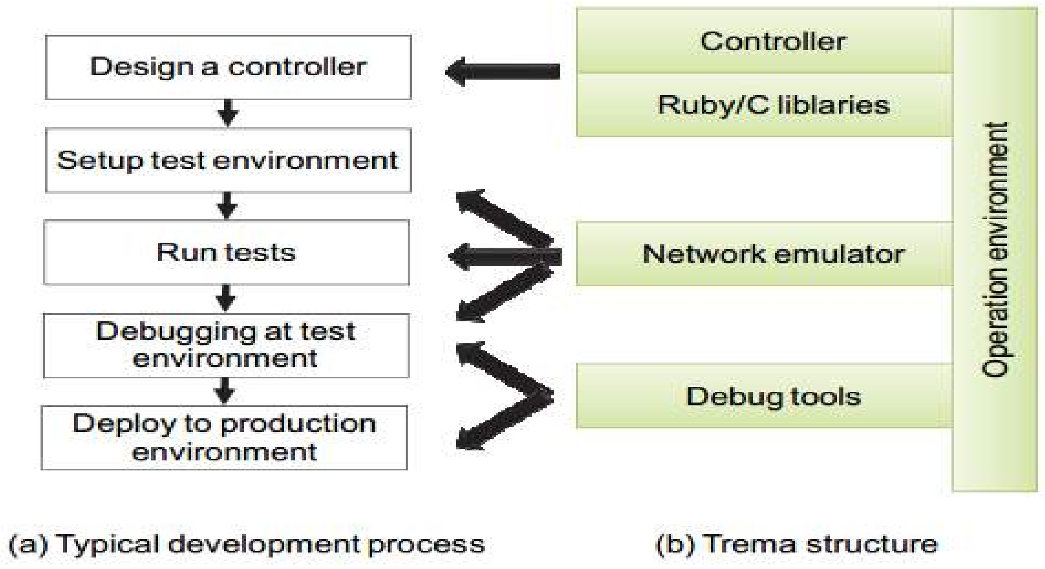


Figure 3. 7 The Architecture of Trema Controller Framework

3.5 Application Layer of SDN

The application layer within Software-Defined Networking (SDN) serves as a crucial domain for developing innovative applications that capitalize on global network information, including comprehensive data on network topology, statistics, and operational status. This layer interacts extensively with the control layer to deploy network services and functionalities effectively. It comprises a diverse array of network applications that leverage the SDN controller's abstractions to streamline network optimization and management. Key components of the SDN application layer encompass northbound APIs, network application development, automation and orchestration features, analytics, and policy management capabilities. The benefits of application layer are:

1. **Customization and Flexibility**: Network operators have the flexibility to develop customized applications that meet specific operational needs and objectives.
2. **Accelerated Innovation**: The programmable environment of the SDN application layer promotes rapid innovation, enabling the quick deployment of novel services and functionalities.
3. **Improved Operational Efficiency**: Automation and orchestration tools within the SDN application layer simplify management tasks, leading to reduced complexity and enhanced network efficiency (Learnenough.com).

To summarize, the application layer within SDN is pivotal, leveraging SDN's flexibility and programmability to deliver advanced network services, improve security, optimize performance, and automate network operations. Open standards and APIs play a crucial role in enabling diverse application development and integration, driving innovation and operational efficiency across modern network environments.

3.6 Management of Flow Entries in OpenFlow Networks

In the context of SDN architecture, the controller is mandated to install flow table entries in the forwarding tables of switches. The match fields, often utilizing wildcard entries, are traditionally housed in ternary content-addressable memory (TCAM) for swift packet matching and forwarding. Nonetheless, TCAMs are both expensive and spatially restrictive, limiting the number of entries that can be included in the flow table. The flow management system in OpenFlow switches is broadly classified into two main methodologies: proactive and reactive.

The proactive approach to flow management involves the controller pre-calculating and populating flow entries in the switch's flow tables. This strategy minimizes setup time and latency because flows do not need constant consultation with the controller. However, it lacks flexibility for real-time adjustments to network traffic and may encounter challenges with fitting a large number of entries into the flow table's TCAM. Reactive installation of flow entries is employed to address the management of large flow tables more adaptively. The basic operations for reactive flow management are depicted in Figure 3.8:

1. Packets arrive at the switch and there are no corresponding flow entries in switch’s flow table.
2. Therefore, the switch informs the controller about the packet.
3. The controller determines the path for the packet and puts in suitable rules in each switch along the path.
4. Packets are forwarded to the destination.

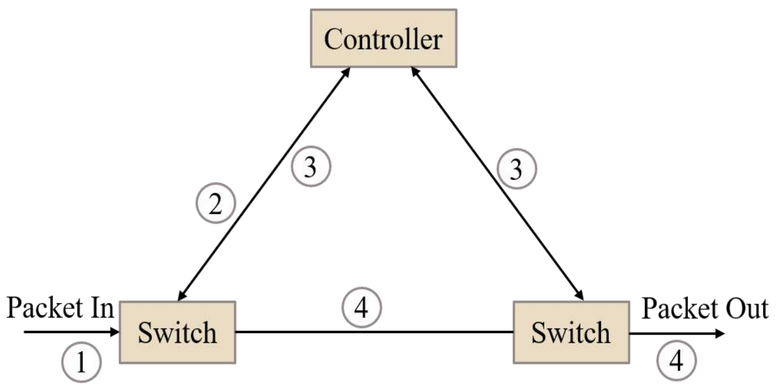


Figure 3. 8 Reactive Flow Management

The reactive approach to flow management operates on a timeout-based mechanism with a default expiry timer, typically set to one second by the controller. When flows expire, the switch removes them and requests new flow entries from the controller to handle subsequent packets.

Both reactive and proactive mechanisms have their own set of advantages and drawbacks. In the reactive method, controller interaction is necessary when a new flow arrives or when the switch's flow table lacks an appropriate entry. This approach efficiently utilizes flow tables but introduces additional setup time for each flow, which depends on the controller's workload and the state of the control channel. Consequently, reactive flow management may reduce the number of large flow tables in switches but can increase latency and reliability requirements for the control channel and control plane software. Failures in the control plane software or control channel can significantly impact network performance if flow entries cannot be established promptly.

In a proactive approach, all required flow entries are pre-installed in the switches' flow tables. This minimizes the workload on the controller and enhances resilience against failures in the control layer, since the necessary flow entries are already programmed into the data plane switches. However, deploying this method in larger networks involves handling numerous flow tables, which may face challenges due to TCAM limitations.

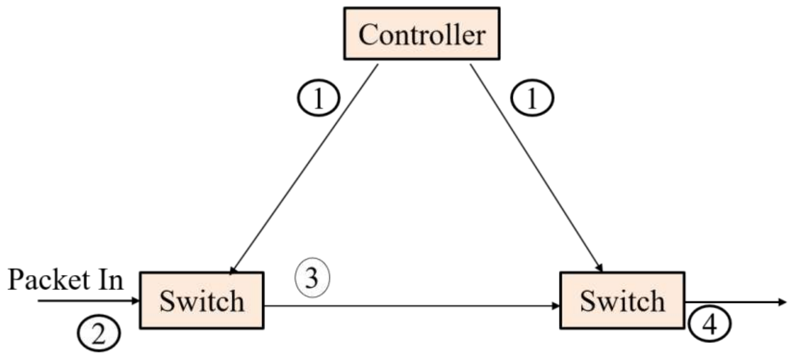


Figure 3. 9 Proactive Flow Management

To overcome the limitations of both proactive and reactive approaches, the hybrid flow management mechanism has gained popularity. This approach combines the advantages of proactive flow rule installation before communication begins with the flexibility of reactive adaptation to traffic during communication.

3.7 Innovation Through Routing based SDN Application

In the realm of SDN architecture, network managers have the opportunity to innovate their applications to align with specific needs. This has led to increased research focus on applications such as traffic engineering, routing, load balancing, and security. Whether in SDN environments or traditional networks, routing fundamentally revolves around two essential components: network state information and routing algorithms. Network state information encompasses node and link resources, including parameters like link utilization, available bandwidth, delay, and packet loss rate.

Routing algorithms leverage network state information to determine optimal routes based on resource availability and demand. Yet, this information is subject to dynamic changes due to fluctuating link statuses, varying loads, and connection statuses. In traditional networks, distributed routing protocols handle the acquisition and dissemination of network state information among routing devices. In contrast, SDN

simplifies this process by allowing controllers to gather and update network state information directly from routing devices using OpenFlow connections.
Routing algorithms involve routers calculating the shortest path between each pair of nodes across a network. The Open Shortest Path First (OSPF) Protocol utilizes the Shortest Path First (SPF) algorithm as its basis. Within networking, the primary focus remains on traffic management and routing, specifically on determining paths that adhere to essential constraints such as network QoS parameters. This method is known as constraint-based routing. Figure 3.10 depicts the various routing algorithms that are widely used in SDN, SDN based IoT networks, SDN based cloud data centers networks, and conventional networks. According to Figure 3.10, there are two main types of routing: shortest path routing and constrained based routing.

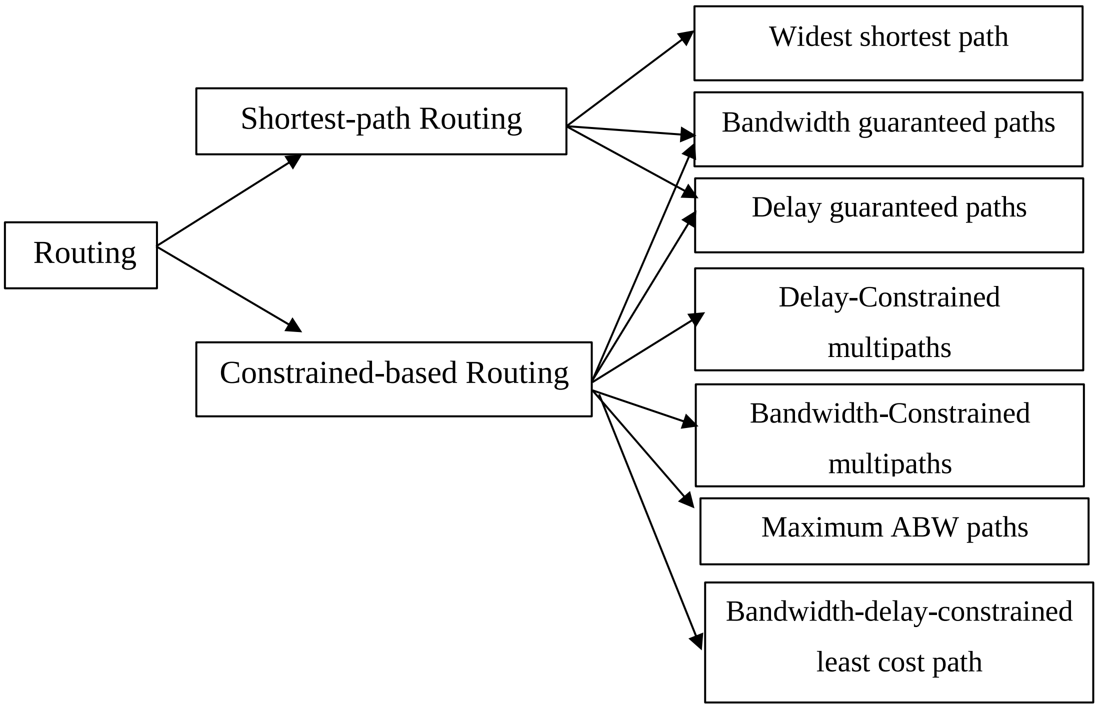

Routing
Shortest-path Routing
Constrained-based Routing
Widest shortest path
Bandwidth guaranteed paths
Delay guaranteed paths
Delay-Constrained multipaths
Bandwidth-Constrained multipaths
Maximum ABW paths
Bandwidth-delay-constrained least cost path

Figure 3. 10 Routing Algorithms in SDN and Traditional Networks.

3.8 Chapter Summary

This chapter provides a concise overview of the foundational theory behind the layer taxonomy of software-defined networks (SDNs) [20]. It details the principal SDN protocol, explaining its operational mechanisms. Additionally, the chapter outlines the architecture and functionalities of Open vSwitch, a widely-used OpenFlow switch and he details about the controller Trema used in this system is presented in this chapter.

This chapter also explores various routing methods commonly employed in both SDN and traditional networks.

CHAPTER 4

THE ARCHITECTURAL DESIGN OF PROPOSED SYSTEM

The purposes of this chapter are identifying the problems of student's network construction exercises and proposing to analyze the dependency and visualize them with sequence diagrams. The design of the system architecture is explained with step by step explanation.

4.1 Problems Definitions of SDN Network Construction

These problem definitions aim to cover a broad range of SDN concepts, providing learners with hands-on experience in setting up and managing SDN networks, implementing flow rules, monitoring and analyzing traffic, ensuring QoS, implementing security policies, and dynamically configuring networks.

4.2 Network Construction Exercise Structure

In these exercises, novice learners build their networks with the following steps;

1. Learners receive network construction exercise problems from the instructors
2. Build data plane networks by executing the shell scripts on their own PCs.
3. Code OpenFlow controllers to satisfy the requirements and then establish connections between the controllers and switches in the data plane networks by executing Trema.
4. Evaluate the behavior of the networks with the ping command and the proposed system.
5. Debug the controllers.

Table 4.1 Devices and Data Structure used in the System

Device	Data structure
Switch	(Device name *name*)
Host	(Device name *name*, IP address *ip*)
Joint	(Device name *name*, Ethernet port number *ep*)
Cable	{Joint1,Joint2}

The ethernet port number ep starts from 1 in switches, and is assigned 1 in hosts.

4.3 Network Configuration

Teachers define network configuration in files such as make-dc.sh. The devices and links that are used in the constructed networks are implemented by means of the following technologies;

Host: a network namespace [18]

Switch: an Open vSwitch process [37]

Controller: a Tema process [50]

Link in data-plane: a veth pair [51]

Link in control-plane: a local loopback interface [18]

The data structures of these elements are shown in table 4.2.

Table 4.2 Elements and Data Structure used in the System

Element	**Data Structure**
Switch	(Device name *name*, Datapath-ID *dpath*, network interface names *NI*)
Host	(Device name *name*, network interface name *ni*, IP address *ip*)
Joint	(Device name *name*, Ethernet port number *ep*)
Cable	{Joint1,Joint2}
Controller	listen port number *lstn*

The NI in the switch is an array that consists of the network interface name implementing an Ethernet port in the switch. The ni in the host is the name implementing the Ethernet port in the host. If we define a joint j1 with the switch sw and its network interface sw.NI[i], we can describe it as j1 = (sw.name, i + 1).

4.4 Preliminary of the System Design

We defined several functions in table 4.3.

Table 4.3 The Functions Created for the System

Function	**Description**
$SrcTCPPort(pkt)$	It returns the source TCP port number of the packet *pkt*.
$OFMsg(pkt)$	It returns the OpenFlow message (header and payload) of the packet *pkt*.
$IsOFPFC\_ADD(ofmsg)$	If the OpenFlow message *ofmsg* means ofp flow mod command::OFPFC ADD, it returns true.
$Match(ofmsg)$	It returns the match field of *ofmsg*.
$Action(ofmsg)$	It returns the action field of *ofmsg*.
$Payload(ofmsg)$	It returns the payload of *ofmsg*.
$M\_IN\_Port(match)$	It returns the in port field of the match field *match*.
$M\_SrcMAC(match)$	It returns the source MAC address field of *match*.
$M\_DstMAC(match)$	It returns the destination MAC address field of *match*.
$A\_OUT\_Port(action, a)$	It returns the *a*-th out port field of the action field *action*.
$DPath(name)$	It returns the datapath-ID of the switch whose name is *name*.
$getNI(name, ep)$	It returns the network interface name that impliments the Eth- ernet port number *ep* of the switch whose name is *name*.
$Peer(ni)$	It returns the network interface name that is another edge of the network interface name *ni* in the cable.
$DevByNI(ni)$	It returns the device name that equips the network interface name *ni*.
$DevByPort(port)$	It returns the device name that establishes connections using TCP port number *port* as the source port in the control-plane.

In this section, we describe the detail explanation of the method and functions used in the proposed system with its data structure and the formulation of all the functions are explained.

4.4.1 Data structure of the System Design

Table 4.4 shows data structure used in the system.

4.4.2 Formulation of Handler Instance

The file log data.txt includes several lines including ’method start’. Eq.2 extracts all lines from the line including ’method start’ to the previous line including ’method start’. HI1 stores hander instances that are extracted from the last line including ’method start’ to the end line in the file. Eq.4 sets the last line including ’method start’ to i. Eq.5 sets h so that CP [h] is the closest to SI[i]. The SI[m] is the statement instance that called ’packet out()’ or ’flow mod()’. The CP [l] is the OpenFlow message that is generated by ’packet out()’ or ’flow mod()’.

$$HI = HI_0 \cup HI_1 \tag{4.1}$$

$$HI_0 = \{(i, i+j-1, null, null) \mid 0 < j \wedge SI[i].is\_ms = true \wedge SI[i+j].is\_ms = true\} \tag{4.2}$$

$$\text{HI1} = \{(\text{i}, \text{Size(SI)} - 1, \text{h}, \text{l}) \tag{4.3}$$

$$\mid \ \text{SI[i].is ms} = \text{true} \wedge \max(\text{i}) \tag{4.4}$$

$$\wedge \ \text{CP [h].time} < \text{SI[i].time} \wedge \min(\text{SI[i].time} - \text{C P [h].time}) \tag{4.5}$$

$$\wedge \ \text{SI[i].method} = \text{‘packet in}'$$

$$\wedge \ \text{SI[i].ARGS[0]} = \text{DP ath(DevByP ort(SrcT CP P ort(CP [h].pkt)))}$$

$$\wedge \ \text{SI[i].ARGS[1]} = \text{OF M sg(CP [h].pkt)}$$

$$\wedge \ \text{SI[m].time} <= \text{SI[Size(SI)} - \text{1].time} < \text{CP [l].time}$$

$$\wedge \ \min(\text{SI[m].time} - \text{CP [l].time})$$

$$\wedge \ \text{SI[m].ARGS[0]} = \text{DP ath(DevByP ort(SrcT CP P ort(CP [l].pkt)))}$$

$$\wedge \ (\text{SI[m].method} = \text{‘packet out}' \wedge \text{T ype(OF M sg(CP [l].pkt))} = 13$$

$$\vee \ \text{SI[m].method} = \text{‘flow mod}' \wedge \text{T ype(OF Msg(CP [l].pkt))} = 14)\} \tag{4.6}$$

4.4.3 Formulation of Flow Entries

The current our system supports only flow-entries addition with OpenFlow messages of ’FLOW MOD’. For this reason, the time e stores just ’null’.

$$\begin{aligned} FE = \{&(DevByPort(SrcTCPPort(CP[trigger\_add].pkt)), CP[trigger\_add].time, null, \\ &Match(OFMsg(CP[trigger\_add].pkt), Action(OFMsg(CP[trigger\_add].pkt)), \\ &trigger\_add) \mid IsOFPFC\_ADD(OFMsg(CP[trigger\_add].pkt)) = true\} \end{aligned} \tag{4.7}$$

4.4.4 Formulation of Forwarding Instance, Data Plane to Control Plane and Control Plane to Data Plane

Switches receive packets and send the packets. After receiving a packet, the switch tries to find a flow-entry whose match field matches the packet.

If it is found, the switch applies the action field to the packet and sends it from the out port. The FI store the details.

$$
\begin{aligned}
FI = \{(rcv, snd, fe, a) \mid\ & fe \in FE, fe.name = DevByNI(DP[rcv].ni) \\
& \land fe.time_s < DP[rcv].time \land min(DP[rcv].time - fe.time_s) \\
& \land getNI(fe.name, IN\ Port(fe.match)) = DP[rcv].ni \\
& \land M\ SrcMAC(fe.match) = getSMAC(DP[rcv].pkt) \\
& \land M\ DstMAC(fe.match) = getDMAC(DP[rcv].pkt) \\
& \land DP[rcv].time < DP[snd].time \land min(DP[snd].time - DP[rcv].time) \\
& \land Peer(getNI(fe.name, A\ OUT\ Port(fe.action, a))) = DP[snd].ni \\
& \land DP[rcv].pkt = DP[snd].pkt\}
\end{aligned}
\tag{4.8}
$$

If it is not found, the switch generates a query (OpenFlow message) and sends it to the controller. The D2C stores the received packet and the sent query.

$$
\begin{aligned}
D2C = \{(rcv, snd) \mid\ & \\
& \neg\exists fe \in FE, fe.name = DevByNI(DP[rcv].ni) \\
& \land fe.time_s < DP[rcv].time \\
& \land getNI(fe.name, IN\ Port(fe.match)) = DP[rcv].ni \\
& \land M\ SrcMAC(fe.match) = getSMAC(DP[rcv].pkt) \\
& \land M\ DstMAC(fe.match) = getDMAC(DP[rcv].pkt) \\
& \land DevByNI(DP[rcv].ni) = DevByPort(SrcTCPPort(CP[snd].pkt) \\
& \land Type(OFMsg(CP[snd].pkt)) = 10 \\
& \land DP[rcv].time < CP[snd].time \\
& \land min(CP[snd].time - DP[rcv].time)\}
\end{aligned}
\tag{4.9}
$$

After receiving the query, the controller may send the answer to the switch. After receiving the answer (OpenFlow message) from the controller, the switch acts based on the answer, which may send packets from the out port field in the answer. The C2D stores the received answer and the sent packets.

$$
\begin{aligned}
C2D = \{(snd, rcv, a) \mid\ & \\
& \neg\exists fe \in FE, fe.name = DevByNI(DP[snd].ni) \\
& \land fe.time_s < DP[snd].time \\
& \land getNI(fe.name, IN\ Port(fe.match)) = DP[rcv].ni \\
& \land M\ SrcMAC(fe.match) = getSMAC(DP[rcv].pkt) \\
& \land M\ DstMAC(fe.match) = getDMAC(DP[rcv].pkt) \\
& \land CP[rcv].time < DP[snd].time \\
& \land min(DP[snd].time - CP[rcv].time) \\
& \land Type(OFMsg(CP[rcv].pkt)) = 13 \\
& \land Peer(getNI(DevByNI(CP[rcv].ni), A\ OUT\ Port(Action(OFMsg(CP[rcv].pkt)), \\
& a)) \\
& = DP[snd].ni \\
& \land Payload(OFMsg(CP[rcv].pkt)) = DP[snd].ni\}
\end{aligned}
\tag{4.10}
$$

4.4.5 Formula of Packet Chain

A packet chain pc ∈ PC is formulated with Eq.11 and 12.

$$pc[0], pc[1] = \begin{array}{ll} re.rcv, re.snd & (re \in FI, DevByNI(Peer(DP[re.rcv].ni)) = \text{'}vhost*\text{'}) \\ & 2c \in D2C, c2d \in C2D, hi \in HI, \\ d2c.rcv, c2d.snd & DevByNI(Peer(DP[d2c.rcv].ni)) = \text{'}vhost*\text{'}, \\ & d2c.snd = hi.cp\_in \wedge ac2d.rcv = hi.cp\_out) \end{array} \tag{4.11}$$

$$pc[i] = \begin{array}{ll} re.snd & (2 \le i, re \in FI, pc[i-1] = re.rcv) \\ & (2 \le i, d2c \in D2C, c2d \in C2D, hi \in HI, \\ c2d.snd & pc[i-1] = d2c.rcv \wedge d2c.snd = hi.cp\_in \wedge c2d.rcv = hi.cp\_out) \end{array} \tag{4.12}$$

Table 4.4 Explanation of Data structure

Name	**Data structure**
DP	It is the array consisting of the tuple (*ni, time, pkt*), where the packet *pkt* was captured from the network interface name *ni* in the data-plane at time *time*.
CP	It is the array consisting of the tuple (*time, pkt*), where the packet *pkt* was captured from the network interface name *ni* in the data-plane at time.
OF PORT	It is the set consisting of the tuple (*name,port*), where the device whose name is *name* establishes OpenFlow connections using the TCP port *port* bin the control-plane.

SI	It stores executed statements in the controller. The array of the tuple (*time, start, end, isms, ismr, method, ARGS*), where the statement from the *start*-th line to the *end*-th line was executed at time *time*. If the statement is the first statement in a handler method (start(), switch ready(), packet in()), the *isms* is true; else if the statement called a function (packet out(), flow mod())for sending OpenFlow messages, the *ismr* is true. The *method* stores the name of the executed hander method or the called method, and *ARGS* stores its arguments. We call an element in *SI* **a statement instance**.
HI	It stores executed handers. It is the set of the tuple (*si s, si e, cp in, CP OUT*), where the *SI*[*si s*] is the first exe cuted statement in the hander, and the *SI*[*si e*] is the lastly executed statement in the hander. And also, the contoller executed - the handler to treat the *CP*[*cp in*] and *CP*[*cp out*], *cp out* ∈ *CP OUT* was sent by the controller because of executing packet out() or flow mod().
FE	It stores flow-entries with validity periods, meaning time-based flow-table. It is the set of the tuple (*name, time s, time e, match, action, trigger add*), where the *name* stores the switch name, the validity period starts from time *time s* to time *time e*. And also, the *match and action* store the match fields and the action fields respectively. After receiving *CP* [*trigger add*], the switch added the flow-entry to the flow-table.
FI	It stores forwarding logs in switches based on flow-tables, called **forwarding instances**. It is the set of the tuple (*rcv, snd, fe, a*), where a switch receives *DP*[*rcv*] and then sends *DP* [*snd*] based on the *a*-th *out port* in the action of a flow-entry *fe*.
D2C	It stores causal connections between the data-plane packets and the control-plane packets. If switches receive *DP*[*rcv*] and then send *CP*[*snd*], the tuple (*rcv, snd*) is added to D2C.

$C2D$	It stores causal connections between the data-plane packets and the control-plane packets. If switches receive $CP[rcv]$ and then send $DP[snd]$ from the a-th out port in the action field of $CP[rcv]$, the tuple (rcv, snd, a) is added to C2D.
PC	The set of the array that stores the captured packets whose pay loads are identical on the assumption that payloads are identifiable. The packets are sorted by captured order. PC stands for Packet Chain. When payloads in DP[i].pkt, DP[i+j].pkt, and DP[i+j+k].pkt are identical and different to others in DP, $pc \in PC$, pc[0], pc[1], and pc[2] store i, i+j, and i+j+k respectively.

In this work, Trema controller is used to implement controller programs and build software defined networks. The detailed requirements and implementations of software and hardware will be explained in section 5.1.

Trema can be used in the following area as case studies, such as

1. **University Networks**: Trema has been used in university campus networks to study and implement dynamic network management and security solutions.
2. **Enterprise Networks**: Enterprises use SDN controllers like Trema to develop custom network management solutions tailored to their specific needs.

In Trema, there are many applications such as;

1. Educational Tools and Simulations
2. Network Management and Monitoring
3. Security Applications
4. Load Balancing
5. Network Virtualization
6. QoS (Quality of Service) Management
7. Research and Prototyping.

By leveraging Trema, developers and researchers can create a wide range of SDN applications that improve network performance, security, and manageability.

4.5 A Dependency Analyzing System Architecture

In our system, we design a sample testbed network topology which includes 3 virtual hosts and 3 virtual switches as in Figure 4.1.

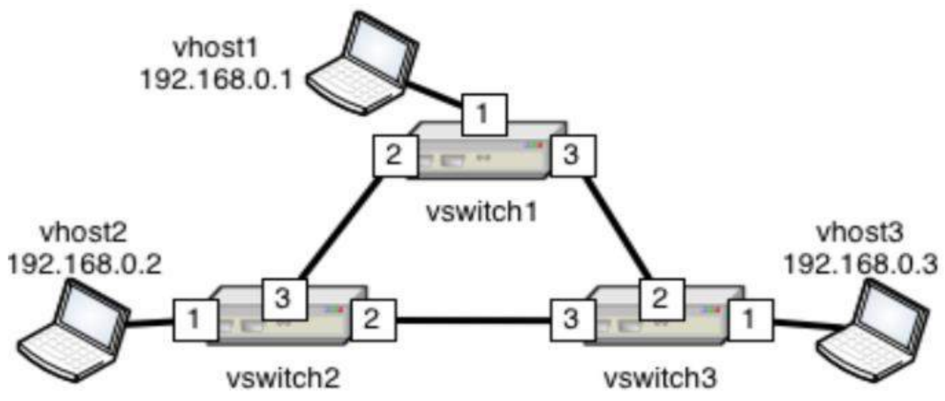


Figure 4. 1. A Sample Network Topology

4.5.1 Overview Explanation of the System

The overall system design of the proposed system is depicted in figure 4.2. It is a block diagram of the proposed system.

In the proposed system, the network configuration information of the exercise problem and the controller program are first received from the user. In such a case, the data collection function starts packet capture and Trema, and logs the communication test performed by the user.

When the user finishes the communication test and makes a termination request, the data collection function terminates packet capture and Trema, obtains packets, OpenFlow messages, and execution history, and saves them in the log DB as packet information, OpenFlow information, and executable statement information to do.

When the data collection function ends, the time series flow table reproduction function, route selection imitation function, and transmission route estimation function are executed in order, and the time series flow table, route selection history, and transmission route information are created and saved in the log Database.

There is a clue creation function as a means of obtaining necessary information from the log DB, and this function is obtained from the log DB and returns the above three clues to the user.

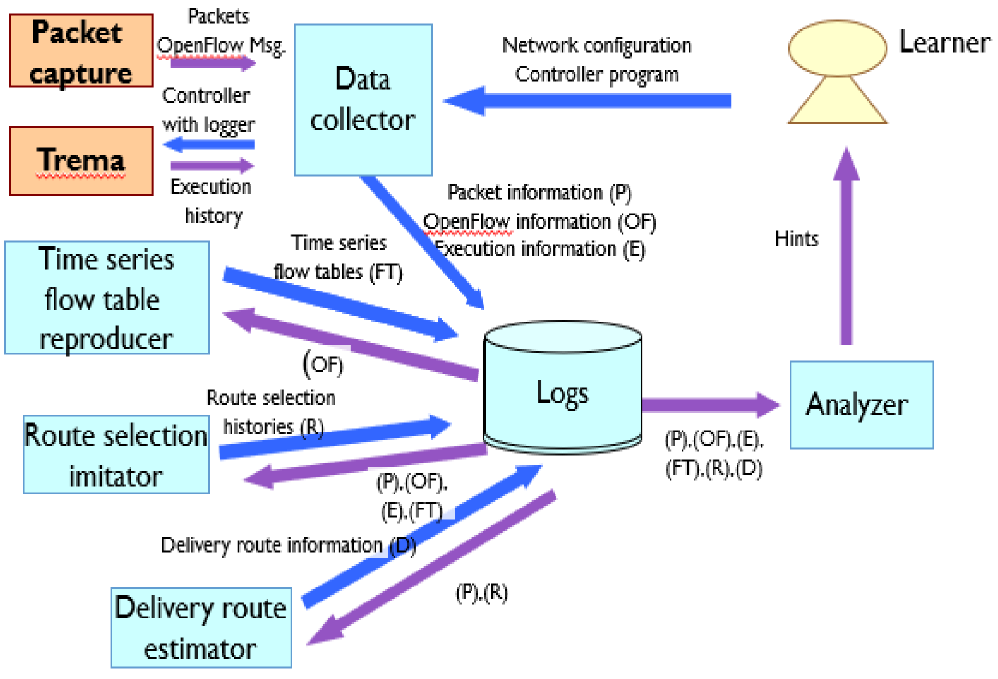


Figure 4. 2 The Overall System Design

4.5.2 Architectural Design of the Proposed System

The architecture of the proposed system is shown in Figure 4.3. The network interface (NI) extractor finds the names of the network interfaces that are used for links. Each tcpdump captures packets from each network interface. The IP address port resolver finds the names of network interfaces used for links and the names of devices equipping the network interfaces by analyzing the network configuration. This information enables the resolver to find the names of devices that send/receive the packets captured by each tcpdump in the data plane. The resolver also finds the port number and name of each Open vSwitch [37] that is used for connecting to Trema. It enables the resolver to find the names of Open vSwitches that send/receive packets in the control plane. Finally, the resolver writes out all captured packets together with the capturing time, the names of the sending devices, and the names of the receiving devices.

The logging code inserter adds codes that write which statements were executed onto the standard outputs. The event extractor finds the logs from the standard outputs. The generator sorts all the events by time order and then generates descriptions for visualizing them with a sequence diagram. PlantUML [42] converts the description of the output log events to the graphics.

4.6 Chapter Summary

The details about the proposed system architectur design is depicted in this chapter. Here we explained the details about the exercise problem, system design and the functions used in this system. The system implementation phase will describe in the next chapter.

CHAPTER 5

IMPLEMENTATIONS OF PROPOSED SYSTEM

In this chapter, the implementations of the proposed system with experimental testbed design is explained. The hardware requirements and software requirements used in this system are also described in detail in this chapter. Then, the experimental testbed topology and the controller programs are designed by using Trema controller with ruby script.

5.1 Design and Implementation of Experimental Testbed

Figure 5.1 explains the flow of SND Network construction exercises. In the exercise, the network configuration is given by the instructor. Given information to learners are network configurations, communication examples (data for sending and packet delivery routes), shell scripts for constructing networks, procedures for executing controllers. The achievement conditions for the learners are while building networks that satisfy the following conditions as an example;

1. The topology is given in Figure 5.2.
2. All hosts can communicate with ping each other.
3. No packet-loss
4. Minimum number of controller events (packet-in and packet-out)

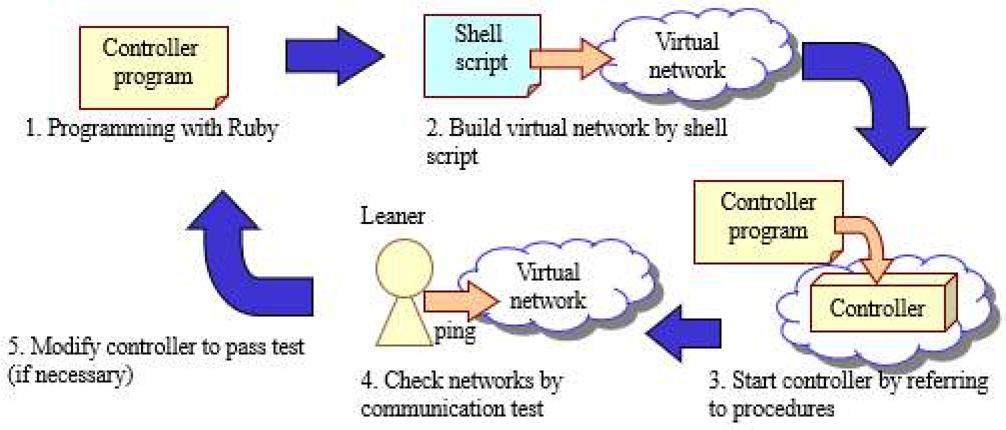

Controller program
1. Programming with Ruby
Shell script
Virtual network
2. Build virtual network by shell script
Controller program
Controller
3. Start controller by referring to procedures
Leaner
Virtual network
ping
4. Check networks by communication test
5. Modify controller to pass test (if necessary)

Figure 5.1 Logical Testbed Design of Network Exercise Flow

To satisfy those conditions the learner proceeds with the exercise using the Trema according to the following procedure. Firstly, learners create a controller program. Next, they set the virtual network environment using a shell script. And then, they execute the controller program according to the controller execution procedure and perform a communication test. If this communication test fails to meet the communication example, the controller program is corrected as necessary. The exercise will proceed by repeating this process.

Table 5.1 describes hardware requirements of experimental testbed and Table 5.2 describes the software, tools that are used in this research.

Table 5. 1 Hardware Requirements of Experimental Testbed Area

Name	Specifications
CPU	Core(TM) i7-6500U CPU @ 2.50GHz 2.59 GHz
RAM	8.00GB
HDD	500 GB
Operating System	Linux 16.04 LTS
Number of PCs	1

Table 5. 2 Software and Tools Used in the System

Software	Versions	Used in
Trema (SDN Controller)	0.4.7	Implementation and Evaluation
Open vSwitch	2.9.2	Testbed (Evaluation)
OpenFlow Protocol	Version 1.3	Testbed (Evaluation)
PlantUML	1.2024.5	Testbed (Evaluation)

5.2 Testbed Network Topology

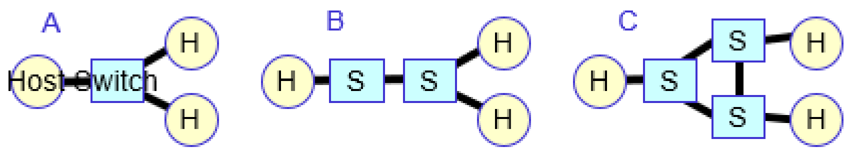


Figure 5. 2 Logical Testbed Nnetwork Topologies

In Figure 5.2, we set three communication network topologies as a testbed network environment. In the first figure A, we set three virtual host and one virtual switch. In the second figure B, we set the topology with three virtual hosts and two virtual switches. And finally, we set three virtual hosts and three virtual switches as in figure C.

5.3 Requirements of Software for Experimental Testbed

In this system the following software are required to develop and experiment;

1. Trema Controller
2. PlantUML

5.3.1 Installation and Running of Trema Controller

For the implementation of the proposed system, we used Trema controller framework with ruby scripts. To install Trema on an Ubuntu system, follow these steps:

Step 1: Update and Install Dependencies

Start by updating your package list and installing the required dependencies:

sh

sudo apt-get update

sudo apt-get install build-essential git ruby ruby-dev libpcap-dev

sh

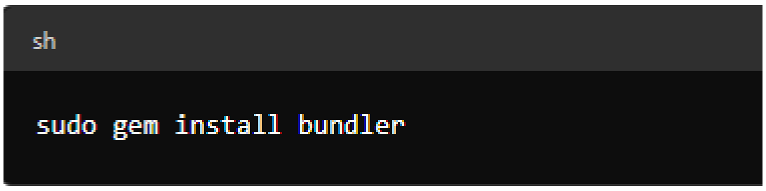
```
sudo apt-get update
sudo apt-get install build-essential git ruby ruby-dev libpcap-dev
```

Step 2: Install Bundler

Bundler is a dependency manager for Ruby projects:

sh

sudo gem install bundler



sh

```
sudo gem install bundler
```

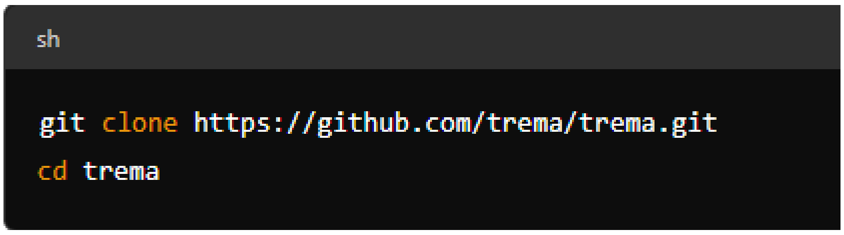
Step 3: Clone the Trema Repository

Clone the Trema GitHub repository to your local machine:

sh

git clone https://github.com/trema/trema.git

cd trema



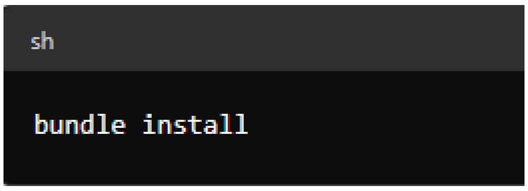
```sh
git clone https://github.com/trema/trema.git
cd trema
```

Step 4: Install Trema

Install Trema and its dependencies using Bundler:

sh

bundle install



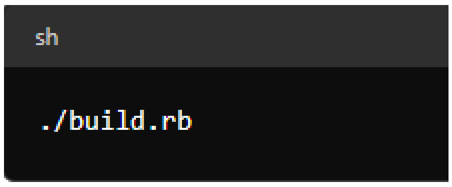
```sh
bundle install
```

Step 5: Build Trema

Build Trema:

sh

./build.rb



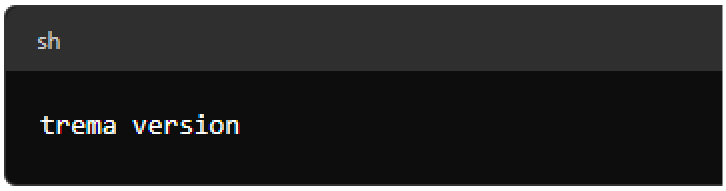
```sh
./build.rb
```

Step 6: Verify Installation

Verify that Trema is installed correctly by checking the version:

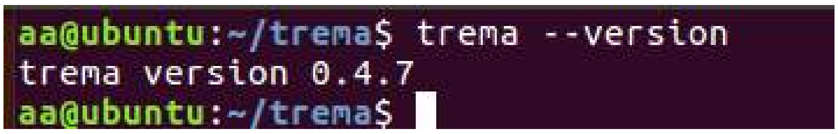
sh

trema version



```sh
trema version
```

If the installation was successful, you should see the Trema version displayed. Here we used the trema version 0.4.7



```
aa@ubuntu:~/trema$ trema --version
trema version 0.4.7
aa@ubuntu:~/trema$ 
```

Note: The installation steps mentioned above are for the Ubuntu OS installation because this dissertation used Ubuntu OS for implementation and hence explain the required step for Ubuntu OS. The detailed installation steps and installation steps for other OS can find in the documentation on the Trema GitHub page or website and it can provide further assistance.

5.3.2 Installation and Running of PlantUML

PlantUML[42] is a tool to create UML diagrams from plain text descriptions. It is widely used for documenting software systems and can generate various types of diagrams such as sequence diagrams, use case diagrams, class diagrams, and more. The installation process may varies depending on the users' OS type and version.

Step by Step Manual Installation Procedures for PlantUML

For more control over the installation process, you can manually download and set up PlantUML.

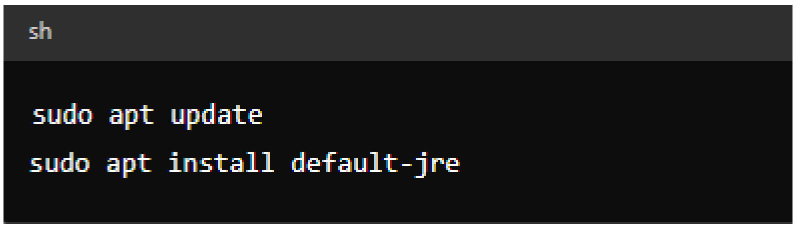
1. Install Java:

   As mentioned before, PlantUML requires Java.

   sh

   sudo apt update

   sudo apt install default-jre



```sh
sudo apt update
sudo apt install default-jre
```

2. Download PlantUML JAR File:

   Download the plantuml.jar file from the official website.

   sh

   wget http://sourceforge.net/projects/plantuml/files/plantuml.jar/download -O plantuml.jar

```sh
wget http://sourceforge.net/projects/plantuml/files/plantuml.jar/download -O plantuml.
```

3. Install Graphviz:

   sh

   sudo apt install graphviz

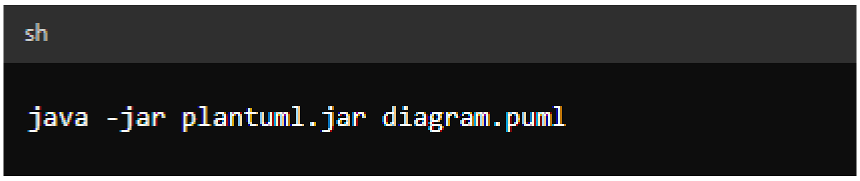
```sh
sudo apt install graphviz
```

4. Run PlantUML:

   Use the following command to generate diagrams from a PlantUML file (e.g., diagram.puml):

   sh

   java -jar plantuml.jar diagram.puml



```sh
java -jar plantuml.jar diagram.puml
```

5. Example Diagram Description

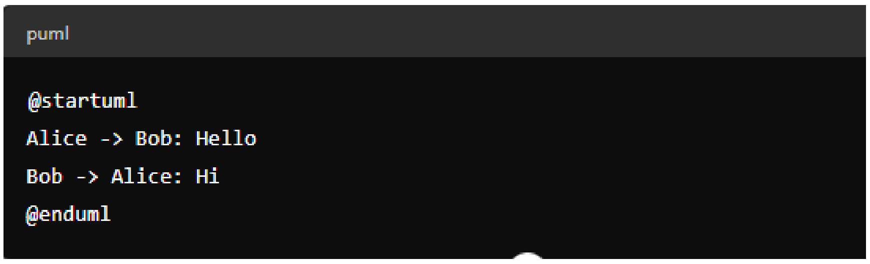
   Create a sample PlantUML file (example.puml): puml

   @startuml

   Alice -> Bob: Hello

   Bob -> Alice: Hi

   @enduml



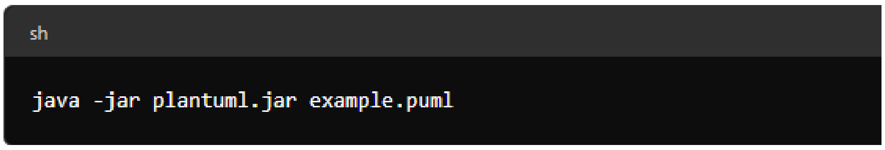
```
puml

@startuml
Alice -> Bob: Hello
Bob -> Alice: Hi
@enduml
```

6. Generate the diagram:

   sh

   java -jar plantuml.jar example.puml



```
sh

java -jar plantuml.jar example.puml
```

By following the above procedures, we can able to install and use PlantUML on our Ubuntu system effectively. For more detailed information, refer to the PlantUML documentation in [42].

5.4 Implementation of the System

In this system, we implement the dependency analyzing system by using functions and methods that are designed and mentioned detailed in Chapter 4.

There are two methods for transmitting communication data using OpenFlow. The first one is transferring data packets by FlowTable and the other is transferring data packets by PacketOut function.

5.4.1 Transferring Packets by FlowTable

The first method is packet transmission based on the flow table explained in figure 5.3. Firstly, FlowMod (Flow Modification Method) is sent by executing the controller. The switch updates the flow table according to this FlowMod. When a packet arrives, the switch uses this flow table to determine the packet output port and send it out.

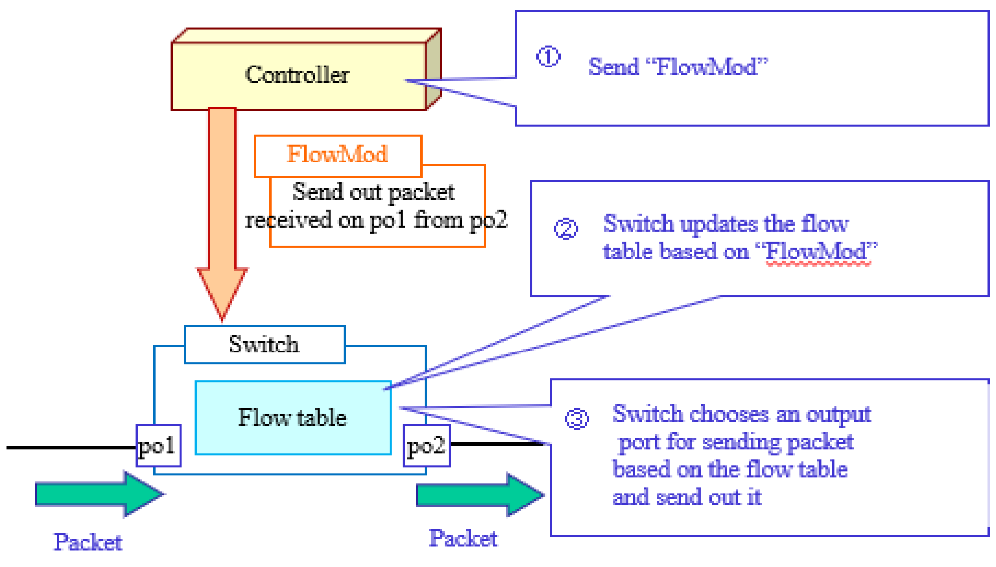

Controller
① Send "FlowMod"
FlowMod
Send out packet received on po1 from po2
② Switch updates the flow table based on "FlowMod"
Switch
Flow table
po1
po2
③ Switch chooses an output port for sending packet based on the flow table and send out it
Packet
Packet

Figure 5.3 Transfer Packets by Flow Table

5.4.2 Transferring Packets by PacketOut Method

The second method of packet transmission based on PacketOut method is explained in Figure 5.3. Firstly, when a packet arrives, the switch verifies that the packet does not match the flow table. The switch then sends a PacketIn containing the packet to the controller. PacketOut is sent when the controller that receives this PacketIn is executed. The switch determines the packet output port by this PacketOut and sends it out.

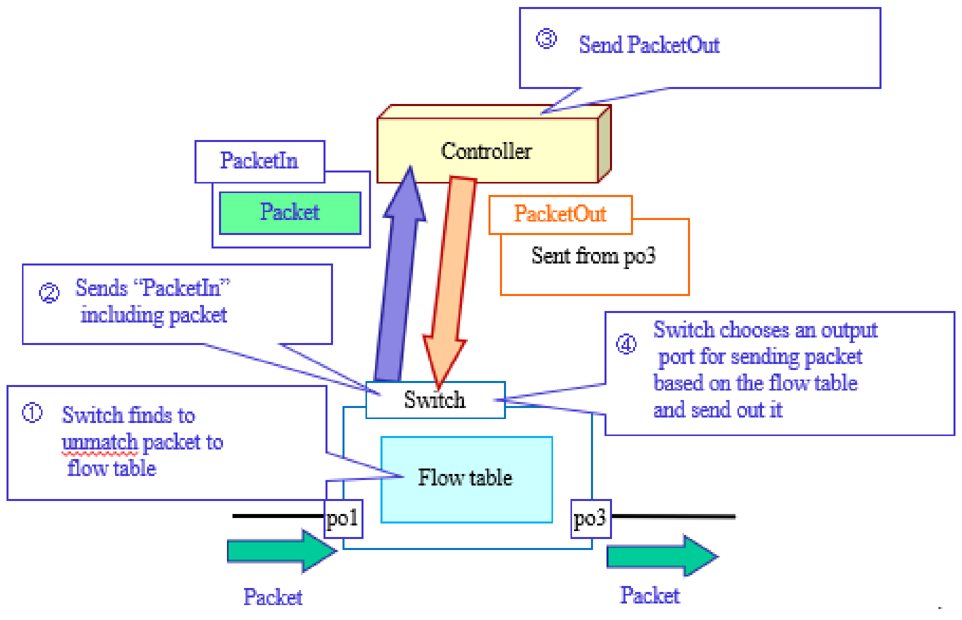

③ Send PacketOut
PacketIn
Packet
Controller
PacketOut
Sent from po3
② Sends "PacketIn" including packet
④ Switch chooses an output port for sending packet based on the flow table and send out it
① Switch finds to unmatch packet to flow table
Switch
Flow table
po1
po3
Packet
Packet

Figure 5.4 Transfer Packets by PacketOut

5.4.3 Data Collection Function

In this stage, the network configuration information of the exercise problem and the controller program are first received from the user. In such a case, the data collection function starts packet capture and Trema, and logs the communication test performed by the user.

The data collection function generates the following logs to the database for future analysis and to generate visualize event;

- Packet information (P)
- Open flow information (OF)
- Execution information (E),

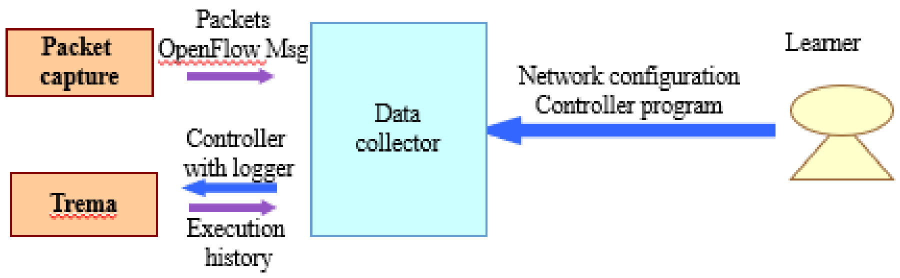
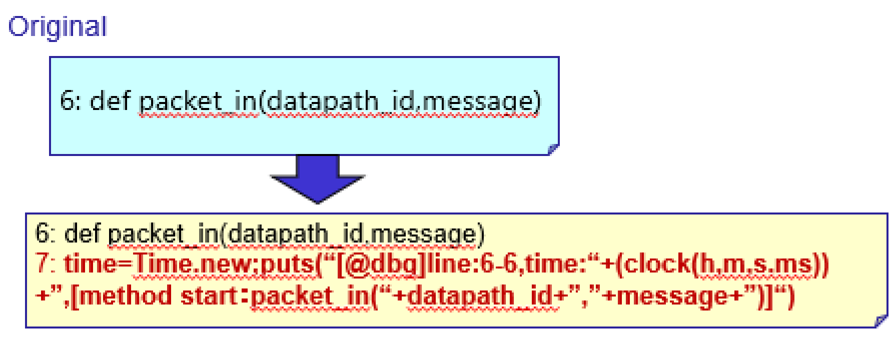


Figure 5.5 Flow of Data Collection Function

In the background, we add logging functions to controller program. And then, we add a standard output function (prefix, line number, and clock) to all statements, and finally we add a standard output function (packet_in 's parameters) to the first line in packet_in function

Describes the data collection function. In this function, input data is created to obtain the execution history. Execution history collection programs are regulated by prefix line numbers. As an example, if the original executable statement is a packetin method definition statement, add a description to output the argument in addition to the prefix, line number, and time as shown in figure 5.6 below [57].



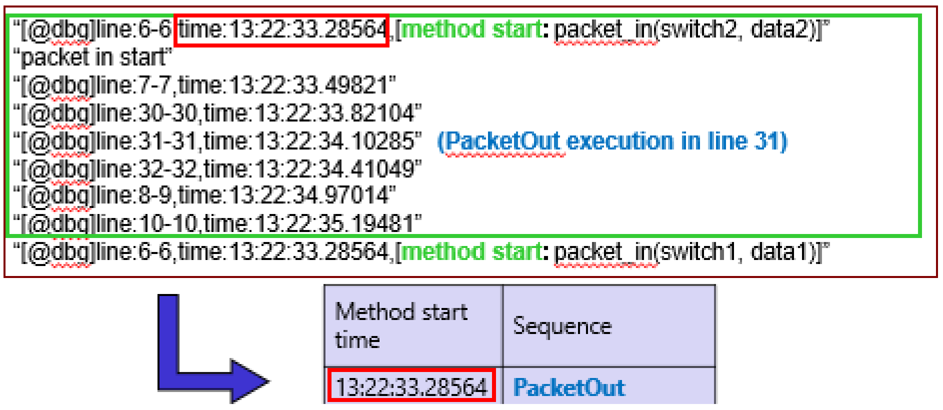
```
6: def packet_in(datapath_id,message)
```

```
6: def packet_in(datapath_id,message)
7: time=Time.new;puts("[@dbg]line:6-6,time:"+(clock(h,m,s,ms))
+",[method start:packet_in("+datapath_id+","+message+")]")
```

Figure 5.6 Data Collector Generate Controller with Logger

In figure 5.7, the data collector analyzes execution history into the following order;

1. Divide execution history into methods
2. Find execution start time of methods
3. Find sequences of executions of PacketOut and FlowMod



```
"[@dbg]line:6-6,time:13:22:33.28564,[method start: packet_in(switch2, data2)]"
"packet in start"
"[@dbg]line:7-7,time:13:22:33.49821"
"[@dbg]line:30-30,time:13:22:33.82104"
"[@dbg]line:31-31,time:13:22:34.10285"  (PacketOut execution in line 31)
"[@dbg]line:32-32,time:13:22:34.41049"
"[@dbg]line:8-9,time:13:22:34.97014"
"[@dbg]line:10-10,time:13:22:35.19481"
"[@dbg]line:6-6,time:13:22:33.28564,[method start: packet_in(switch1, data1)]"
```

Method start time	Sequence
13:22:33.28564	PacketOut

Figure 5.7 Data Collector Analyze Execution History

The data collection function also associates PacketIn and controller executable statements. The controller does not always process PacketIn in the order received. For this reason, the following matching is performed. As an example, when three packetins are sent from switches 1 and 2 and the packet-in method is executed three times, matching is performed as shown in this color coding.

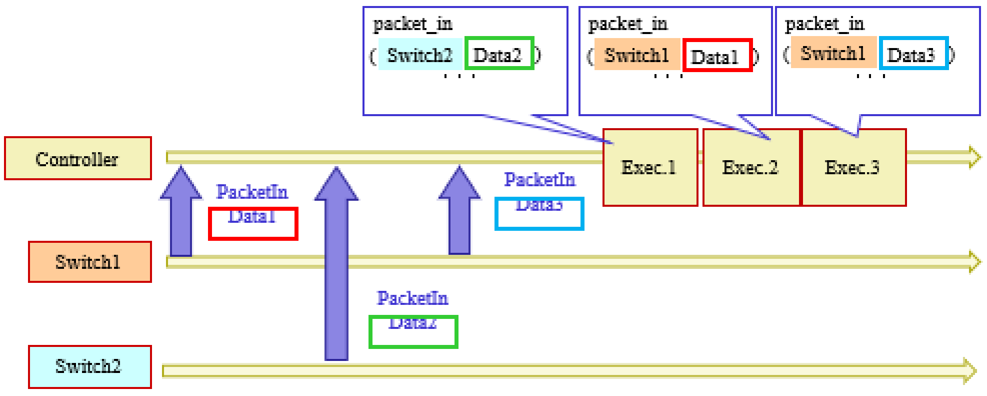


Figure 5.8 Data Collector Corresponds PacketIn to Executed Statements in Controller

Also, Packetout and Flowmod are associated with controller executable statements. In the controller, PacketOut and FlowMod are sent through the controller to switchs. As an example, if a send execution statement is executed in this way with three packet-in methods and packetout flowmod is sent in this way, first, execution 1 and execution 2, 3 are separated from the first argument. And it associates like this color classification from each execution order and transmission order.

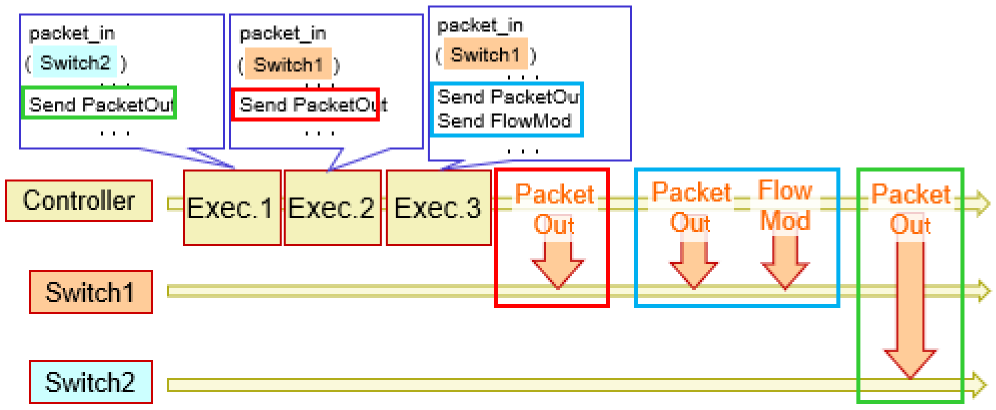


Figure 5.9 Data Collector Corresponds PacketOut and FlowMod to Executed Statements in Controller

5.4.4 Time Series Flow Table Reproducer

After the data collection function generates the following logs to the log database, time series flow table reproducing function has started. It uses the dependency of Openflow Information (OF) log and then generates flow tables (FT) to the database.

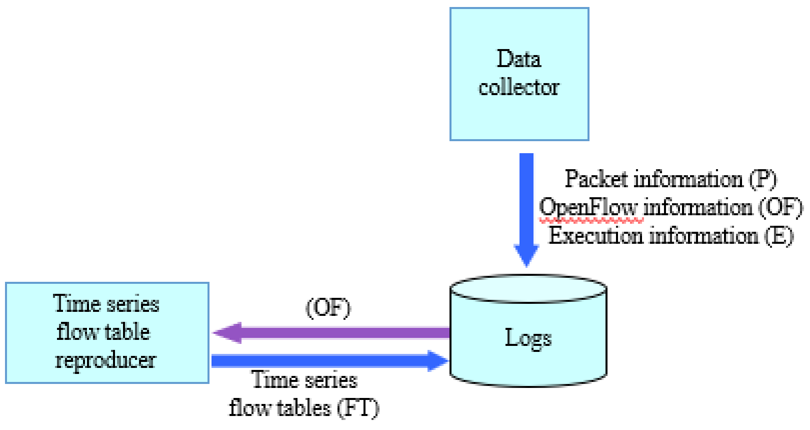

Data collector
Packet information (P)
OpenFlow information (OF)
Execution information (E)
Time series flow table reproducer
(OF)
Logs
Time series flow tables (FT)

Figure 5.10 Time Series Flow Table Reproducer

5.4.5 Delivery Route Estimator

Route selection histories are Logs from receiving packets to choosing output devices in switch.

Here, we estimate delivery routes based on route selection histories

1. Find route selection history "A" whose sending device is a host
2. Find route selection history "B" that satisfies the following conditions
    a. Reception device of A == Sending device of B
    b. Sending device of A == Reception device of B
    c. Packet sent from A == Packet received by B
    d. Reception time of A < Reception time of B
3. If B's reception device is not a host, then assign B to A and go back to step 2.

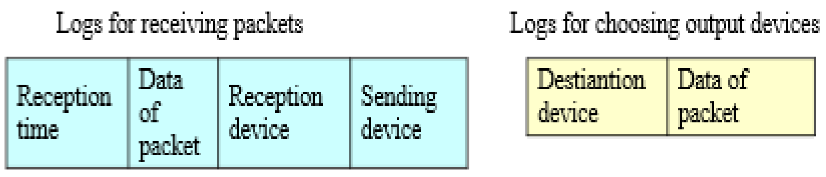


Figure 5.11 Route Selection Histories

It describes the transmission path estimation function. This function estimates the transmission route from the route selection history. The route selection histories are represented by Figure 5.11. This is created by the route selection imitation function.

5.5 Chapter Summary

The implementation of the proposed system with the logical testbed design of SDN network construction exercises is described in this chapter. Here we described the SDN network topology, hardware and software requirements used in this system and the installation and implementation of the system in explained in detail. The experimental results of this testbed evaluation with multiple scenarios and multiple methods will be presented in the next chapter.

CHAPTER 6

EXPERIMENTS AND RESULTS

This chapter carried out the experiments of the system by using three different scenarios with three different controller programs. Firstly, this chapter discussed the experiments results of each by conducting Trema Openflow Controller with and without logging code. Then, it also discussed the experimental results of each controller program with different network construction exercises.

6.1 Experiment Methods

In the proposed system, VMWare Workstation and Ubuntu16.04 LTS are used for the testbed area.

6.1.1 The Main Files and Functions Used in the System Scenario

The following files are used to run the system;

1. trema_netwdbg2.rb

   where, it runs tcpdump processes to a SDN controller and a SDN network.
2. testf2 (source file: pcapanalize_f2.c)

   where, it analyzes tcpdump outputs (logfiles) and then writes packet information and OpenFlow information to files in ./dbglog
3. contanalize_f2.rb

   where, it analyzes execution statements in controllers (logfiles) and then writes the results to files in ./dbglog
4. data_analize.rb

   where, it generates time series flow tables, route selection histories, and delivery route information, and then writes them to files in ./dbglog respectively

6.1.2 Library Files Used in the System

1. External library for Ruby (./clib)

   pktread.c   where it is used for analyzing packets and OpenFlow

   Note: A class name in ruby is PktRead. Refer to section PktRead library.

2. Required files for starting system
    - make-dc.sh: a shell script file for building networks
    - dc-data.txt: a file including network configurations used in item A
    - trema-test1.rb: a controller program in Trema

6.1.3 Operational Flow of the System

Destroy the networks built before based on the shell script file “make-dc.sh” before creating the new network construction exercises

$ ./make-dc.sh del

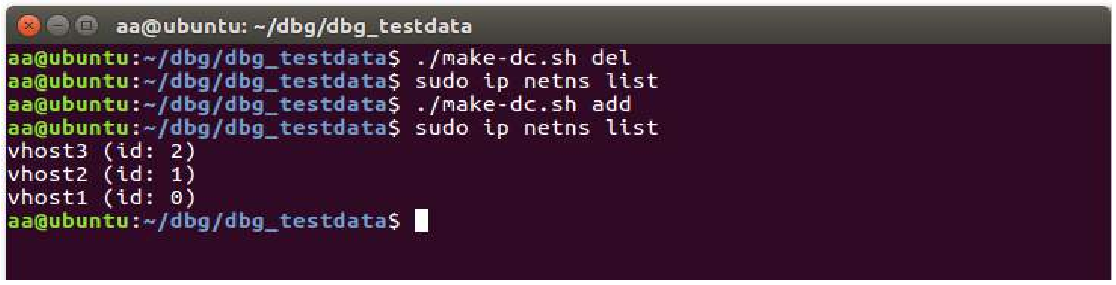
And then build a network with the shell script and the following command;

$ ./make-dc.sh add

Check the networks built with the following command;

$ sudo ip netns list

where we can see the networks build by shell script as shown in Figure 6.1.



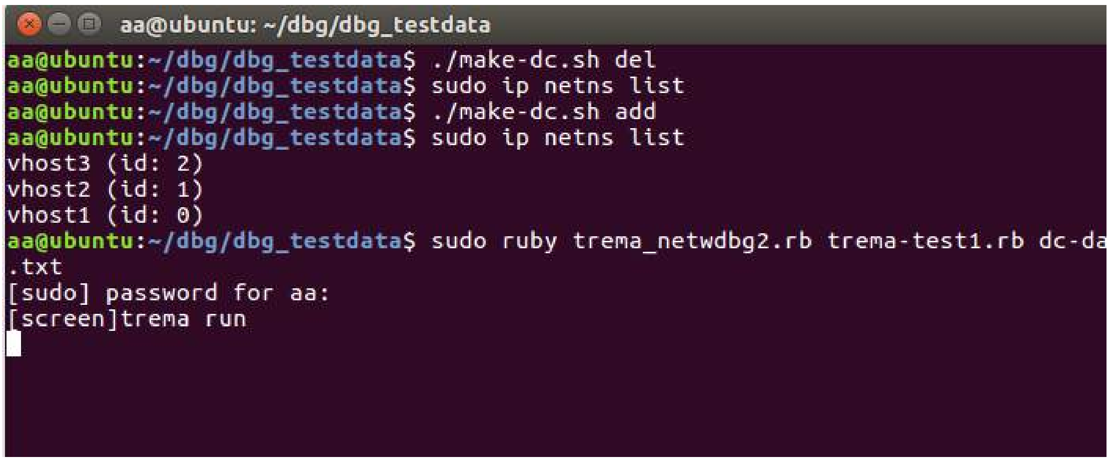
```
aa@ubuntu: ~/dbg/dbg_testdata
aa@ubuntu:~/dbg/dbg_testdata$ ./make-dc.sh del
aa@ubuntu:~/dbg/dbg_testdata$ sudo ip netns list
aa@ubuntu:~/dbg/dbg_testdata$ ./make-dc.sh add
aa@ubuntu:~/dbg/dbg_testdata$ sudo ip netns list
vhost3 (id: 2)
vhost2 (id: 1)
vhost1 (id: 0)
aa@ubuntu:~/dbg/dbg_testdata$
```

Figure 6.1 Building Virtual Networks

After that, run Trema with the following command to execute the system

$ sudo ruby trema_netwdbg2.rb trema-test1.rb dc-data.txt

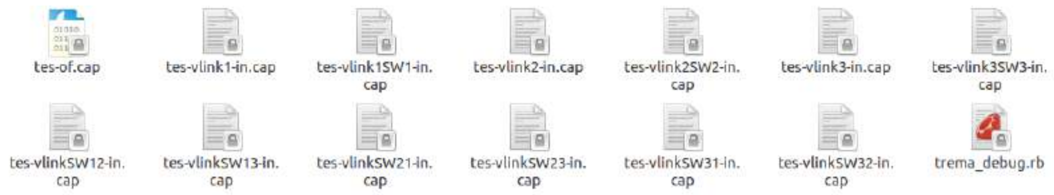
Note: We need to wait approximately 5 seconds after "[screen] trema run" appears on the screen. This is because the time is spent for connecting between the controller and the switch.



```
aa@ubuntu: ~/dbg/dbg_testdata
aa@ubuntu:~/dbg/dbg_testdata$ ./make-dc.sh del
aa@ubuntu:~/dbg/dbg_testdata$ sudo ip netns list
aa@ubuntu:~/dbg/dbg_testdata$ ./make-dc.sh add
aa@ubuntu:~/dbg/dbg_testdata$ sudo ip netns list
vhost3 (id: 2)
vhost2 (id: 1)
vhost1 (id: 0)
aa@ubuntu:~/dbg/dbg_testdata$ sudo ruby trema_netwdbg2.rb trema-test1.rb dc-da
.txt
[sudo] password for aa:
[screen]trema run
```

Figure 6.2 Running Trema with Controller Programs

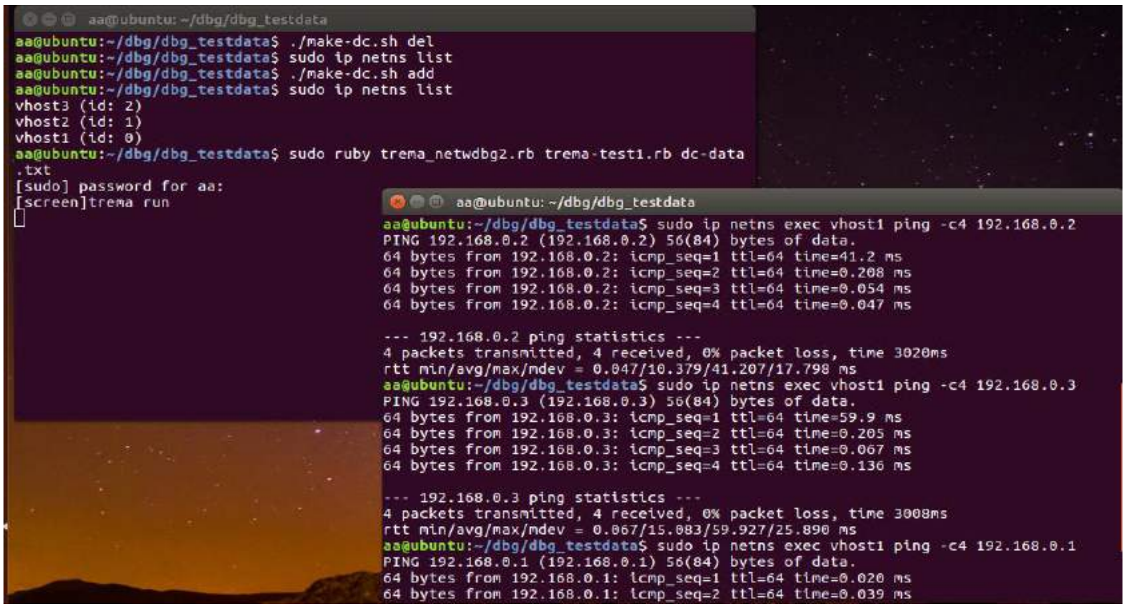
The file network debugger runs tcpdump processes between SDN controller program and SDN network.



```
tes-of.cap   tes-vlink1-in.cap   tes-vlink1SW1-in.cap   tes-vlink2-in.cap   tes-vlink2SW2-in.cap   tes-vlink3-in.cap   tes-vlink3SW3-in.cap
tes-vlinkSW12-in.cap   tes-vlinkSW13-in.cap   tes-vlinkSW21-in.cap   tes-vlinkSW23-in.cap   tes-vlinkSW31-in.cap   tes-vlinkSW32-in.cap   trema_debug.rb
```

Figure 6.3 Output of TCPDUMP Files in the Log Database

Then, try to make network construction test by doing ping with the command ip
$ sudo ip netns exec vhost1 ping 192.168.0.2.



```
aa@ubuntu: ~/dbg/dbg_testdata
aa@ubuntu:~/dbg/dbg_testdata$ ./make-dc.sh del
aa@ubuntu:~/dbg/dbg_testdata$ sudo ip netns list
aa@ubuntu:~/dbg/dbg_testdata$ ./make-dc.sh add
aa@ubuntu:~/dbg/dbg_testdata$ sudo ip netns list
vhost3 (id: 2)
vhost2 (id: 1)
vhost1 (id: 0)
aa@ubuntu:~/dbg/dbg_testdata$ sudo ruby trema_netwdbg2.rb trema-test1.rb dc-data
.txt
[sudo] password for aa:
[screen]trema run
```

```
aa@ubuntu: ~/dbg/dbg_testdata
aa@ubuntu:~/dbg/dbg_testdata$ sudo ip netns exec vhost1 ping -c4 192.168.0.2
PING 192.168.0.2 (192.168.0.2) 56(84) bytes of data.
64 bytes from 192.168.0.2: icmp_seq=1 ttl=64 time=41.2 ms
64 bytes from 192.168.0.2: icmp_seq=2 ttl=64 time=0.208 ms
64 bytes from 192.168.0.2: icmp_seq=3 ttl=64 time=0.054 ms
64 bytes from 192.168.0.2: icmp_seq=4 ttl=64 time=0.047 ms

--- 192.168.0.2 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3020ms
rtt min/avg/max/mdev = 0.047/10.379/41.207/17.798 ms
aa@ubuntu:~/dbg/dbg_testdata$ sudo ip netns exec vhost1 ping -c4 192.168.0.3
PING 192.168.0.3 (192.168.0.3) 56(84) bytes of data.
64 bytes from 192.168.0.3: icmp_seq=1 ttl=64 time=59.9 ms
64 bytes from 192.168.0.3: icmp_seq=2 ttl=64 time=0.205 ms
64 bytes from 192.168.0.3: icmp_seq=3 ttl=64 time=0.067 ms
64 bytes from 192.168.0.3: icmp_seq=4 ttl=64 time=0.136 ms

--- 192.168.0.3 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3008ms
rtt min/avg/max/mdev = 0.067/15.083/59.927/25.890 ms
aa@ubuntu:~/dbg/dbg_testdata$ sudo ip netns exec vhost1 ping -c4 192.168.0.1
PING 192.168.0.1 (192.168.0.1) 56(84) bytes of data.
64 bytes from 192.168.0.1: icmp_seq=1 ttl=64 time=0.020 ms
64 bytes from 192.168.0.1: icmp_seq=2 ttl=64 time=0.039 ms
```

Figure 6.4 Making Network Communication Test

After testing the network, stop trema by running the following command

$ sudo trema killall

6.1.4 Analyzing the System Output Logs

After testing ping communication with the controller programs, we analyze the packets and analyze the dependency between communication routes and flow tables and controllers.

To analyse packets that are generated in section 6.1.3, we run the following files

$ ./testf2

After that, the system generates the following log files into the database

1. pkt_data.txt: packet information
2. pkt_ofdata.txt: OpenFlow information
3. pkt_datahead.txt: meta data for packet information and OpenFlow information

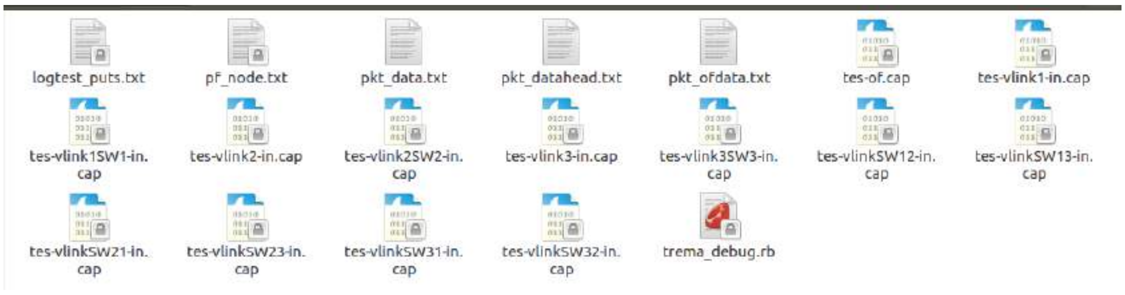


Figure 6.5 Output Logs for Packet Information and OpenFlow Information

To analyze execution statements, we run the ruby file naming "contanalize_f2.rb" with the following command;

$ ruby contanalize_f2.rb

After that, the system generates the file naming

log_data_2t.txt: execution information

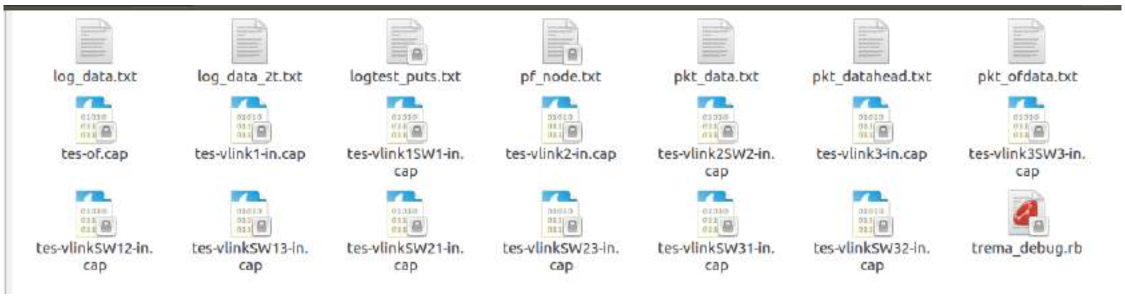


Figure 6.6 Output Logs for Execution Statement Information

To generate time series flow table, route selection histories and delivery route information, we run the ruby file naming "data_analize.rb" and write the data int dc-data.txt by the following command;

```
$ ruby data_analize.rb dc-data.txt
```

After that, the system generates the file naming

1. tft_data.txt: a time series flow table
2. rsd_data.txt: a route selection history
3. prd_data.txt: delivery route information

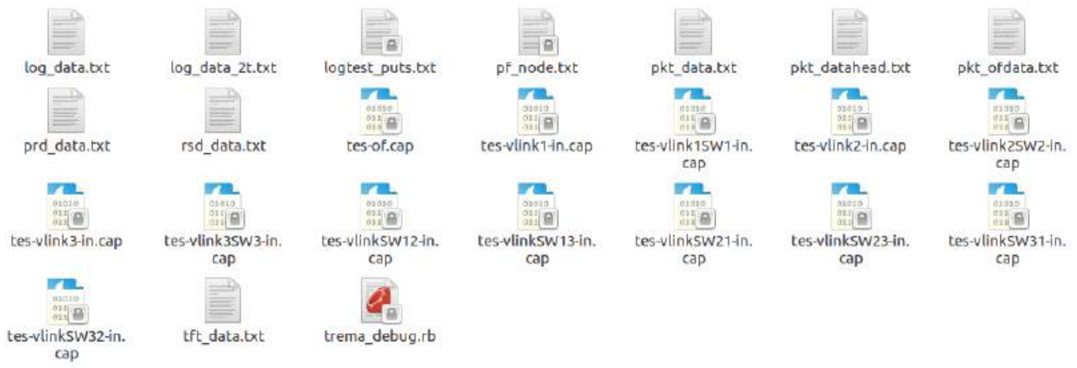


Figure 6.7 All Generated Output Log Files

6.2 Running Testbed Evaluation Environment and Visualizing the System

To highlight the outcome of our proposed system, we compare and analyze three controller programs in two different scenarios. As an evaluation experiment, these three network configurations were prepared, and using the controller program that succeeded in continuity, configurations A to C were configured as in the actual exercise. The execution environment criteria are as follows;

- Controller programs in which ping communications are successful
- Do ping communications between all pairs of hosts
- All hosts in each pair send three ICMP echo requests

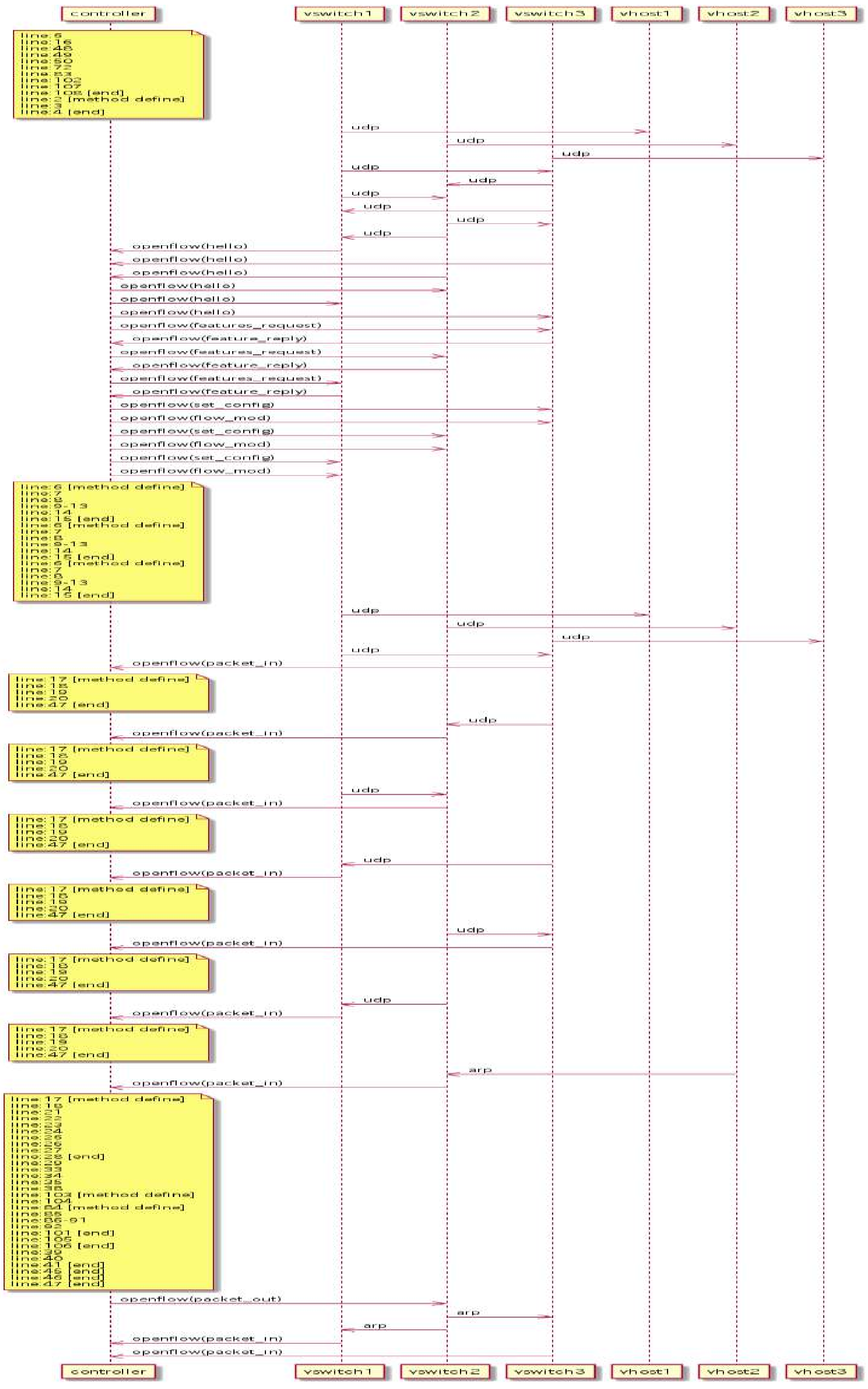
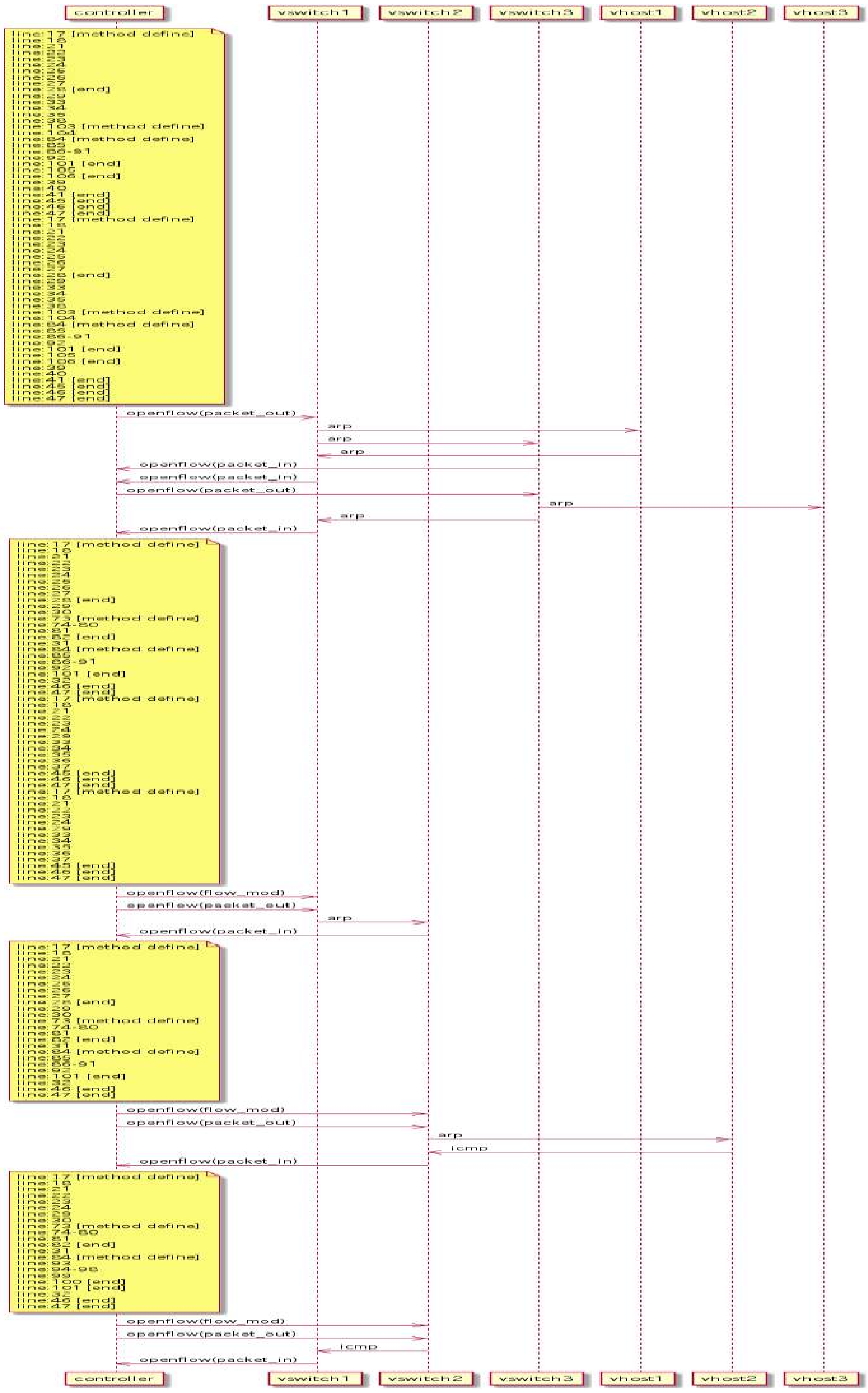


Figure 6.8 Output Event Result Testing with Controller Program 1



```
line:17 [method define]
line:18
line:21
line:22
line:23
line:24
line:25
line:26
line:27
line:28 [end]
line:29
line:33
line:34
line:35
line:36
line:103 [method define]
line:104
line:84 [method define]
line:85
line:86-91
line:92
line:101 [end]
line:105
line:106 [end]
line:39
line:40
line:41 [end]
line:45 [end]
line:46 [end]
line:47 [end]
line:17 [method define]
line:18
line:21
line:22
line:23
line:24
line:25
line:26
line:27
line:28 [end]
line:29
line:33
line:34
line:35
line:36
line:103 [method define]
line:104
line:84 [method define]
line:85
line:86-91
line:92
line:101 [end]
line:105
line:106 [end]
line:39
line:40
line:41 [end]
line:45 [end]
line:46 [end]
line:47 [end]
```

```
line:17 [method define]
line:18
line:21
line:22
line:23
line:24
line:25
line:26
line:27
line:28 [end]
line:29
line:30
line:73 [method define]
line:74-80
line:81
line:82 [end]
line:31
line:84 [method define]
line:85
line:86-91
line:92
line:101 [end]
line:32
line:46 [end]
line:47 [end]
line:17 [method define]
line:18
line:21
line:22
line:23
line:24
line:29
line:33
line:34
line:35
line:36
line:37
line:45 [end]
line:46 [end]
line:47 [end]
line:17 [method define]
line:18
line:21
line:22
line:23
line:24
line:29
line:33
line:34
line:35
line:36
line:37
line:45 [end]
line:46 [end]
line:47 [end]
```

```
line:17 [method define]
line:18
line:21
line:22
line:23
line:24
line:25
line:26
line:27
line:28 [end]
line:29
line:30
line:73 [method define]
line:74-80
line:81
line:82 [end]
line:31
line:84 [method define]
line:85
line:86-91
line:92
line:101 [end]
line:32
line:46 [end]
line:47 [end]
```

```
line:17 [method define]
line:18
line:21
line:22
line:23
line:24
line:29
line:30
line:73 [method define]
line:74-80
line:81
line:82 [end]
line:31
line:84 [method define]
line:93
line:94-98
line:99
line:100 [end]
line:101 [end]
line:32
line:46 [end]
line:47 [end]
```

Figure 6.19 Output Event Result Testing with Controller Program 2

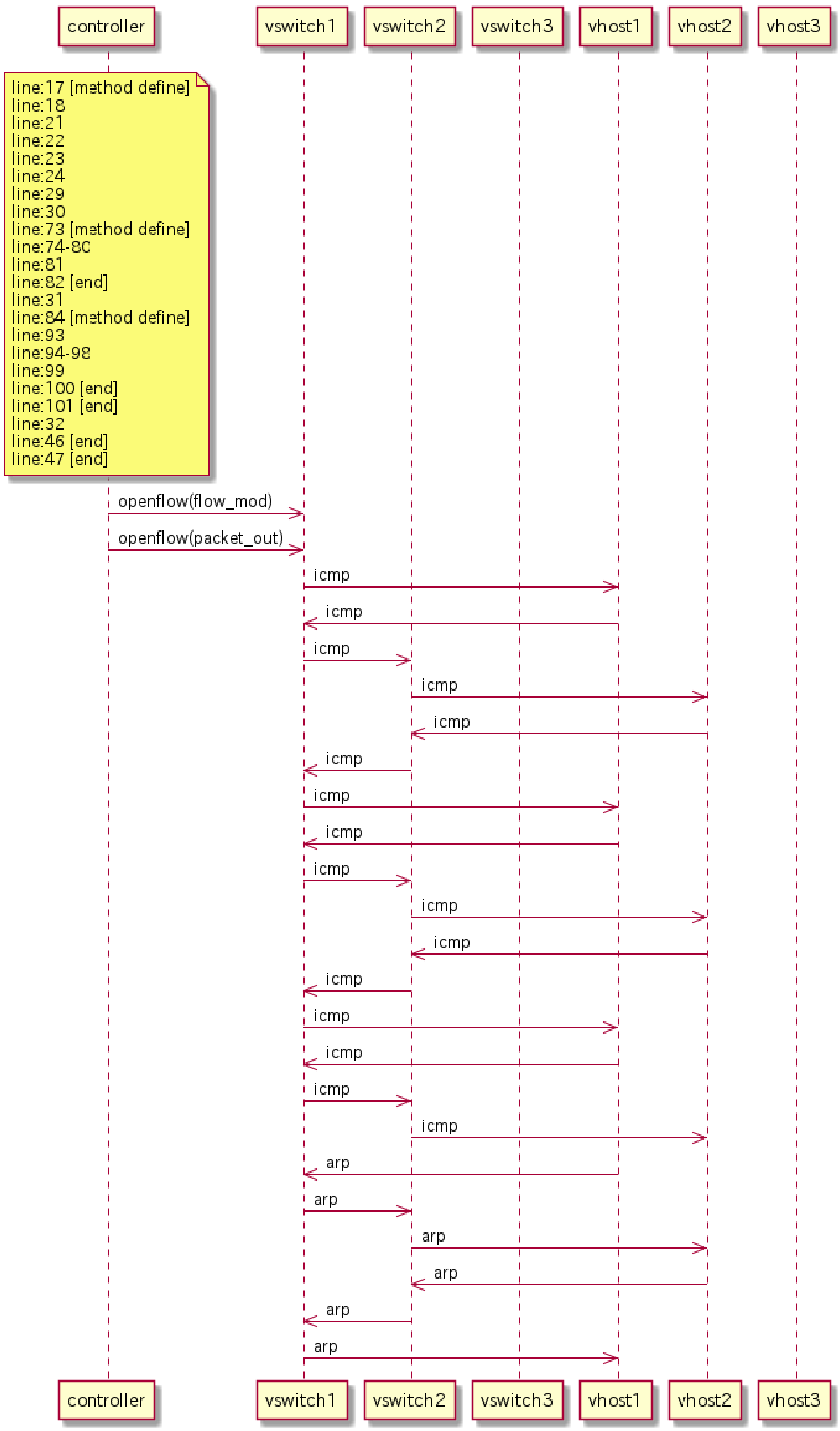


Figure 6.10 Output Event Result Testing with Controller Program 3

6.3 Performance Evaluation

In this system, the evaluation environment includes;

Host OS: Windows 10,

Host CPU: Intel(R) Core (TM) i7-6500U CPU @ 2.59GHz,

Virtual Machine: VMware Workstation 15 Player,

Guest OS: Ubuntu 16.04LTS and Guest memory assignment is 1 GB.

The visualizer and data analyzer are implemented with Ruby and C language, and Plant UML is used to convert the description to graphics. It might make it possible for us to achieve our goals. The analysis results are shown in table 1.

Table 6.1 Analysis Result

Controller Program	Controller Evets	Network Reachability	Packet Information (KB)	Openflow Information	Executed Statement Information
1	High	Yes	18	38	49
2	Medium	Yes	22	24	23
3	Low	Yes	25	20	24

The analysis results of the system are described in table 6.1 where the system was tested with three different controller programs to be tested with the status of controller events, the reachability of network communication and the amount of data in packet information, OpenFlow information and executed statement information.

Figure 6.9 to 6.11 are the results of testbed configuration with three different controller programs. In the Figures, the upper and lower squares are the device names, and the yellow color-coded box is the part where the line number that was executed.

6.4 Chapter Summary

In this chapter, the experimental results of a dependency analyzing system has been described for learners who learn and writing controllers with SDN network construction exercises in universities, research labs and also useful for on job training in business enterprises. The experiments for this research are described in this chapter with step by step execution with the analysis results.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

This dissertation implemented a Dependency Analyzing System for communication activities in Network Construction Exercises using Trema. In this research, we demonstrated significant benefits in aiding students to debug and develop their Software Defined Networking (SDN) controller programs. The system provides a detailed analysis of the causal connections between packet transmissions in the data plane, control plane activities, and the execution of controller program statements. System represents a significant advancement in the educational tools available for teaching SDN concepts. By providing a robust framework for analyzing and debugging network communication activities, it helps bridge the gap between theoretical knowledge and practical implementation, ultimately contributing to the development of more proficient SDN practitioners. Key findings and conclusions from this study include:

Improved Debugging Efficiency: By leveraging the Dependency Analyzing System, students were better able to identify and correct errors in their controller programs. The system's ability to trace packet flows and correlate them with specific statements in the controller code proved invaluable in the debugging process.

Enhanced Understanding of SDN Operations: The system facilitated a deeper understanding of SDN concepts among students. By visualizing the interactions between the data plane, control plane, and controller programs, students gained a clearer insight into the operational mechanics of SDN.

Identification of Common Error Patterns: The analysis revealed common patterns in the errors made by students, such as misconfigured network settings or incorrect controller logic. This information can be used to refine teaching materials and provide targeted guidance to future students.

Increased Success Rates: With the support of the Dependency Analyzing System, a higher percentage of students were able to meet the exercise requirements and build functional SDN networks. This indicates the system's effectiveness in enhancing the learning outcomes of network construction exercises.

Scalability and Adaptability: The system showed potential for scalability and adaptability to different SDN frameworks and network configurations. This flexibility makes it a valuable tool for a wide range of educational and practical applications in SDN development.

7.1 Advantages of the System

The implementation of the system using Software Defined Networking (SDN) offers several advantages, particularly in the context of the modern information age and the rapid development of networking and virtualization technologies. Here are the advantages, based on the provided factors:

Promotion of Research and Innovation: Academic and Research Focus: SDN has become a prominent area for research and innovation in academic and research fields. Universities and research labs serve as hubs for this innovation, driving advancements that can rapidly influence industrial practices. By engaging with SDN, these institutions can contribute to the development of cutting-edge networking technologies.

Accelerated Change: Innovations originating from academia and research organizations can significantly accelerate the rate of technological change in industries. This collaborative environment fosters the exchange of ideas and the development of practical solutions that can be quickly adopted in real-world scenarios.

Enhanced e-Learning Opportunities: Educational Exercises: SDN construction exercises have been integrated into e-Learning platforms, providing novice learners with hands-on experience in building and managing networks. These exercises help learners understand the principles of SDN and gain practical skills that are essential in the modern networking landscape.

Practical Understanding: When performing network construction exercises, learners often struggle to understand network behavior and meet communication data requirements. The system addresses this issue by allowing learners to construct SDN networks using tools like Trema and the OpenFlow protocol for communication between controllers and switches.

Debugging and Problem-Solving Assistance: Visual Clues for Debugging: A significant challenge for learners is identifying and fixing bugs in their network settings. Issues such as the inability to find delivery routes with ping, switches lacking logging functions for output port selection rules, and Trema's inability to locate execution statements for setting rules can hinder learning progress. The system provides visual clues to help learners narrow down executed statements that cause incorrect communication, facilitating easier debugging and problem-solving.

Improved Learning Outcomes: By offering visual debugging aids, the system enhances learners' ability to identify and resolve issues in their network configurations. This leads to a deeper understanding of SDN principles and improves overall learning outcomes.

Hands-On Experience with Advanced Technologies: Use of OpenFlow Protocol: The system utilizes the OpenFlow protocol for communication between controllers and switches, giving learners hands-on experience with a key component of SDN. This practical exposure helps learners become familiar with advanced networking technologies and prepares them for future careers in the field.

Building Real-World Skills: Engaging with SDN exercises and troubleshooting real network issues allows learners to build valuable skills that are directly applicable in the networking industry. This experience can make them more competitive in the job market and better equipped to handle complex networking tasks.

The advantages of implementing this system using SDN include fostering research and innovation in academic settings, enhancing e-Learning opportunities, providing effective debugging and problem-solving tools, and offering hands-on experience with advanced networking technologies. These benefits collectively contribute to a more effective and comprehensive learning experience for students and researchers in the field of networking.

7.2 Limitations of the System

The limitations of the system using SDN include the complexity faced by novice learners, challenges associated with debugging, partial effectiveness of visual aids, technical and resource constraints, scalability issues, and the potential gap between educational exercises and real-world application. Addressing these limitations is essential to enhance the system's educational value and practical relevance.

7.3 Future Work

In this proposed system, there are four main parts; firstly, mentioned the flow of the exercise using in the proposed system is mentioned, secondly, the system overview is described and the description of the logging code was inserted and finally collect raw data and show them in sequence diagram to help learners in visual way of data collection. A function is classified to develop for estimating routes in data planes from capture packets. Then that function will help students to detect incorrect communication routes in their network. The system performance and effectiveness for students' debugging will also be evaluated. And, it also plans to develop functions in order to analyze network traffic for security.

AUTHOR’S PUBLICATIONS

[p1] Hlwam Maint Htet, Khin Than Mya, “A Transparent Tax Data Access Control System Based on Blockchain” 17th International Conference on Computer Applications (ICCA 2019)”, pages 107-111), Yangon, Myanmar on February 27-28, 2019.

[p2] Hlwam Maint Htet, “DEPENDENCY ANALYSIS SYSTEM TO NARROW DOWN MISSETTINGS IN SDN CONSTRUCTION EXERCISES USING TREMA”, Indian Journal of Computer Science and Engineering, VOLUME 14 ISSUE 3 May-June 2024, (pp.358-363).

[p3] Hlwam Maint Htet, Amy Tun, “TREMA BASED DEPENDENCY ANALYSIS SYSTEM FOR LEARNERS IN BUILDING SDN NETWORK CONSTRUCTION EXERCISES”, Indian Journal of Computer Science and Engineering, VOLUME 14 ISSUE 3 May-June 2024, (pp.364-369).

BIBLIOGRAPHY

[1] A. Tootoonchian and Y. Ganjali, “Hyperflow: A Distributed Control Plane for Openflow,” In the Proceedings of the Internet Network Management Conference on Research on Enterprise Networking (Vol. 3), April 2010.

[2] Antonakakis, M., et al. (2014). GNS3 Network Simulator. *Software Review

[3] Berde, P., Gerola, M., Hart, J. K., Higuchi, Y., Kobayashi, M., Lantz, B., & Snow, W. (2014). "ONOS: Towards an open, distributed SDN OS." Proceedings of the third workshop on Hot topics in software defined networking, 1-6.

[4] Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., ... & Walker, D. (2014). "P4: Programming protocol-independent packet processors." ACM SIGCOMM Computer Communication Review, 44(3), 87-95.

[5] Cardenas, R., & Perkins, S., & Pfleeger, C. (2010). Virtual Laboratories for Computer Security Education. *Proceedings of the 2010 ACM Conference on Computer and Communications Security

[6] Cisco. (n.d.). *Packet Tracer*. Retrieved from [Cisco Packet Tracer](https://www.netacad.com/courses/packet-tracer).

[7] Curtis, A. R., Mogul, J. C., Tourrilhes, J, Yalagandula, P., Sharma, P., & Banerjee, S. (2011). "DevoFlow: Scaling flow management for high-performance networks." ACM SIGCOMM Computer Communication Review, 41(4), 254-265.

[8] Fares, M. A., Radhakrishnan, S., Raghavan, B., Huang, N., & Vahdat, A. (2010). "Hedera: Dynamic Flow Scheduling for Data Center Networks." Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation (NSDI).

[9] Forte, G., & Mongiello, M. (2009). "An Educational Testbed for Teaching Networking Based on Netkit." Journal of Educational Technology & Society, 12(3), 152-162.

[10] G. Liang and W. Li, “A Novel Industrial Control Architecture Based on Software Defined Network,” Measurement and Control 51, no. 7-8 (2018): 360-367.

[11] Handigol, N., Heller, B., Jeyakumar, V, Lantz, B., & McKeown, N. (2012). Reproducible Network Experiments using Container-Based Emulation. *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies (CoNEXT '12)

[12] HIDEyuki Shimonishi, Yasuhito Takamiya, Yasunobu Chiba, Kazushi Sugyo, Youichi Hatano, Kentaro Sonoda, Kazuya Suzuki, Daisuke Kotani, and Ippei Akiyoshi, “Programmable Network Using OpenFlow for Network Researches and Experiments”, ICMU 2012.

[13] Hongyi Zeng, Peyman Kazemian, George Varghese§, Nick McKeown, “Automatic Test Packet Generation”, IEEE/ACM TRANSACTIONS ON NETWORKING, VOL. 22, 2014.

[14] https://www.wikipedia.org/

[15] https://www.wireshark.org/

[16] Hu, Y. C., Patel, M., Sabella, D., Sprecher, N., & Young, V. (2015). "Mobile Edge Computing—A Key Technology Towards 5G." ETSI White Paper, 11, 1-16.

[17] I. F. Akyildiz, A. Lee, P. Wang, M. Luo, and W. Chou, “A Roadmap for Traffic Engineering in SDN-OpenFlow Networks,” Computer Networks, 71, pp.1-30, October 2014.

[18] ip-netns(8) - Linux manual page: http://man7.org/linux/man- pages/man8/ip-netns.8.html,

[19] Jain, R., & Paul, S. (2013). "Network virtualization and software-defined networking for cloud computing: a survey." IEEE Communications Magazine, 51(11), 24-31.

[20] Jarraya, Y., Madi, T., & Debbabi, M. (2014). "A Survey and A Layered Taxonomy of Software-Defined Networking." IEEE Communications Surveys & Tutorials, 16(4), 1955-1980.

[21] Jarschel, M., Oechsner, S., Schlosser, D., Pries, R., Goll, S., & Tran-Gia, P. (2011). "Modeling and Performance Evaluation of An Openflow Architecture." Proceedings of the 23rd International Teletraffic Congress, 1-7.

[22] Jones, A., & Wang, L. (2018). Intelligent Tutoring Systems for Network Education: A Review. IEEE Transactions on Learning Technologies, 14(2), 87-99

[23] Kreutz, D., Ramos, F. M. V., Verissimo, P. E., Rothenberg, C. E., Azodolmolky, S., & Uhlig, S. (2015). "Software-Defined Networking: A Comprehensive Survey." Proceedings of the IEEE, 103(1), 14-76.

[24] Kreutz, D., Ramos, F. M. V., Verissimo, P. E., Rothenberg, C. E., Azodolmolky, S., & Uhlig, S. (2015). Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1), 14-76.

[25] Lantz, B., Heller, B., & McKeown, N. (2010). "A Network in a Laptop: Rapid Prototyping for Software-Defined Networks." *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets '10).

[26] Lantz, B., Heller, S., & McKeown, N. (2010). A Network in a Laptop: Rapid Prototyping for Software-Defined Networks. *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets '10

[27] M. A. Fares, S. Radhakrishnan, B. Raghavan, N. Huang, A. Vahdat, “Hedera: Dynamic Flow Scheduling for Data Center Networks,” In the Proceedings of Networked Systems Design and Implementation Symposium, vol. 10, April 2010.

[28] M. Jarschel, S. Oechsner, D. Schlosser, R. Pries, S. Goll, and P. Tran-Gia, “Modeling and Performance Evaluation of An Openflow Architecture,” In the Proceedings of 23rd International Tele Traffic Congr., pp. 1–7, September 2011.

[29] Martin, F., et al. (2018). Enhancing Network Education with GNS3. *Journal of Network and Systems Management

[30] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., ... & Turner, J. (2008). "OpenFlow: Enabling innovation in campus networks." ACM SIGCOMM Computer Communication Review, 38(2), 69-74.

[31] Medved, J., Varga, R., Tkacik, A., & Gray, K. (2014). OpenDaylight: Towards a model-driven SDN controller architecture. *Proceedings of the 2014 IEEE 15th International Symposium on World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, 1-6.

[32] Mongiello, M., & Forte, G. (2013). The Netkit Network Emulation System: Overview and Operational Details. *Technical Report*, University of Bari

[33] Monsanto, C., Reich, J., Foster, N., Rexford, J., & Walker, D. (2013). "Composing software-defined networks." *NSDI*, 13, 1-14.

[34] Netkit. (n.d.). Netkit: Network Emulation Toolkit*. Retrieved from [Netkit official website] (http://www.netkit.org).

[35] Nunes, B. A. A., Mendonca, M., Nguyen, X. N., Obraczka, K., & Turletti, T. (2014). "A Survey of Software-Defined Networking: Past, Present, and Future of

Programmable Networks." IEEE Communications Surveys & Tutorials, 16(3), 1617-1634.

[36] Open Networking Foundation, "Software-Defined Networking: The New Norm for Networks," ONF White Paper 2, pp : 2-6, 2012.

[37] Open vSwitch: https://www.openvswitch.org/,

[38] OpenFlow Switch Specification Version 1.5.1 (Protocol version 0x06), Open Networking Foundation, March 2015, [Online] Available: https://www.opennetworking.org/wp-content/uploads/2014/10/ openflow-switch-v1.5.1.pdf

[39] pages/man4/veth.4.html,

[40] Phemius, K., Bouet, M., & Leguay, J. (2014). "DISCO: Distributed multi-domain SDN controllers." 2014 IEEE Network Operations and Management Symposium (NOMS), 1-4.

[41] Phemius, K., Bouet, M., & Leguay, J. (2014). "DISCO: Distributed multi-domain SDN controllers." 2014 IEEE Network Operations and Management Symposium (NOMS), 1-4.

[42] PlantUML: https://plantuml.com/

[43] R. Jmal and L. C. Fourati, "Implementing Shortest Path Routing Mechanism Using Openflow POX Controller," In the Proceedings of IEEE International Symposium on Networks, Computers and Communications, pp. 1-6, June 2014.

[44] Ricciato, F., Mongiello, M., & Forte, G. (2008). "Netkit: Easy Emulation of Complex Networks on Inexpensive Hardware." *Proceedings of the 2008 ACM Conference on SIGCOMM.

[45] Smith, T., & Brown, H. (2009). Peer Learning in Network Education: Benefits and Challenges. *Computers & Education*, 130, 76-89.

[46] Stalling, W. (2013). "Software-Defined Networks and OpenFlow." The Internet Protocol Journal, 16(1), 2-18.

[47] Takashi Yokoyama, Hisayoshi Kunimune, Masaaki Niimura, Shinshu UniversityJapan, "Determining Learning Status in SDN Construction Exercises",E-Learn 2016 - Washington, DC, United States, November 14-16, 2016.

[48] Toshio Tonouchi Satoshi Yamazaki , “A fast method of verifying network routing with back-trace header space analysis”, 2015 IEEE.

[49] “Trema: A Brief Introduction and Tutorial”, Shyhji Ishii, APAN 32nd Future Internet Testbed Workshop, 2011.

[50] Trosky B. Callo Arias · Pieter van der Spek ·Paris Avgeriou, “A practice-driven systematic review of dependency analysis solutions”, Empir Software Eng (2011) 16:544–586.

[51] veth(4) - Linux manual page: http://man7.org/linux/man-

[52] W. Stalling, “Software-Defined Networks and OpenFlow,” The Internet Protocol Journal, Volume 16, No. 1, March 2013.

[53] Wang, L., & Jones, A. (2021). Automated Feedback in Higher Education: Benefits and Challenges. *Computers & Education*, 130, 76-89.

[54] Y. Jarraya, T. Madi, M. Debbabi, “A Survey and A Layered Taxonomy of Software-Defined Networking,” IEEE Communications Surveys & Tutorials, 16(4), April 2014.

[55] Yeganeh, S. H., Tootoonchian, A., & Ganjali, Y. (2013). "On scalability of software-defined networking." *IEEE Communications Magazine*, 51(2), 136-141.

[56] Yeganeh, S. H., Tootoonchian, A., & Ganjali, Y. (2013). "On scalability of software-defined networking." IEEE Communications Magazine, 51(2), 136-141.

[57] Yuichiro Tateiwa, Akifumi Asano, Yonghwan Kim, Yoshiaki Katayama, and Masaaki Niimura Nagoya Institute of Technology, Shinshu University, “Proposal of an event visualization system for debugging in software-defined networking exercises using Trema”.

[58] Yuichiro Tateiwa, Nagoya University, Japan, “A System for Generating Hints on Network Construction Exercises for Beginners”, ICCE 2016 IEEE.

[59] Hlwam Maint Htet, “DEPENDENCY ANALYSIS SYSTEM TO NARROW DOWN MISSETTINGS IN SDN CONSTRUCTION EXERCISES USING TREMA”, Indian Journal of Computer Science and Engineering, VOLUME 14 ISSUE 3 May-June 2024, (pp.358-363).

[60] Hlwam Maint Htet, Amy Tun, “TREMA BASED DEPENDENCY ANALYSIS SYSTEM FOR LEARNERS IN BUILDING SDN NETWORK

CONSTRUCTION EXERCISES", Indian Journal of Computer Science and Engineering, VOLUME 14 ISSUE 3 May-June 2024, (pp.364-369

List of Acronyms

SDN	Software Defined Networking
vHost	Virtual Host
vSwitch	Virtual Switch
FlowMod	Flow Modification
ONF	Open Networking Foundation
OF	Open Flow
OVS	Open Virtual Switch
RPC	Remote Procedure Call
QoS	Quality of Service
NFV	Network Function Virtualization
TCAM	Ternary Content-Addressable Memory
DP	Data Plane
CP	Control Plane
SI	Statement Instance
HI	Handler Instance
FE	Flow Entries
FI	Forwarding Instance
C2D	Control Plane to Data Plane
D2C	Data Plane to Control Plane
PC	Packet Chain
FlowMod	Flow Modification