A Dissertation submitted in fulfilment of the requirements for the degree of Doctor of Philosophy

New Information Model that Allows Logical Distribution of the Control Plane for Software-Defined Networking

The Distributed Active Information Model (DAIM) can enable an effective distributed control plane for SDN with OpenFlow as the standard protocol

Pakawat Pupatwibul

Autumn 2016

University of Technology Sydney
Faculty of Engineering and Information Technology
Centre for Real Time Information Networks

Supervisor
Professor Robin Braun

Co-supervisor
Dr. Bruce Moulton

Date of the graduation
May 2016

I dedicate this thesis to my lovely father, mother, sister, beloved wife and sons for their love and support.

CERTIFICATE OF ORIGINAL AUTHORSHIP

I certify that the work in this thesis has not previously been submitted for a degree nor has it been submitted as part of requirements for a degree except as fully acknowledged within the text.

I also certify that the thesis has been written by me. Any help that I have received in my research work and the preparation of the thesis itself has been acknowledged. In addition, I certify that all information sources and literature used are indicated in the thesis.

Signature of Student:

Date:

Abstract

In recent years, technological innovations in communication networks, computing applications and information modelling have been increasing significantly in complexity and functionality driven by the needs of the modern world. As large-scale networks are becoming more complex and difficult to manage, traditional network management paradigms struggle to cope with traffic bottlenecks of the traditional switch and routing based networking deployments. Recently, there has been a growing movement led by both industry and academia aiming to develop mechanisms to reach a management paradigm that separates the control plane from the data plane.

A new emerging network management paradigm called Software-Defined Networking (SDN) is an attempt to overcome the bottlenecks of traditional data networks. SDN offers a great potential to ease network management, and the OpenFlow protocol in particularly often referred to a radical new idea in networking. SDN adopts the concept of programmable networks which separate the control decisions from forwarding hardware and thus enabling the creation of a standardised programming interface. Flow computation is managed by a centralised controller with the switches only performing simple forwarding functions. This allows researchers to implement their protocols and algorithms to control data packets without impacting on the production network. Therefore, the emerging OpenFlow technology provides more flexible control of networks infrastructure, are cost effective, open and programmable components of network architecture.

SDN is very efficient at moving the computational load away from the forwarding plane and into a centralised controller, but a physically centralised controller can represent a single point of failure for the entire network. This centralisation approach brings optimality, however, it creates additional problems of its own including single-domain restriction, scalability, robustness and the ability for switches to adapt well to changes in local environments.

This research aims at developing a new distributed active information model (DAIM) to allow programmability of network elements and local decision-making processes that will essentially contribute to complex distributed networks. DAIM offers adaptation algorithms embedded with intelligent information objects to be applied to such complex systems. By applying the DAIM model and these adaptation algorithms, managing complex systems in any distributed network environment can become scalable, adaptable and robust. The DAIM model is integrated into the SDN architecture at the level of switches to provide a logically distributed control plane that can manage the flow setups. The proposal moves the computational load to the

switches, which allows them to adapt dynamically according to real-time demands and needs. The DAIM model can enhance information objects and network devices to make their local decisions through its active performance, and thus significantly reduce the workload of a centralised SDN/OpenFlow controller.

In addition to the introduction (Chapter 1) and the comprehensive literature reviews (Chapter 2), the first part of this dissertation (Chapter 3) presents the theoretical foundation for the rest of the dissertation. This foundation is comprised of the logically distributed control plane for SDN networks, an efficient DAIM model framework inspired by the O:MIB and *hybrid O:XML* semantics, as well as the necessary architecture to aggregate the distribution of network information. The details of the DAIM model including design, structure and packet forwarding process are also described.

The DAIM software specification and its implementation are demonstrated in the second part of the thesis (Chapter 4). The DAIM model is developed in the C++ programming language using free and open source NetBeans IDE. In more detail, the three core modules that construct the DAIM ecosystem are discussed with some sample code reviews and flowchart diagrams of the implemented algorithms. To show DAIM's feasibility, a small-size OpenFlow lab based on Raspberry Pi's has been set up physically to check the compliance of the system with its purpose and functions. Various tasks and scenarios are demonstrated to verify the functionalities of DAIM such as executing a ping command, streaming media and transferring files between hosts. These scenarios are created based on OpenVswitch in a virtualised network using *Mininet*.

The third part (Chapter 5) presents the performance evaluation of the DAIM model, which is defined by four characteristics: round-trip-time, throughput, latency and bandwidth. The ping command is used to measure the mean RTT between two IP hosts. The flow setup throughput and latency of the DAIM controller are measured by using *Cbench*. Also, *Iperf* is the tool used to measure the available bandwidth of the network. The performance of the distributed DAIM model has been tested and good results are reported when compared with current OpenFlow controllers including NOX, POX and NOX-MT. The comparisons reveal that DAIM can outperform both NOX and POX controllers. The DAIM's performance in a physical OpenFlow test lab and other parameters that can affect the performance evaluation are also discussed.

Because decentralisation is an essential element of autonomic systems, building a distributed computing environment by DAIM can consequently enable the development of autonomic management strategies. The experiment results show the DAIM model can be one of the architectural approaches to creating the autonomic service management for SDN. The DAIM model can be utilised to investigate the functionalities required by the autonomic networking within the ACNs community. This efficient DAIM model can be further applied to enable adaptability and autonomy to other distributed networks such as WSNs, P2P and Ad-Hoc sensor networks.

Contents

List of Figures

List of Tables

Nomenclature

AA	Autonomous Agent
ACNs	Autonomic Communication Networks
ACs	Autonomic Communications
ADSs	Autonomous Decentralised Systems
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
BGP	Border Gateway Protocol
CLI	Command Line Interface
DAI	Distributed Artificial Intelligent
DAIM	Distributed Active Information Model
DHCP	Dynamic Host Configuration Protocol
DMI	Desktop Management Interface
DMTF	Distributed Management Task Force
DNS	Domain Name System
FCAPS	fault, configuration, accounting, performance, security
FTP	File Transfer Protocol
IETF	Internet Engineering Task Force
ISO	International Organization for Standardization
LLDP	Link Layer Discovery Protocol
LTE	Long Term Evolution
MANET	Mobile Ad hoc Network

MEs	Managed Elements
MIB	Management Information Base
NETCONF	Network Configuration Protocol
NFV	Network Functions Virtualisation
NGN	Next Generation Network
NIB	Network Information Base
NOS	Network Operating System
OFLOPS	OpenFlow Operations Per Second
OSCA	Operating System Communication Application
OSPF	Open Shortest Path First
OSS	Operations Support System
OVSDB	OpenvSwitch Database Management Protocol
QoS	Quality of Service
RESTful	Representational State Transfer
RNC	Radio Network Controllers
SDN	Software-Defined Networking
SID	Shared Information and Data Model
SLAs	Service Level Agreements
SOAP	Simple Object Access Protocol
SSL	Secure Sockets Layer
STP	Spanning Tree Protocol
TLS	Transport Layer Security
TMF	TeleManagement Forum
UML	Unified Modelling Language
VETH	Virtual Ethernet

VMs	Virtual Machines
WASNs	Wireless Ad-hoc Sensor Networks
WiMAX	Worldwide Interoperability for Microwave Access
WSN	Wireless Sensor Network
XACML	eXtended Access Control Markup Language
XML	Extensible Markup Language
XML-RPC	XML-encode Remote Procedure Call

Acknowledgements

I would like to acknowledge all the support and encouragement received during my PhD research. Firstly, I would like to express my deep gratitude to Professor Robin Braun, who has been my supervisor and very good friend. His valuable guidance through this research was a great source of support and encouragement and always made me go that extra mile to solve the various problems that lead to this work. I cherished the opportunity to watch and learn from his knowledge and experience. His frequent insights and patience with me are always appreciated.

I also thank my co-supervisor, Dr. Bruce Moulton, for supporting me throughout this work. During the course of this research, I also benefited greatly from interactions and technical discussions with other talented and warm-hearted members from the CRIN centre, including Dr. Zenon Chaczko and Dr. Abdallah Al Sabbagh, to whom I wish to give sincere thanks. The technical conversations with them also helped me over the course of this project not only in terms of resolving quick technical difficulties but also with regards to lightening up.

It has been my privilege to work closely with Ameen Banjar, my research collaborator and best friend, I am gratefully thanking him for his invaluable contributions and innovative ideas towards this project. My special mention also goes to Md. Imam Hossain, for helping me with the solution to the problems in C/C++ programming, and setting up the test bed used for this research. I am very proud of what we have achieved together, thank you both.

I would like to thank Suan Dusit University, Bangkok, Thailand for providing international student academic research scholarship, and Emeritus Professor Tony Moon for nominating me for UTS International Research Scholarship (IRS) to support my PhD degree financially.

I am truly indebted to all my friends who have supported me over the last few years: Wael Alenazy, Jiajia Shi, Raniyah Wazirali, Lucia Gordon, Anup Kale, Denise Umuhoza, Shaher Slehat and Sanya Khruahong for their help in various ways. I have enjoyed many useful and happy chats with them. I have been very fortunate to have them around during my PhD study.

Last but by no means the least, I wish to give special thanks to my lovely family, Dr. Sawarng, Asst. Prof. Dr. Kanungnit and my sister Pitinut, for their immense support and all of the sacrifices that they have made on my behalf. My parents always gave me constant support and tried to provide me with the best education they can afford. They have been an important driving force to encourage me behind this PhD research.

I would like to express my sincere appreciation to my beloved wife, Duangporn, for her endless support, dedicated love, patience and understanding in every possible way. She always supported me in the moments when there was no one to answer my queries. Finally, I would like to thank my sons, Neptune and Neymar, for the happiness and cheers they have given me as their own way to support me.

Related Publications

The technical contributions and discussions in the thesis are mainly based on the following publications written by the author in which five of them are co-authored:

A. International Journal Publications:

[J1] Pupatwibul, Pakawat, Ameen Banjar, Abdallah AL Sabbagh, and Robin Braun. "A Comparative Review: Accurate OpenFlow Simulation Tools for Prototyping." Journal of Networks 10, no. 5 (2015): 322-327.

[J2] Banjar Ameen, Pakawat Pupatwibul, and Robin Braun. "DAIM: A Mechanism to Distribute Control Functions Within OpenFlow Switches." Journal of Networks 9.1 (2014): 1-9.

[J3] Banjar Ameen, Pakawat Pupatwibul, Abdallah AL Sabbagh, and Robin Braun. "Using an ICN Approach to Support Multiple Controllers in OpenFlow." International Journal of Electrical & Computer Sciences 14, no. 2 (2014).

B. International Conference Publications:

[C1] Pupatwibul, Pakawat, Ameen Banjar, and Robin Braun. "Using DAIM as a Reactive Interpreter for OpenFlow Networks to Enable Autonomic Functionality." ACM SIGCOMM Computer Communication Review. Vol. 43. No. 4. ACM, 2013.

[C2] Pupatwibul, P., Jozi, B. and Braun, R. 2011, "Investigating O:MIB-Based Distributed Active Information Model (DAIM) for Autonomics", International Conference on Information and Communication Technologies and Applications IIIS, Orlando, Florida, USA, pp. 7-12.

[C3] Pupatwibul, P., Sabbagh, A.A.L., Banjar, A. & Braun, R. 2012, "Distributed Systems in Next Generation Networks", 1st Australian Conference on the Applications of Systems Engineering ACASE'12, p. 32.

[C4] Pupatwibul, P., Banjar, A., Al Sabbagh, A., & Braun, R. (2013, October). "Developing an Application Based on OpenFlow to Enhance Mobile IP Networks". In Local Computer Networks Workshops (LCN Workshops), 2013 IEEE 38th Conference on (pp. 936-940). IEEE.

[C5] Al Sabbagh, A., Pupatwibul, P., Banjar, A., & Braun, R. (2013, October). "Optimization of the OpenFlow Controller in Wireless Environments for Enhancing Mobility". In Local Computer Networks Workshops (LCN Workshops), 2013 IEEE 38th Conference on (pp. 930-935). IEEE

[C6] Banjar, A., Pupatwibul, P., Braun, R., & Moulton, B. (2014, February). "Analysing the Performance of the OpenFlow Standard for Software-Defined Networking Using the OMNeT++ Network Simulator". In Computer Aided System Engineering (APCASE), 2014 Asia-Pacific Conference on (pp. 31-37). IEEE.

[C7] Pupatwibul, P., Banjar, A. & Braun, R. 2014, "Performance Evaluation of TCP/IP vs. OpenFlow in INET Framework Using OMNeT++, and Implementation of Intelligent Computational Model to Provide Autonomous Behaviour." The Asian Conference on Technology, Information & Society 2014, The International Academic Forum (IAFOR) Osaka, Japan, pp. 43-56.

C. International Book Chapter Publications:

[B1] Pupatwibul, Pakawat, Ameen Banjar, Abdallah AL Sabbagh, and Robin Braun. "An Intelligent Model for Distributed Systems in Next Generation Networks." In Advanced Methods and Applications in Computational Intelligence, pp. 315-334. Springer International Publishing, 2014.

[B2] Banjar, Ameen, Pakawat Pupatwibul, and Robin Braun. "Comparison of TCP/IP Routing Versus OpenFlow Table and Implementation of Intelligent Computational Model to Provide Autonomous Behaviour." Computational Intelligence and Efficiency in Engineering Systems. Springer International Publishing, 2015. 121-142.

Part I.

Elaborating on the “Propositions”

1. Introduction

1.1. Introduction

In this chapter, the goal of this research on developing a new information model named: Distributed Active Information Model (DAIM) as a logically distributed control plane for OpenFlow networks and its impact on the Software-Defined Networking (SDN) community are described. Furthermore, the motivations of this research are explained. The promising research questions are defined on the basis of understanding and identifying a gap in the scholarly literature. The overall thesis structure is also presented. The statements of technical contributions are demonstrated.

1.2. Background of Network Management Complexity

Data communication networks were first invented in the early 1900's; back then most of the computing power was from human brains. The only way networks could be managed was from centralised manual exchanges and to use common management and data paths. When the first evolution of automatic exchanges arose, they were actually just mimicking the manual exchanges. Also, networks gradually evolved towards the separation of the management plane and the data plane. All the allocation of system resources was made before a communication could be established. Network operators did not truly understand the separation of work required to make a connection, and hence had very little need for the "layered" networking structure.

In the 1970's, there was a rapid increase in computing power (Moore's Law) [91]. This power has made it possible to shift the computing needs of the network away from the centre to the end points. Moreover, this power also made it possible for the network to divide data into "chunks", which are called packets. Packets get delivered based on the meta-data that they carry with them such as source and destination IP addresses. Therefore, it is feasible to add meta-data to these "chunks" of data, so that they can become packets and can be delivered by the network elements themselves. When the network delivers packets, first it has to deliver packets to the local post office, and then the postal service delivers them globally, and then locally

to the destination. This packet delivering process can be clearly seen as a division of labour, and hence the 7/5 layer model.

All of this proliferation of devices and division of labour gives rise to the allocation of management to the network elements, and thus to the massive rise in complexity of telecommunication networks. A recent count suggests that there have been upwards of 15 Billion connected devices with at least 1 CPU. The networks have become complex in the classic sense of the word. Networks are stochastic, non-linear, chaotic, and they are intractable. Not only do network operators have to consider the above, but also this distributed nature forces them to combine the management and data forwarding paths, leading to further management complexity and vulnerability.

These issues will be the real inhibitor to network growth in the future. Complexity is the barrier, not electronics or optics. There is a need to re-evaluate the basic management model so as to reduce complexity. Some of the research trends have been proposed to tackle network complexity for example by separating the data and management planes, and by swapping network parameters for network requirements. If network control can separate the data and management planes, then it only has two areas of work (the bosses and the workers). All data forwarding then practically becomes a layer 2 function. The functionalities of layers 3 and 4 are subsumed into the management function.

Complex electronic environment refers to a group of electronic devices connected (wired or wireless networks) to share information and resources. Any networked system needs a management protocol that performs different kinds of tasks such as operation (monitoring the performance of the network and detecting any occurred problem as soon as possible), and maintenance (fix any occurred problem and always maintain network infrastructure to ensure the network is configured and operating at optimal performance).

Most of the proposed network management protocols are based on the International Organisation for Standardisation (ISO) definition for management model. Performance management, Configuration management, Accounting management, Fault management and Security management are five characteristics of this definition [2]. Performance management refers to the operation task with the aid of collecting and processing important management information. Configuration management refers to monitoring and controlling the effects of any device on network performance. This characteristic can be used to search for any useful information when any problem occurs. Accounting management guarantees a fair usage of network resources by any user. Fault management refers to the maintenance task that can detect the problem, isolate and fix it in the network. Security management controls the access level of all users to the resources of the network based on specific policies [40].

1.3. Research Motivations

1.3.1. Motivation from Management of Distributed Complex Networks

Firstly, the research is inspired by the consideration of many new characteristics of complex distributed networks, which are expected to support multiple IP-based services. Furthermore, the current static network architecture is not sufficient to cope with the dynamic computing and storage needs of today's enterprise and carrier environments. The existing networks are growing rapidly in size and complexity, thus may generate significant challenges in network management, operations and maintenance. The aim is to study the growth of large-scale technological renovations in telecommunication core and access networks that would be deployed in the next 5 to 10 years. For example, these networks include mobile communication, peer-to-peer networks, pervasive/nomadic computing, and wireless ad-hoc sensor networks (WASNs). They have dramatically influenced our daily life in the past few years. It is expected that high-end devices such as routers and switches will become increasingly programmable, and become able to execute more software control directly on the devices. In the conventional configuration, management approaches are often inefficient and involve too many human efforts. The research outcome can be readily applied to manage these distributed networking systems. This thesis analyses the relationship among adaptation algorithms, managed elements, and distributed computing systems, and applies distributed management strategies to achieve an effective distributed control plane for SDN.

1.3.2. Motivation from Self-Management Strategies

Secondly, this research is motivated by the emergence of autonomic systems and the concepts of self-x properties. Because current networks are evolving rapidly in size and complexity, this may bring important challenges in terms of administration, operation, and maintenance of networks and services. Moreover, traditional network management typically needs to reconfigure most of the devices if any circumstances or requirements change. Therefore, the autonomic configuration capabilities should be implemented to manage the networks. Also, dynamic configuration and topology will require quick response and network wide configuration capability in order to manage large-scale networks. Self-management in autonomic communication is considered as one of the promising solutions to cope with such complex distributed environments. It gives systems the capability to adapt to ever-changing environments in order to improve its scalability, survivability of services, and robustness, as well as reducing the need for human operations from low-level reconfiguration work. With autonomic functionality, future networks could resist any attacks and positional damage, under the ability to reconfigure themselves. It is important to

research the proposed self-x paradigm and other available mechanisms as they are essentially required in achieving high-level network autonomy and recognised as being extremely related to Autonomic Communication Networks (ACNs) as well as other real networks. The implementation of the DAIM model will enable the development of autonomic communication with some level of distributed features to overcome the scalability issues in the SDN architecture.

1.3.3. Motivation from OpenFlow-Based SDN

Currently, network management is changing towards a more flexible network management. Open Network Foundation (ONF) has been dedicated to the promotion and adoption of Software-Defined Networking (SDN) that emerged in recent years. SDN is an emerging computer networking architecture in which the control plane is separated from the physical network and implemented in a software application. This setup allows network administrators to have programmable central control of network traffic without requiring physical access to routers and switches. The separation of the control plane from data forwarding plane offers the promise of new paradigms in the management of networks, including the development of autonomic behaviours.

By increasing network resource utilisation and decreasing operational costs, networks can be more flexible and cost effective with the deployment of SDN. This research is motivated by the development of OpenFlow network, which enables researchers to easily run their experiments and deploy innovative routing and switching protocols in campus networks we use every day. OpenFlow is an open standards-based protocol that can be used for services and applications such as high-security networks, virtual machine mobility, and next generation distributed IP-based networks. However, OpenFlow system is implemented with a centralised control plane, and thus is struggling to cope with key limitations concerning performance, lack of scalability, reliability, and large establishment delay issues. Therefore, OpenFlow will be a very useful test bed networking system for applying and extending the DAIM model to enable autonomic functions. By adopting this DAIM structure, it is possible to improve the performance of a centralised OpenFlow controller.

1.4. Research Objectives and Scope

This research aims at applying the DAIM principles to the SDN structure to provide a logically distributed control plane for OpenFlow networks. The notion of distributed communication networks is proposed as a promising solution to accommodate the increased complexity of enterprise-grade networks. The key is to design a comprehensive distributed system in which the computational load will be distributed to a lower level at the switches, allowing them to manage local information of network entities. This study will provide the methodologies, structures and algorithms

working to support self-adaptation strategies. Once being developed and tested by experiment in this dissertation, the research outcomes will create an environment that enables autonomic functionality to be introduced, and can also be applicable for other real networks such as Wireless Sensor Network (WSN), mobile computing, and bush fire alarm system.

1.4.1. Research Objectives

1. The primary goal of this research is to develop a new network management base structure, the DAIM model, to be integrated into the SDN architecture at the level of the switches. This will enhance switches to manage flows locally by using distribution controllers that operate on packets instead of a centralised controller in OpenFlow environment in order to improve its performance, scalability and robustness.

2. To produce the methodologies, models and algorithmic protocols that will help enable a desirable distribution paradigm to improve its survivability for the complexity in future telecommunication networks. The DAIM model can be developed with more dynamic and consistent configuration of policies, fewer errors and reduced operational expenses. After being designed and implemented in this thesis, they can migrate to other real networks such as WSNs, pervasive communication networks, and mobile communications.

3. To investigate direct network development efforts that will enable and support advanced technology development such as distributed learning and self-adaptation strategies, quality of service (QoS), and autonomic management strategies to maintain the system in case of changing requirements and unexpected threats. This will also open the possibility for future thesis research by other students at the University of Technology Sydney (UTS) with relevant research field.

The expected outcome of this research is a successful integration of the DAIM model in an OpenFlow network, and in so doing enable it to perform operational functions. This will enable the richness of innovative adaptation algorithms to be applied to any distributed computing environment. The DAIM model is developed as a physically distributed control plane to organise massive network elements. DAIM has a local processing engine that instructs the switch to forward flows without the vetting from a centralised controller. DAIM is implemented based on the SDN paradigm with some level of distributed functions that can simplify network management. The practical feasibility of DAIM is demonstrated through experimental evaluations conducted with the Mininet emulator.

1.4.2. Research Scope

The scope of this research covers the following areas and can be described as follows:

- Proposing a new distributed active information model (DAIM) theory

One possible implementation of DAIM introduced in this thesis is the creation of a DAIM cloud embedded with smart agents implemented in a Java virtual machine environment (e.g. JadeX). These agents will be bounded to a particular variable or a value in the routing table, and have some level of distributed intelligence and self-adaptation strategy to manage those variables if any changes occur in the network environment.

In the case of OpenFlow networks, the concept of managing the variables is having each one of the entries in the flow table or each OpenFlow message "owned" by a unique DAIM agent. In addition, an OpenFlow-enabled switch has flow tables built into it and a Secure Sockets Layer (SSL) connection to the controller to manage these flow tables and install the flow entries. The DAIM theory proposes to convert the controller into a system requirement database and have some level of distributed intelligence environment (DAIM agents) that resides in all the switches. These smart agents should then be able to identify what values to compute in the flow tables based on the system requirements, which are driven by the business needs.

- Implement and re-engineer the OpenFlow-based SDN structure by integrating the DAIM model at the level of the switches to create a physically distributed control platform. This may also include a portion of the network operating system within switches to support some level of network control. Moreover, the thesis scope covers the deployment of a distributed flow management architecture in which each DAIM controller manages its connected switch in order to collect network information (*e.g.*, links, hosts, and ports) using the OpenFlow protocol and communicating messages (get, get-response, and set). This information will be stored in the system's local memory, which is used for managing the network. After deploying the DAIM model, an OpenFlow-based SDN network can perform operational functions in terms of forwarding packets and installing appropriate flows along the chosen path.

- Decentralisation is an essential element of autonomic systems: any centralised resource such as an SDN controller or control point will act as a brake on a system's ability to adapt, especially in terms of robustness and performance. New techniques are urgently needed to understand the exact robustness, performance, and complexity characteristics of decentralised algorithms. Each DAIM controller will be physically distributed across OpenFlow switches with the ability to organise local information of distributed network elements for management purposes. By adopting the DAIM model, OpenFlow switches will have the capability to manage flows locally by using DAIM to operate on packets instead of a centralised controller in an OpenFlow environment.

Note that it is outside the scope of this research to develop the autonomic strategies themselves.

1.5. Problem Statement

The increasing demand and growth of network traffic over the past few years has led to the bottlenecks of the traditional networking paradigm. Moreover, today's networking environment typically contains a large number of communication devices such as router, switches and hosts with many complex protocols that are embedded and implemented on them. Network operators are responsible to ensure best connectivity and configure policies to accommodate a wide range of network events and application services. They usually transform these high-level policies into lower level of configuration commands manually. This is a complex task that often is achieved with very limited access to tools. As a result, network control and management including other activities related to provisioning of networked systems are quite challenging and error-prone tasks.

Furthermore, current networks have many restrictions including difficulties to meet the business and technical requirements over the past few decades, while the industry has developed protocols for the networks to provide an optimal performance, reliability and greater connectivity, and security is more stringent. Moreover, there are difficulties to add, modify and remove any device or configure these devices, which must be touched by an IT person that needs to configure many switches, routers and firewalls using the device-level management tools [44].

The explosive deployment of Internet ossification is another challenge for network operators and researchers. Because of huge Internet adoption and its effects on many aspects in our life, it is difficult to develop the Internet in regards to physical infrastructure along with the protocols and performance. The demands of new emerging applications are becoming more complex, and thus the existing state of the Internet seems unable to evolve to overcome these emerging challenges.

One of the promising research directions in data communication is stimulated by Software Defined Networking that emerged in recent years. The technology deploys the concept of programmable networks as a way to facilitate network evolution, which endeavour to resolve the bottlenecks of traditional switching and routing based network deployments. By adopting SDN, network operators can gain network programmability, network control and automation, which will enable them to create highly scalable and flexible networks that can adapt to changing business needs.

Although SDN is able to provide flexibility in network management and promises to dramatically simplify network control to enable innovation and evolution, SDN particularly opens a set of currently unresolved key challenges concerning reliability, service orchestration, scalability and performance. How to adopt and operate SDN in a reliable manner by providing redundancy and load balancing? Can SDN provide performance figures required for network operations and where should SDN be placed to optimise the network subject to different design criteria? How can SDN applications be orchestrated? How can network monitoring be adapted to such flexible networks? This research aims at addressing some of these and similar questions

in OpenFlow-based SDN focusing specifically on performance and scalability.

1.5.1. Research Questions

The ultimate goal of this research is to develop the DAIM model that allows logical distribution of the control plane for SDN with OpenFlow as the standardised protocol. Using DAIM, systems will react according to predefined rules and create their own adaptation strategies driven by system objectives to meet the operators' Service-Level Agreements (SLAs) and able to satisfy users. The implemented DAIM model will hopefully address current OpenFlow problems as well as the research challenges in managing future distributed network systems aiming at an autonomic management strategy.

There is an urgent need to explore the distributed autonomic ways in managing network systems to address the proposed issues above. The main research question can be further expanded into sub-questions as follows:

1. Can we develop a new distributed active information model (DAIM), and integrate into the OpenFlow architecture at the level of the switches to provide a logically distributed control plane?

2. Can the OpenFlow switches manage flows locally by using DAIM to operate on packets instead of a centralised controller in an SDN environment?

3. Can the performance of DAIM model be proven to provide efficiency, robustness, resilience, flexibility and improve the flow setup throughput and latency of an OpenFlow controller?

4. Can the DAIM model provide a distributed computing environment that will create the appropriate "hooks" and allow the development of autonomic management strategies?

1.5.2. Propositions Derived from the Research Questions

This dissertation describes the author's research over the last years in developing the DAIM model as a logically distributed control plane with the hope to address the limitations of current OpenFlow deployments with respect to scalability and a single point of failure. Moreover, the expected research outcomes will also contribute to the SDN and the networking research community.

To the best of the author's knowledge, this work proposes one of the most recent efforts to develop a distributed active information model and enable distributed computing strategies in an OpenFlow network. The aim of this research is to build up a scalable, robust and efficient distributed control plane for SDN.

Given that we have an OpenFlow environment as a standardisation of Software Defined Networking, this research proposes that:

Proposition 1: The DAIM model will be able to integrate seamlessly into the OpenFlow architecture at the level of the switches to provide a logically distributed control plane.

Proposition 2: This logically distributed control plane will allow the switches to manage flows locally by having DAIM operate on packets instead of a centralised controller in an SDN environment.

Proposition 3: The performance of the DAIM model can be proved to possess efficiency, robustness/resilience, flexibility, and will improve the flow setup throughput and latency of an OpenFlow controller.

Proposition 4: Most importantly, in building the distributed computing environment offered by DAIM, this can create the appropriate "hooks" that will allow the development of autonomic management strategies.

To facilitate the propositions above, the DAIM controllers are physically distributed where each controller manages its connected switch and distributes useful information to other instances within the cluster and communicates if necessary with the neighbouring domain.

1.6. Approach and Methodology

The section describes the methods for analysing problems and identifying gaps, followed by proposing a potential solution to resolve the problems, implementing the designed model, conducting experiments and verifying the outcomes. Finally, results evaluation are discussed to demonstrate that the implemented DAIM model is operating functionally. This thesis has presented a list of activities and steps to obtain outcomes that will give evidence as well as test (prove/disprove) the propositions, which are described in the following sections.

1.6.1. Reviewing the Scholarly Literature

The literature review focuses on identifying the problems of OpenFlow in regards to scalability and performance, and analyses them to obtain research gaps. The goal of the literature study is to evaluate alternative approaches and to propose the most feasible and desirable systems for development. Moreover, it aims to review several theoretical and practical aspects of previous research. For this purpose, various network management protocols and other implementations that were proposed to improve the performance of OpenFlow have been revised in detail. Many related works in the past few years have been summarised and evaluated in order to generate

new research hypothesis as well as showing similarities and differences (consistencies and inconsistencies) in previous research.

The study is conducted mainly in the field of distributed systems and its functionalities, which include what, why, where, and how they can be implemented, and what are the potential development environments (*e.g.*, SDN/OpenFlow). This review also surveys the state of the art in programmable networks, which focuses particularly on OpenFlow and discusses other perspectives from earlier ideas to recent developments. Then current alternatives of OpenFlow for implementation, testing of SDN-based applications, services and protocols are examined. The literature search used standard sources and tools such as UTS Library databases, online resource papers and other related dissertation. Thus, the method to form this literature review is identifying the candidate system requirements, which requires the interpretation of network services that can improve the scalability issues in the current OpenFlow network. These reviews have been analysed to specify the gaps, which motivates this research approach.

1.6.2. Designing a Candidate System

After the research gaps are determined, the next phase to facilitate the development of DAIM is system design. Designing a candidate system is a process to solve the issues and consider as a plan for research software solution. System design may be regarded as the most creative and challenging phase of the research methodology due to the fact that it is critical and very important, which involves technical activities such as defining the required system output, specifying data required to produce the output, documenting all aspects of design, and so on. The main purpose of the design phase is to create a DAIM structure that is suitable for implementation in some programming language. There are two main activities in the design phase. Firstly, a high-level architectural design is conducted to decompose the system into modules, and representing the interfaces and the invocation relationships among the modules. Secondly, a low-level design is undertaken to design the internal structure of individual modules in greater detail. In addition, the algorithms and data structure of the modules are designed and documented.

It is important to have a well-defined software requirement specification for the designed DAIM model, which is a description of the system laying out the functions that DAIM must perform. The aim is to distribute the DAIM controllers and have them running on every OpenFlow switch. These controllers can also actively share network information between each other to manage the network and process flows locally according to the information collected. For example, the task where a switch encapsulates an incoming packet and forwards it to the controller, instead the switch should be able to pass the packet to a local processing engine, which could be addressed by a distributed processing system like DAIM. In order achieve this, flow table entries need to be observed and a set of rules developed on how the

DAIM model will process the flows. Once a switch has received an unknown packet, the action does not require forwarding to the controller for high-level decisions. This should be the case where DAIM is actually accessing its own information database (local memory) and checking for the designated destination, and making their local decisions to install the necessary flow entries in the flow table.

In this phase, the functionalities of the DAIM and the communication protocol between each DAIM controller need to be clearly defined in order to develop a fully distributed network environment. For example, the communication protocol is to specify the network structure in terms of network topology, traffic type, and how network data are shared among each DAIM controller. Two main elements of the shared information are (1) describing the infrastructures that are available, and (2) describing the requirements and services that the infrastructure has to provide. Once the DAIM model has been designed, it is ready for implementation.

1.6.3. Emulating the Candidate System Using Mininet

Specifying a benchmark network and building up scenario topologies is done in Mininet (virtual machine environment). Mininet is a Linux-based networking environment giving the ability to handle and test IP packets, FTP transfer, Ping, and so on. Mininet uses Linux processes in network namespaces instead of running multiple VMs to be more scalable. It allows you to quickly create an emulated OpenFlow network and provides a smooth path to running on hardware. The codes that are developed and tested on Mininet for an OpenFlow controller, modified switch or host, can move to a real system with no changes for real-world testing, performance evaluation, and deployment.

After setting up the Mininet environment, the proposed DAIM model can be tested by having DAIM controllers manipulate the flow entries in a flow table. In addition, the environment setup will use dedicated OpenFlow-enabled switches. This way the DAIM controller can manage the flow entries according to the flow modification messages computed from the collected network information. To start experimenting with OpenFlow, we will need both an OpenFlow switch (either a software-based switch or a hardware one) and a controller that supports higher-level applications.

According to Lantz (2010), Mininet is a network emulation platform that supports research, development, prototyping, testing, debugging, and other tasks related to having a complex experimental network on a single device. Mininet gives a simple and inexpensive network test bed for implementing OpenFlow applications. It uses OS-level virtualisation features, including Linux processes, virtual Ethernet pairs, and network namespaces to create a virtual OpenFlow network. In virtual machine environment, Mininet can provide an easy way to quickly create, customise, and share a software defined network prototype with gigabits of bandwidth up to hundreds of nodes (*e.g.* controller, switches and hosts). Lantz further states that users can develop a new network architecture, run it on large topologies with application

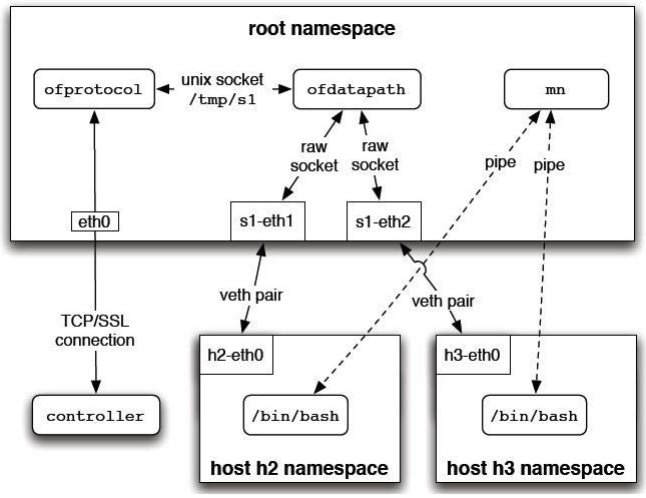


Figure 1.1.: The Connections and Components of Two-Host Network Created by Mininet[57]

traffic, and then apply the exact same codes and scripts into the real production network. Figure 1.1 illustrates a virtual network created by Mininet, placing a two-host process in network namespaces and connecting them with Ethernet pairs (veth). It is an example of a connection to a user-space OpenFlow switch.

By combining both extensible CLI and API with lightweight virtualisation, Mininet emulates hosts, links, switches, and controllers of the OpenFlow network.

Hosts – the network namespace that contains network state. It provides processes of interfaces, ports, and routing tables. Every host has its own virtual Ethernet interface(s) and a pipe to a parent Mininet process in order to send monitor outputs and commands.

Links – A virtual Ethernet pair, which acts as a wire connection of two virtual interfaces. Each interface is functioned as a full Ethernet port; sending and receiving packets from all system and application software. Is may be attached to virtual switches (*e.g.*, Linux bridge, software OpenFlow switch).

Switches – software-based OpenFlow switches, which perform the same packet delivery semantics as a hardware switch. They are available in both kernel-space and user-space switches.

Controllers – Controllers can be anywhere in the real or simulated network as long as the running switch has IP-level connectivity to the controller. As Mininet operates in VMs, the controller could run in the VMs, natively on the host, or in the cloud.

Figure 1.2 shows the console application using Mininet API to interact with and monitor hosts, switches, controllers and MiniEdit is a simple graphical network editor to open up CLI for nodes.

Figure 1.2.: Mininet Emulation Software

1.6.4. Implementing the Candidate System

This phase is the practical implementation of the candidate DAIM system. In this phase, the main goal is to implement the model in the best possible manner. For the implementation of DAIM, the *openflow.h* header file is used to model the OpenFlow protocol and its defined messages. DAIM is developed based on the theory and the designed model presented in Chapter 3 as closely as possible. Firstly, it is important to specify the new system's components and structure in order for it to be successfully implemented. Also, the functionality of the NOX controller needs to be identified. A thorough study of the NOX controller, OpenFlow protocol, and flow table contents and rules are required to develop the DAIM model efficiently. This new system needs to mimic NOX functionalities and have the ability to process incoming packets locally, directing them from source to destination without forwarding to a centralised controller for routing decisions. This research is done following the NOX controller implementation. This choice is based on the fact that the NOX controller is fully open source and the modules which dictate the behaviour of the controller can be programmed in C++. Moreover, NOX has few dependencies and does not contain any complex networking functions such as routing, host detection, or topology reconstruction. It also comes with some useful example network applications.

The programming language used to develop DAIM is C++. In addition, a free and open source NetBeans IDE is used to develop DAIM, which provides a great set of tools for C/C++ applications. The DAIM model is implemented in three phases including a basic carrier, semi-distributed and fully distributed functionality. The main purpose of this work is to develop a logically distributed control plane by integrating the DAIM model into the SDN structure at the level of the switches. This will enable the switches to process events locally by distributing DAIM to operate flows instead of a centralised controller in an SDN environment. Furthermore, the three core modules that construct the DAIM ecosystem are described on the basis of a software specification. The implemented modules are comprised of the Communication module, the Local Storage module, and the Controller module as well as

the most significant messages that are needed for the communication between the switch and the DAIM controller. More details are presented in Chapter 4.

1.6.5. Validating the Candidate System

After developing the DAIM model, a set of actions have been conducted to check the compliance of the overall system. These actions include testing and verifying that DAIM is operating functionally and meets requirements and specifications which fulfil its intended purpose. The method for testing DAIM is to generate a ping command and measure the mean RTT in different network scenarios. The scenario is created in a virtualised network using Mininet. For performance evaluation, Cbench [96] tool is used to determine controller performance in terms of throughput and latency, whereas Iperf [50] is performed to measure the maximum TCP and UDP bandwidth utilisation of the network. Moreover, a small-size OpenFlow network based on Raspberry Pi's has been set up physically to demonstrate that DAIM can be deployed in real networks.

Next step is collecting the experiment results for performance evaluation, which will show that the DAIM controller produces comparable results to the current SDN/OpenFlow controllers and that it is possible to run DAIM on productive networks. The performance of the DAIM model is compared with NOX, POX and NOX-MT controllers and good results are reported regarding throughput and latency. Validation is to ensure the DAIM system is implemented to deliver all functionalities as defined in the requirements. There are important distinctions between verification and validation. Verification evaluates documents, code, requirements and specifications to determine whether they meet the specified requirements for that phase. In contrast, validation evaluates the candidate system at the end of the development process to determine whether it satisfies specified business requirements. The inputs of verification are illustrated as checklists, issues lists and reviews. The input of validation is the actual testing of the candidate DAIM system.

1.6.6. Concluding Observations

The purpose of a conclusion is to demonstrate the significance of the thesis and the implications of the key research findings. The main goal of this phase is to summarise the results that have been achieved by comparing DAIM performance with other OpenFlow controller implementations. The conclusions are presented and validated by referring back to the research propositions. The results will demonstrate that an effectively distributed network for business needs can be implemented by applying DAIM to the SDN architecture. After deploying DAIM in OpenFlow networks, it is also possible to introduce the self-x functionalities that will allow the development of autonomic management strategies. The performance evaluation shows the proposed

DAIM model is operational, flexible, scalable and adaptable for any complex electronic environment. Furthermore, the limitations and weaknesses of this research as well as how these factors can be improved in future research are discussed in the conclusions. The discussion ends with some suggestions for possible follow-up study in response to this research.

Ultimately, the proposed DAIM structure will hopefully address the limitations of current approaches in the novel and future distributed network systems aiming at an autonomic management strategy.

1.7. Outline of the Thesis

This thesis consists of six chapters and is organised as follows:

Chapter 2: Literature Review

Chapter 2 gives a comprehensive review of related literature sorted in terms of different main aspects and the current state of art in software defined networking. In addition, it provides an overview of the standard sets of the information model, issues of managing complex telecommunication networks, and additional problems of SDN including single-domain restriction, scalability, and the ability for switches to act autonomously. Relevant previous approaches to resolving the scalability issues are described, and their possible connections with this study are also analysed.

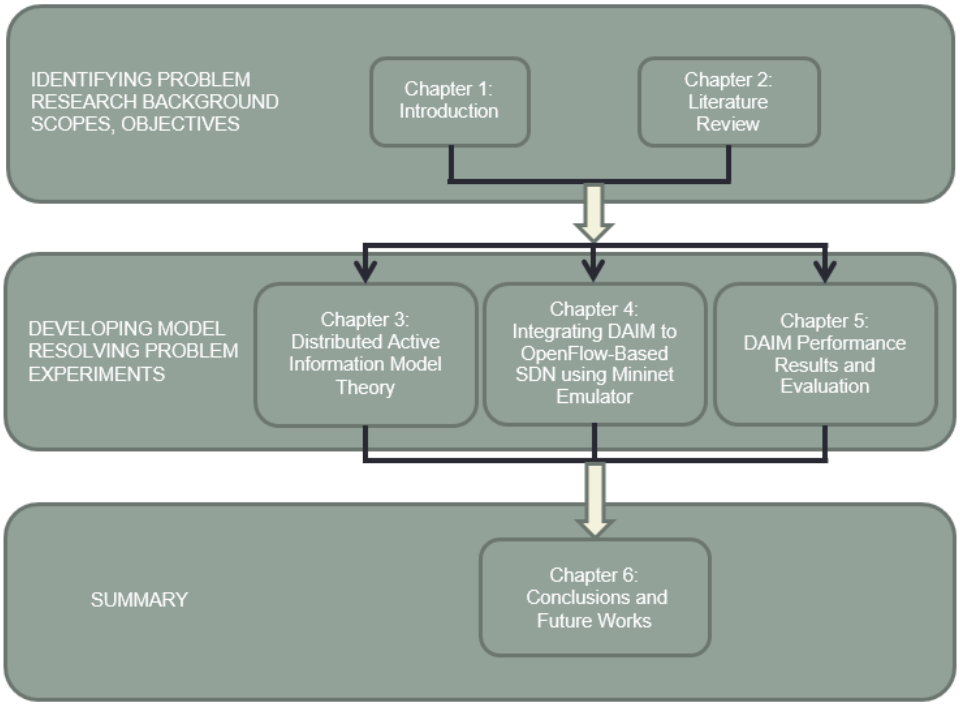


Figure 1.3.: Thesis Structure

Chapter 3: Distributed Active Information Model Theory

The theoretical background based on the O:MIB structure and O:XML technique are presented in this chapter. Details of the proposed DAIM model are described and its practical feasibility is also discussed. In addition, the fundamental principles of the DAIM model such as design, architecture and agent implementation have been introduced to the SDN paradigm. The difference between conventional networks and SDN have been analysed in the context of implementation and deployment. The reason why the DAIM model is important and required is further explained, the contribution of DAIM therefore is justified. The DAIM model is expected to satisfy the requirements of distributed network functionalities. This chapter also discussed the process of packet forwarding within DAIM as well as the risk scenarios of the DAIM model.

Chapter 4: Integrating DAIM to OpenFlow-Based SDN using Mininet Emulator

DAIM model implementation have been described in three phases including the basic carrier, semi-distributed and fully distributed functionality. The details of the overall system are further discussed from a software specification perspective. In more details, the three core modules that construct the DAIM ecosystem are discussed with some sample code reviews and flowchart diagrams of the implemented algorithms. The developed modules consist of the communication module, the local storage module and the controller module. The most significant OpenFlow messages required for the communication between the switch and the DAIM controller are also explained. To validate DAIM, a set of actions is used to check the compliance of the system with its purpose and functions. In addition, various tasks and scenarios are demonstrated to test the functionality of the DAIM controller such as executing a ping command, streaming media and transferring files between hosts. The scenarios are created based on OpenVswitch in a virtualised network using Mininet emulator.

Chapter 5: DAIM Performance Results and Evaluation

The technical details and testing methodology of experimental evaluation are presented in the first place. The network topology and scenarios used for the performance evaluation are further explained. Performance evaluation of the DAIM controller is defined by four characteristics: round-trip-time, throughput, latency and bandwidth. In order to quantify DAIM controller performance, the ping command is executed to measure the average RTT values. Moreover, the Cbench tool is used to measure the number of flow setups per second that the DAIM controller can handle in terms of throughput and latency. Also, Iperf is served as a benchmarking tool for measuring the network's maximum TCP and UDP bandwidth utilisation. The performance of the distributed DAIM controller has been tested and good results are reported when compared with current OpenFlow controllers including NOX, POX and NOX-MT. Other parameters that can affect the performance evaluation are also described in this chapter.

Chapter 6: Research Findings, Conclusions and Future Works

Lastly, on the basis of the research propositions presented in Chapter 1, the proposed model and its implementation as well as results of analyses in Chapter 3, 4 and 5, the major research findings are discussed in Chapter 6. This chapter concludes with some suggestions for possible future research and developments. Chapter 6 summarises the contribution (achievements and impact) to knowledge, with results, developed models and approaches to other related networking research areas such as wireless, mobile, and sensor networks, or infrastructures that provide global services like the National Broadband Network (NBN). Figure 1.3 depicts a block diagram of the thesis structure.

The contents of this thesis have been published in part in the following papers by the author and co-author in recent years [9], [79], [80], [81], [82], [83], [84], [85].

1.8. Statement of Contributions

In this thesis, there has been an investigation of the research questions as well as proof of the propositions. Moreover, all proposed propositions have been evaluated with new Information Modelling theories and new distributed network architectures, and further results of the research contribute to the current knowledge system of Software Defined Networking and the network management paradigm. The key contributions have been made according to the following:

With regard to Chapter 3:

- A new theory of distributed active information model (DAIM) is proposed and designed as an alternative to the previous efforts on object-oriented management information base (O:MIB) and *hybrid O:XML* semantics [31]. The DAIM model is further integrated with the SDN architecture at the level of the switches to provide a physically distributed control plane that can manage the flow setups. Details of the DAIM framework are explained and documented. A journal paper and a conference paper based on this model have been published as contributions to the knowledge [83], [9].

With regard to Chapter 4:

- An effective DAIM model is implemented from scratch based on its theory and designed architecture as closely as possible. A new message channel is created to facilitate the communications between the controller and OpenFlow switches. This communication channel is implemented using UNIX BSD socket programming API to process OpenFlow messages without any modification.
- The three core modules that construct the DAIM ecosystem are also implemented including the communication module, the storage module and the controller module. Inside the communication module, there are various routines

for creating different types of OpenFlow messages. The storage module has two main components, the *Hosts* and the *Ports* table to store the information of network devices. The controller module is responsible for maintaining the connectivity with the switches and all control functions.

- An intelligent L2 switch application is developed in DAIM. When a packet arrives at a switch, DAIM can simply maintain a lookup table that associates the MAC address of the host with the port on which they are connected. DAIM stores the source MAC address of the packet and the incoming port in the table. Upon receiving a packet, DAIM looks up the destination MAC address and in the case of a match, it installs the necessary flows with a specific output port to its correct destination host.
- Currently, the cross-controller communication protocol has not been widely explored in the OpenFlow specification. It is one of the early approaches to be introduced based on the LLDP mechanism. The initial implementation can demonstrate that it is possible to share messages between DAIM controllers through the existing network medium.
- The feasibility of using distributed DAIM controller in different OpenFlow scenarios has been tested, discovered and documented. In addition, a set of experiments is carried out to verify the functionality of the implemented DAIM controller and to check the compliance of the overall system with its purpose and functions.

With regard to Chapter 5:

- The experimental validation of DAIM communication channel is conducted and shows its good performance regarding flow setup speed. The DAIM channel when connected to NOX and POX are compared to normal NOX and POX controllers with respect to handling events.
- The performance evaluation of the DAIM controller is carried out and shows its better performance with respect to throughput and latency. The DAIM controller is compared with the current SDN/OpenFlow controllers including NOX, POX and NOX-MT.
- Different network performance metrics comprising mean RTT, throughput, latency and bandwidth have been tested and discovered by the used of available and appropriate benchmarking tools (*Cbench* and *Iperf*).
- A small-size OpenFlow network has already been set up physically in the lab with a set of Raspberry Pis to show that DAIM can perform operational functions and can be deployed in real networks. The setup installation and configuration as well as some useful tests are also performed and documented.
- There are different factors such as programming language and device resources influencing the performance of an OpenFlow controller. The important parameters that can affect the performance evaluation have been analysed.

With regard to Chapters 3, 4 and 5:

- A reference implementation of the distributed DAIM controller is made available to help further research, trials and validation of the performance of scalability and robustness for other researcher's use. It is accessible on the public domain (GitHub) and is considered as a contribution to the SDN and the networking research community.
- The DAIM model with distribution management strategy is developed to enable autonomic communication as a promising solution to versatile service requirements. The implemented DAIM model will hopefully address the limitations of current approaches in the novel and future distributed network systems aiming at an autonomic management strategy.

2. Background and Literature Review

2.1. Introduction

Technological innovations in communication networks, computing applications, and information modelling have played a significant role in providing management services for large and complex systems. The adoption of these advanced technologies have dramatically escalated over the past few decades, especially distributed networks, which has led to an increase in the complexity of network systems. Using a human operator as the manager is not economical and also error-prone. As distributed systems grow significantly in both size and complexity, effective management requires monitoring, interpreting, and handling the behaviour of the managed resources to ensure required Quality of Service (QoS) and optimal network performance. Currently, large-scale electronic systems like Wireless Sensor Networks (WSNs) and bush fire alarm systems are becoming more difficult to manage, configure, operate, maintain and re-structure. It is important to propose a new Operations Support System (OSS) management structure to cope with such complex distributed networks and systems.

Currently, many network management systems pursue a platform centred paradigm, where all of the computation is controlled at a central location. As an example, in traditional Simple Network Management Protocol (SNMP), a fully centralised management paradigm is used. Agents are accessed by applications via management protocol to monitor the system and collect the network information. Furthermore, several researchers in the network management field believe that in most, if not all, network management problems can be addressed by using appropriate centralised systems and intelligence control. However, in today's real networks, there are many network management complexities and limitations that cannot be adequately solved by a fully centralised approach such as lack of flexibility and information bottlenecks.

Software-Defined Networking (SDN) is one of the most promising and disruptive network management paradigms in recent years. SDN has gained considerable attention from both academia and industry with the potential to imply network innovation and create choice, which help realise new capabilities and address persistent networking challenges. SDN can also allow network administrators more control of their infrastructure providing customisation and optimisation. This results in significant reduction in the overall operational and capital costs.

OpenFlow-based SDN, initially developed by Stanford University in 2008, is a way for researchers to run their experimental protocols in networks that carry production

traffic. OpenFlow is regarded as a useful project tool for researchers to easily deploy innovative applications like DAIM, in which some or all of the intelligence and management control are logically distributed within the network entities. OpenFlow will be used as a standardised communication protocol to employ and extend functionalities of the DAIM model.

This research is also applicable to the area of communication networks of wired and wireless telecommunication networks. The DAIM model is proposed with the hope to solve new core challenges in managing future heterogeneous and large-scale distributed networks. An effective DAIM model can provide a distributed computing environment that will enable the richness of adaptation algorithms to be implemented in such complex systems. These algorithms could be implemented with intelligent agents that are essentially distributed, and have some self-learning strategies to make their local decisions according to the changing topology. The DAIM structure can also be applied to other distributed communication networks to enable autonomic functions, and therefore reduce the high-level decision making of a centralised remote controller.

In this chapter, the FCAPS network management principles and some conventional network protocols, including their advantages and disadvantages are investigated. Because centralised management cannot stand and achieve the requirements for distributed network environments, the needs of distributed systems in next generation networks are further described. Moreover, a background review of fundamental SDN paradigm is discussed and the associated scalability problems, as well as related work solutions in this area, are covered afterwards. Three approaches to solving OpenFlow scalability issues are summarised because they are the most relevant literature to the research presented in this dissertation. This chapter concludes with an overview and the current state of the art in the autonomic communications network.

The Autonomic Communications (ACs) is a concept to deal with the ever-growing complexity of distributed networks. This term gives systems the ability of self-management, which means each component in ACs can adapt itself to changing conditions of the dynamic environment. Autonomic networks with high-level functions can manage themselves and create their adaptation strategies driven by the system requirements to support self-management capability. The concept of autonomic behaviour is classically encapsulated in the self-x capabilities enunciated by IBM. Using DAIM, we can imagine how a logically distributed control plane in an OpenFlow network could be made to exhibit autonomic behaviour.

2.2. Network Management Background

In the last few years, network technologies have been increasing significantly in performance, complexity and functionality driven by the needs of the modern world. However, existing network infrastructure lacks adaptability, and demands device

centric and centralised management paradigms. Networks today have become massive and intractable due to complexity leading to challenges of scalability. Moreover, communication networks need to support a wide range of services and functionalities with the capability of autonomy, scalability and adaptability for managing applications to meet business needs. Networking devices are increasing in complexity among various services and platforms from different vendors. Managing these complex networks can bring many challenges and may require expert operators. New network management paradigms may take several years to develop, and much longer to become widely spread. From the vendors' side, they have a lack of standard and open interfaces, and there are limitations on the ability of network operators to design the network to meet different individual requirements. This makes a gap between market requirements and network capabilities [14]. Also, vendors do not have standardised and open interfaces. Hence, there is a need for open and flexible architectures to implement autonomic management functionality, which has been considered as a promising solution to ameliorate the complexity of network management [105]. This section focuses on the FCAPS network model and framework for network management and some of the conventional network management protocols such as SNMP and NETCONF. Different network structures of centralised and distributed management paradigms are also explained.

2.2.1. Five ISO Functional Areas in Network Management

International Organisation for Standardisation (ISO) has classified and standardised network management into five major categories: Fault, Configuration, Accounting, Performance, and Security management. This standardisation is also widely known as FCAPS. It is used to organise and sort network management techniques. In today's market, most of the network management tools support FCAPS via Simple Network Management Protocol (SNMP). This section will describe the five functional categories of the FCAPS framework.

- **Fault Management**

Boutaba and Polyrakis (2001) define network fault as "network downtime, poor performance and service degradation". The main purpose of fault management is to detect, log network faults, and inform individuals of the problem and remotely fix them to have the network working properly. Fault management increases the network's reliability by giving the network management tools to quickly detect problems and initiate recovery procedures. Fault management also deals with identifying fault symptoms within the network and isolating the problem [86]. Regarding fault management tools, network managers use them to detect or predict unusual events. If a network fault is identified, it is either resolved automatically or reported to network administrators to fix the problems manually. After the problems are resolved, the solution needs to be stored for future use in case a similar event occurs [19].

- **Configuration Management**

Configuration management is a procedure of collecting configuration information as well as updating the relationships between each component and its status during network operation. Configuration management is also responsible for managing network equipment remotely. According to Rao and Mohapatra (2010), configuration management aims to monitor configuration information in networking devices, so that the effect on network process of new versions of software and hardware elements can be identified and managed. Additionally, configuration management information is recorded in a database and when a problem arises, this configuration database can be searched to solve the problem [86]. Moreover, network professionals can add and modify network configuration remotely from a centralised location by using configuration management tools. These tools can perform many tasks such as auto discovery and auto mapping of new devices to maintain consistent topology information.

- **Accounting Management**

Rao and Mohapatra (2010) point out that the purpose of accounting management is to understand the behaviour of converged networks. This can be accomplished by having an inventory of networking devices, clients, bandwidth consumption and analyse the information to provide current network utilisation data. In addition to this analysis, the quotas of resource usage such as memory, bandwidth, CPU, and schedule priority can be set to users fairly [19], [86]. Accounting management also refers to managing the user and administration as well as billing users for using the network resources and services. Account management includes tasks such as setting the cost for network usage, giving access to use network resources, logging network utilisation, name and address registration, and charging the clients according to this. Because the users are charged for the services that they consume, the billing process is more accurate.

- **Performance Management**

Network performance is characterised by multiple performance characteristics and can be measured according to throughput, delay, loss and jitter. Hence, performance management deals with measuring the performance of a network as well as maintaining the network performance at a satisfactory level [19], [86]. One of the most important tasks of performance management is data collection. By collecting statistical performance data, network operators can have a better understanding of the current network load. They can also develop long-term trends for capacity planning based on the current growth. Furthermore, network administrators can use performance management tools to measure the overall throughput, average utilisation, error rates and maintain to control them. However, performance management is often related to setting performance thresholds such as TCP links and traffic of packets sent and received, so exceeding these thresholds on a variety of items can affect the network performance.

- **Security Management**

The purpose of security management is to secure the safety within the network environment. It also aims to control traffic in and out of a secure network, which prevents a network from external attacks. For example, intruders attempting to place a virus on critical servers, reconfiguring, or causing network faults that will debase the network performance. Moreover, functions of security management can protect sensitive information on network devices by specifying the configuration of network clients and devices to control access point. Security management functions can also advise network administrators of actual or attempted breaches of security. Rao and Mohapatra (2010) state that security management deals with controlling users' access to network resources as well as preventing them from damaging or accessing unauthorised devices and data. To meet these security management goals, network operators need to identify and secure access points, analyse potential threats, conduct security policies, and monitor the user's identities to network resources.

2.2.2. Network Management Protocols

Current networks have many restrictions, including difficulties to meet the business and technical needs over the past few decades. The industry has developed protocols for the networks to provide a high performance, reliability and greater connectivity, and to make security more stringent. Some of the important network management protocols are explained in this section. The new DAIM model is proposed as an alternative approach to these conventional schemes.

1. **SNMP**: First protocol for network management was Simple Network Management Protocol (SNMP) proposed in 1988, which was easy to use and did not need complex management support, was flexible and could be used in most of the devices. Moreover, it was the first protocol that was widely accepted and used for a long period of time. SNMP protocol consists of three parts: managed devices, agents, and network management system. Agents are software modules running on each managed device, which can provide management information of the device asked by the manager through a communication interface called Management Information Base (MIB). MIB is a database that holds important management information used for managing the entities in a communications network. As this communication is in-band, if any problem occurs in the network, it is almost impossible to diagnose and recover the network without using an external device. Management information in devices is always different from MIB; a table called method routines is defined in SNMP in order to implement the access mechanism to management information on each device. Another disadvantage of SNMP is the lack of standard definition of mentioned implementation. There are also other shortcomings as follows: completely centralised and agents rarely have an active role; inferior scalability and inflexible; cannot handle the massive increasing size and complexity of networks [40], [24].

2. **NETCONF**: The limitation of SNMP has lead to the development of alternative approaches in managing large-scale and complex network environments. One of the newly approved network management protocols proposed by IETF in December 2006 is the Network Configuration Protocol (NETCONF). It is a document-oriented approach based on Extensible Markup Language (XML) technology that aims to address the weaknesses of SNMP, especially the application in configuration management. NETCONF protocol is regarded as the next generation of automated XML-based network management system. This is because the communication between the NETCONF manager and the agents are formed in an XML document and based on XML-encoded Remote Procedure Call (XML-RPC) [119]. Moreover, NETCONF can also upload and retrieve configuration data of the network devices separately with high-level configuration operations. To assure the security of message transmissions, NETCONF adopts a transport independent protocol so-called Simple Object Access Protocol (SOAP) [25]. The NETCONF protocol brings many great advantages when compared to SNMP. For example, NETCONF provides more advanced functionalities, more effective transactions of complex configuration data, it is more secure and much easier to develop new applications than SNMP. Although NETCONF protocol is better than SNMP in some aspects for instance in configuration management, there are also some important drawbacks associated with this approach. Firstly, one major issue related to NETCONF is the lack of support from industries and because there are few publications regarding NETCONF implementation. Secondly, new elements of NETCONF security aspects should be added, especially for access control. For example, using an XML-based access control standard, the eXtended Access Control Markup Language (XACML) as a good open source support [117]. Thirdly, there is a need for a new data model and new data modelling language as this plays a significant role in the universality of NETCONF.

3. **Policy-Based Network Management**: Policy-based management framework was proposed by Internet Engineering Task Force (IETF) in 2001 [1]. Applying this method can make a network highly automated in terms of configuration. Policies that are defined by the administrator in this method will be used to configure all of existing or future devices of the network, and there would be no need for the admin to configure any device itself. As a result, the administrator's configuration task will be simplified and this task will be done automatically. There are four elements in this framework (see Figure 2.1) as follows:

- Policy repository which is used to save the policies made by the administrator.
- Policy enforcement point which refers to a controlled device in the network.
- Policy decision point which is a communication element between enforcement point and repository, as these two elements can be placed in same or different devices; previous protocols such as SNMP can be used to make communication between these

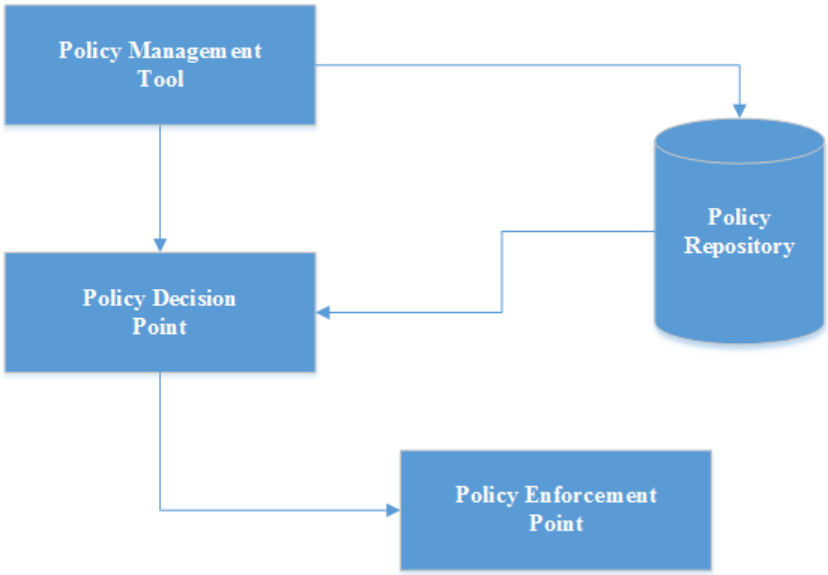


Figure 2.1.: Four Elements of Policy-Based Framework [112]

two elements.

• Policy management tool that enables the administrators to define their desired policies. In this element, two levels of policies were defined, (1) business level that is based on business needs as well as the user language (does not include technical terms and specifications), and (2) technology level policy that is the interpretation of business terms into a technically applicable policy for devices.

The policy management tool consists of four basic elements. Firstly, a user interface is a tool that the administrator can input the policies in a form of business levels. Secondly, a resource discovery that verifies the network topology. Thirdly, transformation policy logic is the most important aspect of this framework, which guarantees reliability, suitability and practicability of the network as well as the administrator's policies. Fourthly, a policy distributor that distributes the policies to all network devices [112].

Although policy-based network management brings many benefits such as the state of being automatic and highly user-friendly, the main disadvantage of this method is using a centralised information model where all policies are defined by a central manager. By increasing the complexity of networks, conflicts between high-level policies will increase and, as a result, the framework performance will drop or the network will not operate functionally.

2.2.3. Centralised Management Paradigm

The increasing adoption of advanced technologies in communication networking, computing applications, and information modelling have played a significant role in

providing management services for large and complex systems. As the complexity of the centralised system grows over time, effective management requires monitoring, interpreting, and handling the behaviour of the managed resources to ensure required Quality of Service (QoS) and improve networks performance.

This subsection briefly reviews the centralised network paradigm and some of its limitations. Figure 2.2 illustrates a centralised network that consists of one single computer server and a number of computers connected to that server by using Operating System Communication Application (OSCA) [71]. In a centralised network, this server will have its own operating system and applications to configure the communicated computers, which also have their own operating systems and applications.

The centralised network (star) paradigm has a single node as a core node, and multiple nodes connected to that core node where each node has an operating system and applications. The core node is able to configure all connected nodes using an OSCA. Many network management systems today pursue a platform centred paradigm, where all of the computation is controlled at a central location. As an example, in traditional Simple Network Management Protocol (SNMP), a fully centralised management paradigm is used. Agents are accessed by applications via management protocol to monitor the system and collect the network information. Moreover, researchers in the network management domain state that a centralised network implies a central point of control, and through effective management, it can control customer service, speed, reorganisation and flexibility within the network [62]. However, in today's real networks, there are many network management complexities and limitations that cannot be adequately addressed by a fully centralised approach such as the possibility of a complete failure and traffic bottlenecks.

The centralised management paradigm has a major drawback. For example, increasing the number of connected nodes will affect the performance and the processing power of the core node. It will also increase the overhead on the server because of the increasing number of connected computers [65]. In addition, this type of network is often vulnerable if destruction happens to the central server, which can terminate the communication between end nodes [12].

Centralised implementations are also inefficient to handle the huge number of high-level decision makings. This subsection compares the centralised paradigm with the distributed system paradigm (decentralised), in which some or all of the intelligence and management control is locally distributed within the network entities. In wired networks, distributed system minimises the complexity that occurs in layer 3 devices (e.g. routers) by distributing some roles into layer 2 devices (e.g. Ethernet switches). OpenFlow is an example of a network that may be able to apply new distributed models. In wireless networks, a distributed system also minimises the complexity that occurs in the core network. For example, the Radio Network Controllers (RNC) such as in Beyond 3G (B3G) network can reduce complexity by distributing some management functions (*e.g.*, decision making for allocation of radio resources) into

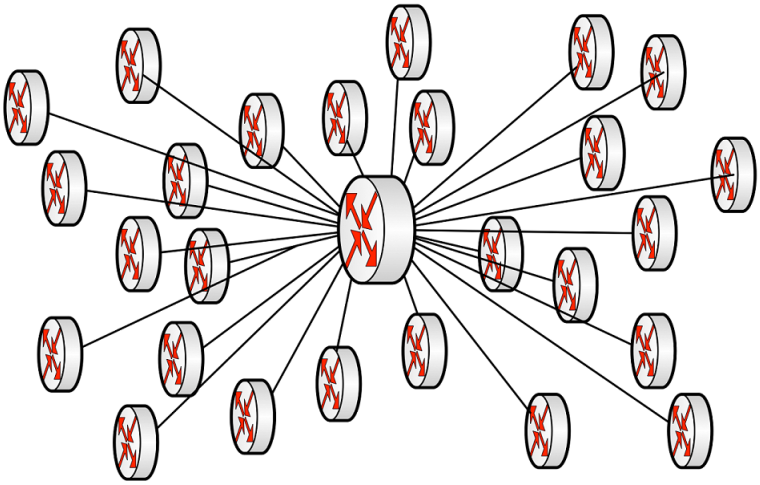


Figure 2.2.: Centralised Network Paradigm [84]

the Evolved Nodes B (eNodes B). In Long Term Evolution (LTE) networks, the management functions are distributed into Base Stations (BSs) such as in mobile Worldwide Interoperability for Microwave Access (WiMAX) [90], [87].

However, the main advantage of the centralised architecture is that one core node can be responsible for managing all connected nodes, and thus managing the entire network from a single point [65]. As a result, this research expects that the Next Generation Network (NGN) should rely on distributed paradigm such as mesh and grid networks.

2.2.4. The Needs of Distributed Systems

Over the past twenty years, network technology has been improved rapidly regarding speed, performance, components and functionalities. Many different types of network devices have been developed and this has led to an increase in the complexity of network systems. Traditional network structures are inadequate to meet today's requirements. It is the centralised network that requires human operators to have a high level of experience on how to detect changes, configure new services, recover from failures and maximise Quality of Service (QoS). Therefore, network management involves heavy reliance on expert operators. The adopted centralised network management is not suitable for new technologies emerging, which are complex and difficult to interact with heterogeneous networks that contain different types of services, products and applications from multiple vendors. As a result, the current network management lacks efficiency and scalability; however, it has an acceptable performance in general. The centralised information model cannot stand and achieve the requirements from such complex, distributed electronic environments.

In recent times, there has been a growing movement led by both industry and academia, aiming to design mechanisms to reach a control model which achieves

the separation of the control plane from data plane and to build it as a distributed system [40]. Different network management examples show that centralised systems have some fundamental problems. This subsection discusses the needs of distributed systems and shows that a decentralised network system has many benefits over a conventional centralised network, and would be a promising evolutionary approach in the network management development of next generation networks.

Traditional static networks are struggling to meet the rapidly growing requirements of today's enterprises, carriers, and end-users. Besides, as the networks increase in size and complexity, a distributed networking system is needed to ensure good quality of network services and performance. A distributed network system can refer to an application that executes a set of protocols to correspond with the actions of multiple processes on a network [108]. Moreover, all components work together to operate a single or small set of related tasks. The devices that are in a distributed system can be physically linked together and connected to a local network. They can also be geographically distant and connected by a wide area network [49]. A distributed system can consist of any number of possible configurations such as personal computers, minicomputers, mainframes and workstations.

A distributed system aims to make a network environment work as a single computer in order to cope with the extremely significant demand of users in both data storage and processing power. Examples of distributed systems may include Distributed File Systems (Hadoop), P2P Network, Cloud Network, Grid Computing, Web Server and Indexing Server, and Pervasive Computing.

There are several advantages such as the ability to connect remote users with remote resources in an open and scalable way. Regarding open, we mean that each component is continually open to interaction with other components. Regarding scalable, we mean that the system can easily be altered to accommodate changes in the number of users, computing entities, and resources [108].

Therefore, a distributed system can bring many benefits given the combined capabilities of the distributed components, more than the combinations of stand-alone systems [59]. However, it is not enough for a distributed system to be useful; it should provide system reliability as well. This is a very difficult goal to achieve due to the complexity of the interactions between simultaneously running components. Birman (2005) indicates the concern for reliability in distributed computing systems [18]. The following Table 2.1 summarises the characteristics of a reliable distributed system.

Some of the key computing trends driving the need for a distributed system paradigm include [44]:

2.2.4.1. Change of Traffic Patterns

Regarding the enterprise data centre, traffic patterns have changed dramatically. In contrast to client-server applications, the bulk of the traffic occurs between one client

Characteristic	Description
Fault Tolerant	It can recover from network failures without performing incorrect actions.
Highly Available	It can restore operations, instructing it to resume network services even when some components have failed.
Recoverable	Failed components can reboot themselves and re-join the system after the cause of failure has been recovered.
Consistent	The system can execute corresponding actions of multiple components often in the case of concurrency and failure. This underlies the ability of a distributed system to act as a non-distributed system.
Scalable	It can operate properly even some aspect of the system is scaled to a larger size. For example, if the number of users or servers increases, the overall load on the system should not have a significant effect.
Predictable Performance	The ability to provide desired responsiveness in a timely manner.
Secure	The system authenticates access to data and network services.

Table 2.1.: Characteristics of a Reliable Distributed System

and one server, applications today access various databases and servers, and thus creating a flurry of "east-west" machine-to-machine traffic before returning data to the users' device in the classic "north-south" traffic pattern. Meanwhile, network traffic patterns are changed by users as they push for access to corporate content and applications connecting from any type of device anywhere and anytime. Moreover, many enterprise and carrier managers are contemplating a utility computing model, which might include a public cloud, private cloud, or a mix of both. This may result in additional traffic across the wide area network.

2.2.4.2. The Consumerisation of IT

Users are increasingly employing mobile technology such as smartphones, tablets, and laptops to access the corporate network. This can cause pressure for IT to accommodate these personal devices in a fine-grained manner while protecting intellectual property as well as corporate data and meeting compliance mandates.

2.2.4.3. The Rise of Cloud Services

The high demand of enterprises for both public and private cloud services has resulted in an unprecedented growth of these services. Today's enterprises want the ability to access applications, infrastructure, and resources on demand. Providing

self-service provisioning in either a public or private cloud requires flexible scaling of storage, computing, and network resources, basically from a common viewpoint and with a common suite of tools.

2.2.4.4. Huge Data Demand More Bandwidth

Dealing with today's mega datasets requires efficient parallel processing on thousands of servers, which all need direct connections to each other. This emerging trend of mega datasets has led to a constant demand for additional network capacity in the data centre. Administrators of large-scale data centre networks face the daunting task of managing the network to previously unimaginable size due to ever increasing network complexity.

2.3. Standard Sets of Information Model

An information model in software engineering and data modelling is a structured representation of concepts, rules, the relationships and operations that specifies relations between kinds of things or relations with individual things. The term of the information model is typically an abstract, formal representation of entity types that may include their properties, the relationships and operations that can be performed on them. The information model can build a data model, which is embedded in a platform, language and protocol. The data model is used to define how data is structured and accessed to specify data semantics and for implementation purposes [103]. The common standard sets of information models that are used in network management to enable self-management functionalities in heterogeneous networks and complex electronic environments include:

- The Distributed Management Task Force (DMTF) Common Information Model (CIM).
- The TeleManagement Forum (TMF) Shared Information and Data Model (SID).

The SID model uses *roles* that can be played with network elements in a scalable manner. Also, roles in the SID model are not limited to just people, rather, they may also represent products, locations, resources or other managed entities of interest. The CIM model enables multiple parties to exchange information of managed elements. In addition, it provides the ways to control and manage these elements actively. These information models cover most of today's elements in IT resources such as computer systems, communication networks and operating systems. Moreover, information models have a lot of similarities and overlap in the domain they cover and currently they are used by a number of different management solutions.

2.3.1. Common Information Model (CIM)

A number of standardised information models are currently industry standards. The standard of Common Information Model (CIM) has developed by Distributed Management Task Force (DMTF), with the goal to produce an object-oriented scheme to model the hierarchical data of the managed IT environment. CIM is a conceptual view of the managed environment that attempts to unify and extend the existing traditional management standards such as SNMP and Desktop Management Interface (DMI) using object-oriented constructs and design [3]. Moreover, CIM can provide a consistent definition and structure of data by presenting managed elements as a basic set of objects and relationships between objects. This standard includes the CIM infrastructure specification and the CIM schema. In regard to its infrastructure, the managed objects are described as a class, and the relationship between them are represented by associations. In addition, CIM applies object-oriented concepts of inheritance to effectively define the common framework of managed objects and inherited sub-objects [26]. The values of object orientation techniques from CIM also provide support for object design with the following capabilities:

• Classification – High-level and fundamental concepts are defined when objects are grouped into types (class), identify common features and characteristics (properties), relationship (associations), and behaviour (methods).

• Object inheritance – Subclassing the high-level and fundamental objects. A subclass inherits all the information (properties, associations, methods) defined for its higher level objects. Sub-classes are created to manage the same level of detail and complexity at the same level in the model.

• Ability to show dependencies, components and the connection or relationship between objects.

• Standard inheritable methods – The capability to identify standard object behaviour (methods) and encapsulate standard methods with an object's data.

Regarding the structure of data modelling, CIM scheme can provide greater representation of information than SNMP static MIBs. Conventional SNMP MIBs, on the other hand, have been used in IT industries for decades since the ISO layer model was introduced. Moreover, SNMP MIBs describe the information of managed objects from a different view in contrast with CIM models.

2.3.2. Shared Information and Data model (SID)

Another standard set of the information model is the Shared Information and Data model, or SID. It is one of the frameworks defined by the Telecommunication Management Forum (TMF) as an advanced object model for the telecommunication domain. The SID model uses the Unified Modelling Language (UML), a standardised general-purpose modelling, to define entities and the relationships between

them as well as the attributes and operations/methods that make up the entity or object. The SID model is a unified data reference model that provides common information and data vocabulary, and common language for communicating with business, system, implementation and deployment. They include views of the business, service and resource domains within the telecommunication industry as the standard rules for OSS designers [26]. The main objective is to provide a single set of terms for business objects in telecommunications and enable people from different departments, geographical locations or companies to use this same term to describe the same real world objects, relationships and practices. By providing definitions that are simple to understand, the SID model is applicable to most network resources, which also makes it possible to align between businesses as an organisation and IT as software.

Although the released specification of SID model has covered the business domain and the device management field well, SID was insufficient in its ability to support information integration, logical networks and capacity [45]. These limitations are currently addressed through the revision of the model by including concepts such as topologies. However, the results have shown poor utilisation of the model in certain telecommunication fields in terms of inventory management.

2.3.3. Limitations of CIM and SID

The SID and the CIM are not related to each other. Traditionally, the SID has a different operational domain than the CIM model. The CIM mainly focuses on the IT resources such as systems, storage and IP networks. On the other hand, SID provides a description of technical and business systems. However, SID and CIM principles are overlapping and have a lot of similarities, where both are used by many in management solutions.

The TeleManagement Forum (TMF), producer of the SID model, have tried to align with the Distributed Management Task Force (DMTF), producer of the CIM model, for more than five years of collaboration. They have joined their efforts together to start a new project that aims to develop an alternative approach, which can apply to other domains.

This project was divided into two phases. The first phase includes all the results presented in these two documents: (1) DMTF/TMF Model Alignment Physical sub-Model Alignment, DMTF DSP2004/TMF GB932; and (2) DMTF/TMF Model Alignment SID Logical Resources and CIM Networks Sub-Models, DMTF DSP2000/TMF GB933. Moreover, this phase identifies physical and logical resources. The second phase was aiming to identify guidelines and processes for CIM-SID model mapping, with a focus on logical and physical resources as identified in the first phase.

Unfortunately, this collaborative project was not successful due to three major issues [103]. The first issue relates to the concepts of building the CIM model such as

classes, attributes and relationships. CIM does not use UML as a data and process modelling language to construct its proprietary meta-model, whereas SID uses UML to define entities, relationships and attributes between them [103], [104]. The second issue is that CIM does not use *patterns* in model design. Unlike CIM, the SID framework uses *patterns*, where "patterns" in this context are the enabler to reuse the successful designs and can help produce models that are easier and faster to learn. The third issue is that CIM does not use *roles* like the SID model. Roles make a design more scalable by abstracting individual services, devices and users into roles that can be played by different managed entities [103], [104]. Entities can also play many roles while retaining a basic set of facts about the entity in general.

2.3.4. Current Information Models vs. Proposed DAIM Model

The CIM model is a conceptual scheme that encapsulates all the Managed Elements (MEs) such as devices, applications, services, storages, network systems and computer systems. Essentially, CIM provides a common management information of various applications and devices, so that network administrators can easily access them in a universal way. CIM also applies object-oriented notation and can be described in XML syntax in several ways. Similar to CIM, Shared information model (SID) also uses UML to formalise the expression of the needs in terms of business view. However, the CIM schema contains limited methods or implementations for each class, as predefined methods are limited to the static class methods. Furthermore, executing corresponding actions are still in need of separate control (*e.g.*, centralised control) to be achieved, which is independent of the CIM structures.

In comparison, the proposed DAIM model is not only providing a distributed service but also aiming to adopt the object-oriented principle to manage the entire network effectively. OpenFlow-based SDN can be considered as a possible environment to implement the DAIM model, where the DAIM controller will be distributed to process the flow entries and the messages between network components. Doing this may essentially divert the CPU load into a large amount of local CPU of a current OpenFlow controller. However, distributed DAIM controller residing on each OpenFlow switch can use its built-in methods, algorithms, attributes and messages to dynamically invoke the execution of corresponding actions. Most of the decision-making tasks are processed locally at the switches, and thus reduce the workload of network administrators or management systems in CIM standard. Consequently, the DAIM model is designed particularly for distributed components with local execution capabilities to improve the network's overall performance [20], [31], [28].

2.4. Software Defined Networking (SDN)

"Software Defined Networking (SDN) may often be referred to as a revolutionary new idea in computer networking and promises to dramatically simplify network

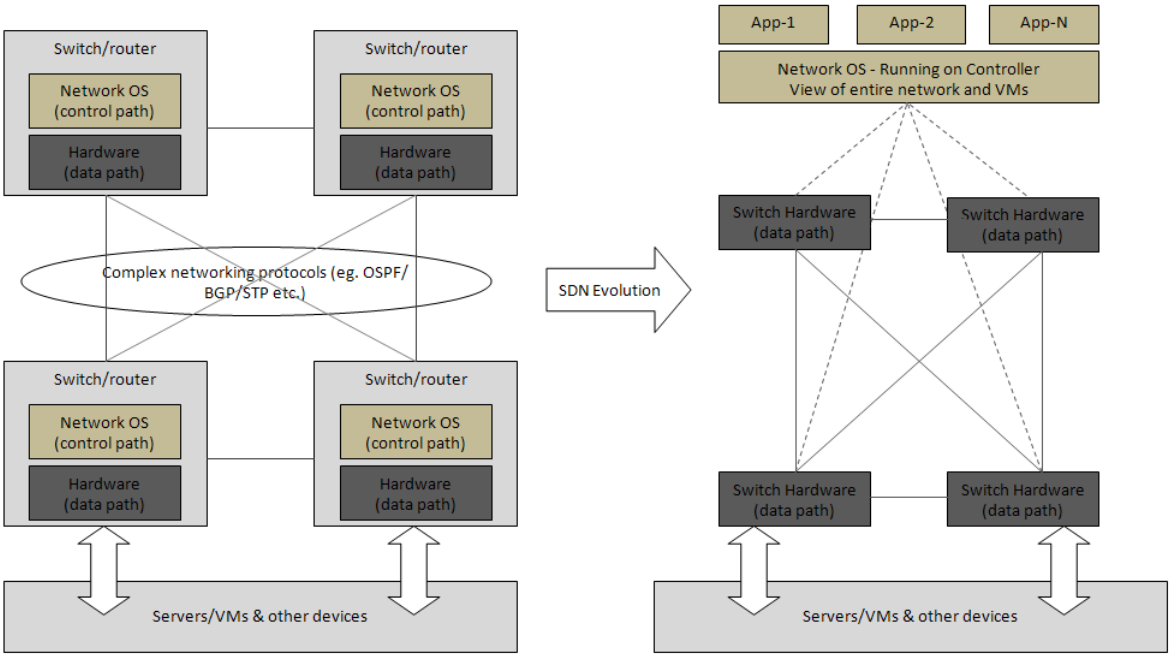


Figure 2.3.: SDN Evolution - Segregation of Control and Data Plane [97]

control, management and enable innovation through network programmability" [6].

The future of the networking domain will rely more and more on software. Software-Defined Networking, standardised by a non-profit industry called the Open Networking Foundation (ONF), is an emerging network architecture that seeks to transform traditional static networks into flexible programmable platforms by decoupling the network control and data planes. This is a migration of control from being formerly tightly bound to individual network devices, now into accessible computing devices. In addition, network intelligence and state are logically centralised, can enable the underlying infrastructure to be abstracted for applications and network services, which can treat the network as a logical or virtual entity [68]. With SDN, carriers and enterprises can gain unprecedented automation, programmability and vendor-independent control over the entire network from a single logical point that will enable them to create highly scalable and flexible networks in order to meet the changing business needs.

In Figure 2.3, on the left-hand side of the diagram is the structure of conventional network deployments comprising of switches and routers interconnected to each other and devices such as servers and Virtual Machines (VMs) on the edge. Distributed protocols are used to build the control path that current network deployments rely on. After the control path is built, the data path is installed in device hardware and productive traffic can pass on a programmed route. In conventional networks, distributed protocols such as Border Gateway Protocol (BGP), Spanning Tree Protocol (STP), and Open Shortest Path First (OSPF) are dispersed across the network

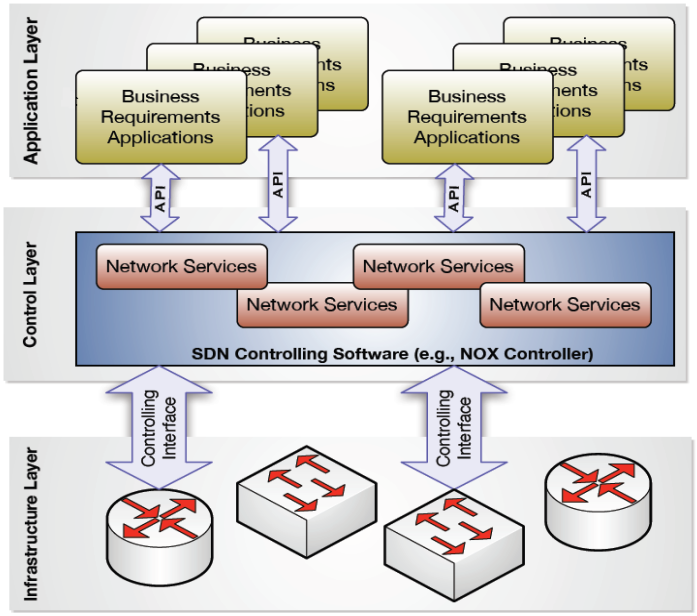


Figure 2.4.: The Three-Tier Logical Layers of SDN[44]

to make the control path and data path that are mainly Layer 2 and Layer 3 protocols.

On the right-hand side of Figure 2.3 represents the definition of SDN in the same network. The Network Operating System (NOS) will run on a single external device (*e.g.*, controller), and all of the high-level flow decisions will be made by this controller. After the decision is made, the controller will compute the data path of the switch making the network. The communication between the controller and OpenFlow switch via a secure channel is represented by the dotted line in the right half of Figure 2.3. The controller can obtain full network view of deployments from these links. The solid lines are normal data links in which data can flow once the routers and switches hardware are installed.

Figure 2.4 depicts the three tier logical layers of SDN, where network intelligence is (logically) centralised in software-based SDN controllers, which maintain a global view of the network. As a result, the network appears to the policy engines and applications as a single logical switch. SDN provides vendor independent control over the entire network from a single logical point and a programmatic interface for application designers to build a wide variety of management applications [60], [44], [51]. This can greatly simplify the network design and operation for enterprises and carriers. Moreover, SDN also greatly simplifies the network devices themselves, as they no longer need to understand and process thousands of standard protocols but merely accept instructions from the SDN controllers.

In essence, the SDN paradigm allows application designers to use a single control platform for developing a set of control functions such as routing, access control,

and traffic engineering over a range of control granularities in various contexts (*e.g.*, data centres, WANs, enterprises) [17]. Most importantly, network operators and administrators can programmatically configure this simplified network abstraction. They do not have to hand-code tens of thousands of lines of configuration scattered among thousands of distributed nodes [69]. This migration of control gives network operators the flexibility to manage, configure, secure, and optimise network resources via the SDN control software dynamically. As a result, they can efficiently alter the network behaviour in real-time as well as deploying new network services and applications.

By using OpenFlow-based SDN technologies, it is possible for IT to address the high demand for business needs, dynamic nature of today's services and applications, adaptation to the ever-changing network environments, and consequently reduce the complexity of network management and operations. OpenFlow-based SDN can bring several advantages to carriers and enterprises by modifying the network architecture in such a way including [44]:

• One of the greatest benefits of OpenFlow is creating flexibility in networks on how it will be used, operate and sold. The control program specifies the behaviour of the abstract model and configures the abstract network view, which can be written by network operators and administrators using common programming environments.

• OpenFlow increases the automation of management by using common APIs to abstract the underlying networking details from provisioning systems and applications to meet business objectives.

• OpenFlow-based SDN can introduce rapid innovation through customisation because network administrators can develop the features they need in the control program, instead of having to wait for software vendors to find a place in the plan for their trademark products.

• It can allow programmability by administrators, enterprises, users, and software vendors using common programming environment. It enables network virtualisation and network integration of computing and storage. This makes all the IT operations to be managed more easily by using a single viewpoint and tool set.

• It can increase network reliability and security as a result of centralised and automated management of networking devices, low configuration errors, and uniform policy statement.

• Users have a better experience since applications exploit the central view of network information to seamlessly adapt the network behaviour to user needs.

• OpenFlow-based SDN provides a standard way to convey flow table information to network devices and thus fosters open and multiple vendor markets.

This SDN paradigm requires some method for network control to communicate with the switch data path. One such mechanism is "OpenFlow" which is a standard interface for controlling computer networking switches. The OpenFlow platform is an

open standard that enables new opportunities to realise rich network control mechanisms and experiments by allowing researchers to flexibly program control path functionalities on the controller in production networks. The OpenFlow architecture separates the high-level routing decisions from the forwarding elements, which makes it possible to run in software and to be programmable [56]. Therefore, the OpenFlow switches become simple and flexible to manage. It only implements the data path packet switching functions. A specific node, the controller manages the control path functionalities and centralises the execution of all decision tasks by installing, deleting, and modifying flow entries on the OpenFlow switches. The OpenFlow protocol is used to communicate between the OpenFlow switch and controller. This protocol defines messages such as packets received, send packets out, modify forwarding tables, and get stats [80]. As a result, companies can gain programmability, automation, and the control of the network, which enable them to build highly scalable, flexible networks that can be easily adapted to different changing environment [5]. Recent development techniques enable dynamic reprogramming of the devices through the data flow.

2.4.1. Overview of OpenFlow-Based SDN

OpenFlow was initially developed by Stanford University in 2008 as the first standard communications interface defined between the control and forwarding layers of the SDN architecture. It is a networking protocol that allows researchers to operate experimental protocols in switches and routers in a uniform way, without having to expose the vendor's internal workings of their products. OpenFlow is based on an Ethernet switch with the goal to encourage networking vendors to apply OpenFlow features to their product devices for deployment in university backbone networks and wiring closets. OpenFlow could serve as a useful project tool in proposed large-scale test bed as well as offering researchers the ability to evaluate their ideas in real-world traffic settings. The OpenFlow protocol is a key enabler for SDN, and it allows direct manipulation of the forwarding plane of network devices. OpenFlow is being widely adopted by infrastructure vendors, who typically have implemented it via a simple firmware or software upgrade [74], [97], [68].

A protocol like OpenFlow separates network control from switches to logically centralised control software. OpenFlow can be compared to the instruction set of a CPU. The protocol specifies basic primitives that can be used by an external software application to program the forwarding plane of network devices, just like the instruction set of a CPU would program a computer system [6]. OpenFlow can be implemented on both sides of the interface between network infrastructure devices and the SDN control software. OpenFlow uses the concept of flows and identifies network traffic based on predefined match rules, which can be statically or dynamically programmed by the SDN control software [46]. It also allows IT to define how traffic should flow through network devices based on parameters such as usage patterns, applications and cloud resources since OpenFlow allows the network to be

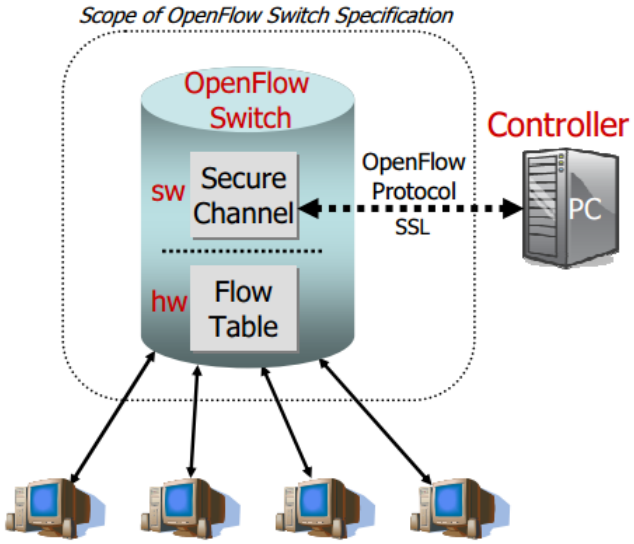


Figure 2.5.: Idealised OpenFlow Switch. A remote controller manages the Flow Table via the Secure Channel.

programmed on a per-flow basis. An OpenFlow-based SDN architecture provides extremely granular control, enabling the network to respond to real-time changes in the application, user, and session levels. Figure 2.5 is an example of an OpenFlow switch implementation.

Before continuing, it is helpful to define the term of what a flow means**.** In general terms, a flow is a set of packets transferred from one network endpoint (or set of end points) to another end point (or set of end points) that share a set of header field values. For instance, a flow could consist of all packets with the same VLAN ID or all packets with the same IP source address and destination address. The end points may be defined as IP address, TCP/UDP port pairs, VLAN end points, L3 tunnel end points or input ports, among other things.

The basic idea of OpenFlow is providing an open protocol to program the internal flow table and standardised interface in different switches and routers to add and remove flow entries. In addition, researchers can control a portion and a flow of their local network by choosing the routes of the packets and also the processing they retrieve. By adopting OpenFlow, researchers can run experiments on new networking protocols, security testing, and addressing schemes without disrupting others who depend on the production traffic [8]. OpenFlow provides high flexibility of novel packet forwarding and routing of network flows that can be adapted to real-time demands and needs by using virtualisation and flow-based routing. However, the flow tables are controlled by a remote centralised controller via the Secure Socket Layer (SSL) connection. This notion has raised legitimate questions concerning performance, scalability, and reliability of a centralised controller [60].

In traditional switches and routers, the fast packet forwarding (data plane) and the high-level routing decisions (control plane) operate on the same device. An OpenFlow-enabled switch separates these two functions. Using this strategy, OpenFlow allows network element devices such as routers and switches to be programmable via a standardised interface. The data plane portion still resides on the switches, while all of the high-level decision makings are moved to a separate controller or a standard server. This allows the controller to modify the forwarding rules in modern switches and enable implementations of user mobility, virtual networks, and new network protocols for example [51]. The controller communicates with the OpenFlow switch via the OpenFlow protocol, which defines messages such as, packets sent out, packets received, modify forward tables, and get stats. By using this protocol, the controller can also proactively add, update, and remove flow entries in response to packets [33].

2.4.2. Packet Processing in OpenFlow

The handling of incoming packets where an OpenFlow switch performs the functions is shown in Figure 2.6. Whenever an OpenFlow switch receives a flow's first packet, in which it has never seen before nor has a matching flow entries, that first packet will be transmitted to the controller. The first packet is called a "Flow Request". The controller then makes a decision and computes a path for this flow, and installs flow entries on every switch throughout that chosen path. Ultimately, the packet itself will be forwarded back to the origin switch from the controller and sent to the destination.

The forwarding plane of an OpenFlow switch contains one or more flow tables abstraction. Each flow table entry consists of a set of packet fields to match (*e.g.*, switch ingress port and different packet header fields), counters, and an action (*e.g.*, modify-field, send-out-port, or drop). Each entry is identified by its expendable match fields that contain various packet header fields and the switch ingress port. When an OpenFlow switch receives a packet, it tries to match the ingress port and packet headers within match fields of the flow entries. The matching can continue to additional flow tables starting from fully defined entries and to less defined entries. This will allow further processing and provide information, in the form of metadata, to communicate between tables. If similar flow entries are defined, in that case, a priority field indicates which entry should be selected.

The packet header fields in the flow table look-ups are matched depending on the type of the packet. This typically includes various match fields such as the MAC source address or IP destination address. Matching packets with flow table have two methods; one is specific match and the other one is wild-cards. For the specific matching, the packet arrives to the flow table with many identification fields such as source/destination address, MAC address, Ethernet type, TCP, IP, and exact port(s), whereas the wild-carded matching receive packets and send them to any

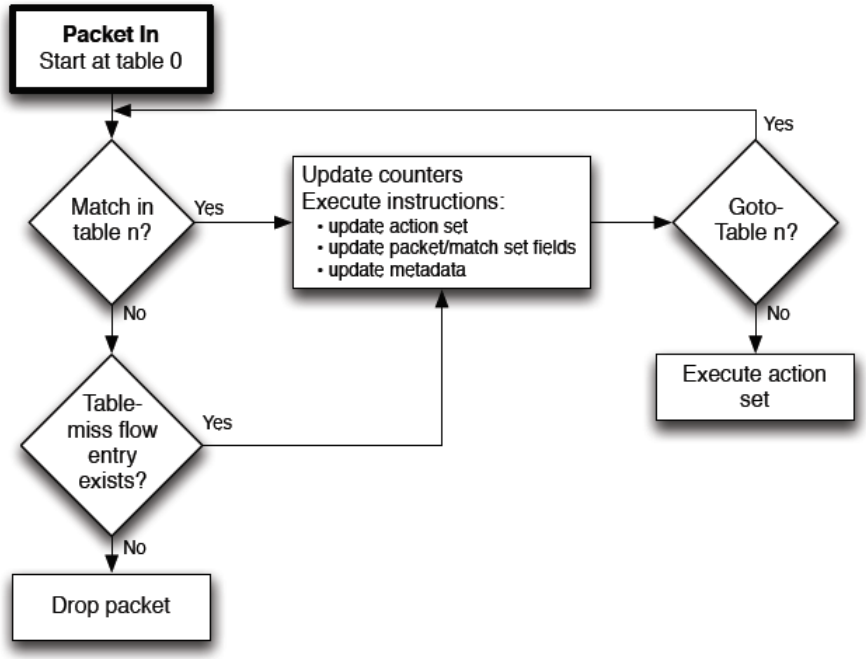


Figure 2.6.: Flowchart Detailing Packet Flow Through OpenFlow Switch [33]

port(s), address and destination, often depicted to in diagrams with the identification (*), which means has a value of ANY. An OpenFlow switch must update the associate counters and apply the instruction set of only the highest priority flow entry that matches the packet [11], [33].

The instructions specified in the associated flow entry are executed when a matching entry is found in a flow table. However, if an OpenFlow Switch receives a packet with no matching rules of flow entries in the flow table, depending on the switch configuration, the default is to send packets to the controller. The controller will first verify this flow against security policies. It can drop the packet, or it can install a flow entry directing every switch along the chosen path how to handle similar packets in the future [33].

2.4.3. OpenFlow Switch

An OpenFlow switch is a basic forwarding element in the data plane, which is accessible through OpenFlow interface and protocol. It is a typical networking switch that lacks the intelligent and distributed protocols. In an OpenFlow network, OpenFlow switch is classified into two types, Hybrid (OpenFlow-enabled) and Pure (OpenFlow-only). Hybrid switches support OpenFlow in addition to traditional Ethernet protocols (L2/L3 switching). Pure OpenFlow switches have no legacy features and support only OpenFlow, which completely rely on a controller for high-level decisions. Currently, most available and commercial switch implement-

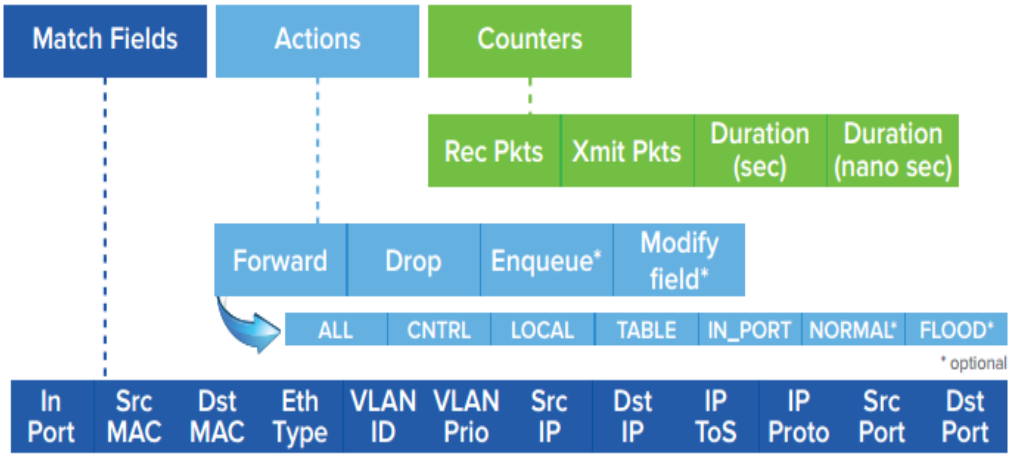


Figure 2.7.: Components of a Flow Entry in a Flow Table

ations are hybrids. Because OpenFlow switches are managed via an open interface, it is crucial for that communication link to remain secure and available. Regardless of switch type, the OpenFlow switch needs to support a module to establish a connection (SSH/TCP) with the controller and a flow table. OpenFlow protocol (a messaging protocol) is a standard implementation of SDN controller and switch interactions, which defines the communication between an OpenFlow controller and the OpenFlow switch.

The basic principal of a logical OpenFlow switch architecture is the flow table. The data plane of an OpenFlow switch presents a clean flow table abstraction, which performs packet lookups and forwarding. The OpenFlow switch consists of several flow tables, and every one of them includes multiple flow entries. Each of the flow table stores flow entries consist of three main components as shown in Figure 2.7: (1) **Rules** containing a set of packet headers that specifies the flow; (2) **Action** an associated action that is implemented to specify the procedure of the packets such as modifying field, forwarding out a port, or to drop the packet; (3) **Stats** that update the statistics for a particular matching flow such as the number of dropped packets, duration of a flow entry that has been installed in the switch, the number of packets and bytes that have been received and transmitted per port, per flow table, and per flow table entry. When an OpenFlow switch receives an unknown packet in which it has no matching in the flow entries, the switch will encapsulate this packet and send to the controller. The controller then determines how to forward and process this packet. It can drop, update the packet, or add a new flow entry directing the switch on how to handle similar packets in the future [69], [33], [60].

A flow table may include a table-miss flow entry, which renders all Match Fields wildcards (every field is a match regardless of value) and has the lowest priority (priority 0). When defining a flow, there are variables taken into account called

Field	When applicable	Bits	Overview Description
Switch Port	All packets	16	Numerical representation of incoming port
VLAN ID	All packets of Ethernet type 0x8100	Last 12	0xffff means untagged
VLAN Priority	All packets of Ethernet type 0x8100	Last 3	VLAN PCP field
Ethernet Source address	All packets	48	L2 MAC source address
Ethernet Destination address	All packets	48	L2 MAC destination address
Ethernet Type	All packets on enabled ports	16	An OpenFlow switch needs to match the type in both standard Ethernet and 802.2 with SNAP header and OUI of 0x000000. The special value of 0x05FF is used to match all 802.3 packets without SNAP headers.
IP Source address	All IP and ARP packets	32	L3 Source IP address
IP Destination address	All IP and ARP packets	32	L3 Destination IP address
IP TOS bits	All IP packets	First 6	Specify as 8-bit value and place TOS upper 6 bits
IP Protocol	App IP and ARP packets	8	Only the lower 8 bits of ARP op-codes are used
TCP/UDP Source Ports	All TCP, UDP and ICMP packets	16	Only lower 8 bits used for ICMP Type
TCP/UDP Destination Ports	All TCP, UDP and ICMP packets	16	Only lower 8 bits used for ICMP codes

Table 2.2.: Details of Flow Headers (Twelve Tuples)

tuples. When a packet arrives at an OpenFlow switch, it needs to match a specific value out of these twelve tuples of the header field. As per the OpenFlow 1.0 spec, the match field components of a table entry consist of the following required 12-tuples header field listed in Table 2.2.

Therefore, network traffic that involves a variety of network services and protocols can use OpenFlow. It is also important to state that at the MAC/link layer, only Ethernet is supported. Unfortunately, OpenFlow in its current definition cannot manage Layer 2 traffic over wireless networks. The term *flow* can now offer a more precise definition. In regards to an individual switch, a flow can refer to a sequence of packets that matches a specific entry in a flow table. The definition is packet-oriented, in the sense that it is a function of the values of header fields of the packets that constitute the flow, and not a function of the path they follow through the network. A set of flow entries on multiple switches defines a flow that belongs to a specific path.

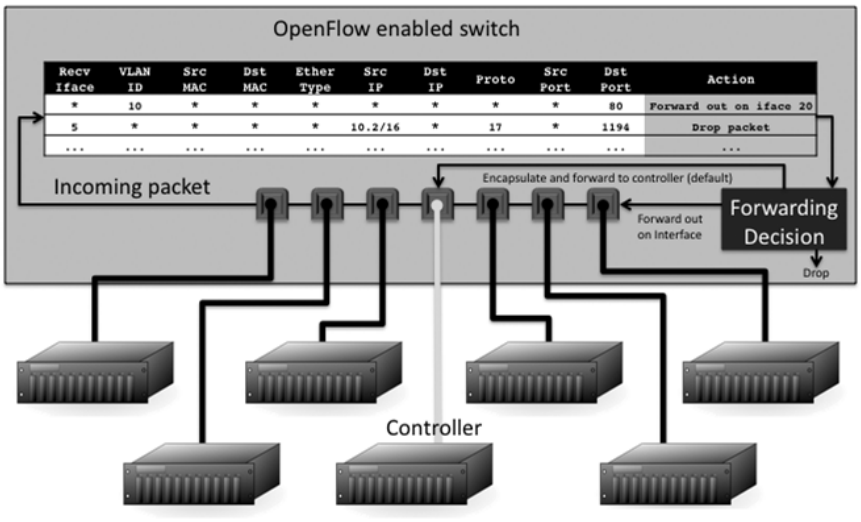


Figure 2.8.: OpenFlow-enabled Switch with Flow Entries [116]

For more flexibility, Figure 2.8 illustrates an example of a flow table entry. This can be seen in the OpenFlow switch, where one attached device acts as a controller. When a packet arrives, the header information is extracted and then matched with the header fields of the flow entries. The translation of a valid Flow Table entry will be "send all packets from any interface with VLAN ID 10 (taken from Ethernet Header) and destination port 80 (taken from TCP/UDP Header) out on interface 20" [116]. An OpenFlow switch can also cope with multiple flows, having the capability to handle up to 128k exact matches and up to 100 wild-card matches, combined with fairness throughput, latency and good performance [17].

There are a number of OpenFlow software switches available today that can be used to run an SDN test bed or when implementing new services and applications over SDN. In Table 2.3, a list of recent software switch implementations are shown. In addition, the OpenFlow standard version that the current implementation supports and a short description of the implementation language are also described.

It is readily apparent that the industry has a strong commitment to SDN due to the availability of commercial hardware switches that are OpenFlow-enabled. The OpenFlow standard is the main SDN-based technology that is currently being implemented in commodity networking hardware. Table 2.4 presents a list of native SDN switches currently available on the market with a brief description including the version of OpenFlow they support and some information regarding their model and manufacturer.

2.4.4. OpenFlow Controller

An OpenFlow controller is the brain of the protocol, making all of the high-level decisions per flow and installing those decisions to OpenFlow switches. These de-

Software Switch	Version	Language	Overview Description
OpenvSwitch [114]	v1.0	C/Python	This is a multi-layer and production quality virtual switch, which supports standard management interfaces and provides network automation through programmatic extension. Can be ported into ASIC switches
Pantou/OpenWRT [42]	v1.0	C	This turns a commercial wireless Access Point or router into an OpenFlow-enabled switch.
ofsoftswitch13 [75]	v1.3	C/C++	An OpenFlow 1.3 compatible user-space software switch implementation.
Indigo [100]	v1.0	C	Open source OpenFlow implementation that runs on physical switches and uses the hardware features of Ethernet switch ASICs to run OpenFlow.
LINC [6]	v.1.2/1.3.1	Erlang	Open source project led by FlowForwarding effort designed to use generally available commodity x86 hardware

Table 2.3.: Current Software Switch Implementations Compliant with the OpenFlow Standard

Manufacturer	Switch Model	OpenFlow Version
Hewlett-Packard	8200zl, 6600, 6200zl, 5400zl, and 3500/3500yl	v1.0
Brocade	NetIron CES 2000 Series	v1.0
Broadcom	BCM 56846	v1.0
IBM	RackSwitch G8264	v1.0
NEC	PF5240 PF5820	v1.0
Extreme Networks	Black Diamond 8K, Summit X440, X460, X480	v1.0
Pronto	3290 and 3780	v1.0
Juniper	Junos MX-Series	v1.0
Netgear	GSM 7352Sv2	v1.0
Pica8	P-3290, P-3295, P-3780 and P-3920	v1.2
Arista	7150, 7500 7050 series	v1.0

Table 2.4.: Current Available Hardware Switches by Markets, Compliant with the OpenFlow Standard

cisions are pushed to the flow table in the form of actions such as adding, deleting, and modifying a flow in the switch. The OpenFlow controller works similarly to the operating system, which offers a programmatic interface to the OpenFlow switches. By using this programmable interface, network applications (Net Apps) can be written to perform management and control tasks and provide new functionalities. An SDN controller of the control plane is considered logically centralised, and Net Apps are developed as if the network is a single domain.

The controller is responsible for maintaining an entire view of the network, providing policy decisions, and managing all the SDN nodes that comprise the network infrastructure, and offering a northbound API for services and applications. The network applications of an OpenFlow controller usually come with their own set of common application modules such as a learning switch, a router, a simple firewall and a basic load balancer [92]. An OpenFlow controller has several modules that provide its core functionality as well as network applications that are used to manage the network. The southbound API is used as an interface to connect with network devices on the data plane. In an SDN architecture, OpenFlow is the first and probably most well-known southbound interface because of its definition and standardisation. There are other de facto standards for the southbound-facing interface such as OF-CONFIG a companion protocol to OpenFlow [6], [97], Cisco CLI and SNMP, and OpenVswitch Database Management Protocol (OVSDB) by Nicira. However, they are limited to configuration rules. Unlike the controller-switch interface, there is currently no accepted standard for controller-to-applications interactions. This lack of a standard is one of the main drawbacks of SDN. Implementing a northbound API can provide collaboration opportunities between the open source community and vendors, which are likely to be implemented on an ad hoc basis for specific applications.

Currently, there are many existing SDN controller implementations available on the market. Table 2.5 presents a summary of current controller implementations including both open source SDN controllers and commercial SDN controllers. Unless stated otherwise, all the controllers in the table today support the OpenFlow protocol version 1.0. The table also gives a short overview description, language implementations, and the developers of the listed controllers.

The SDN-based controller used mainly for testing purposes in this research is the popular NOX OpenFlow controller. This choice is based on the fact that the NOX controller is fully open source and targeted largely at research and education, and the modules can be programmed to dictate the behaviour of the controller. In an OpenFlow system, NOX is a "Network Operating System" that acts as a central controller to provide control and visibility into a network of OpenFlow switches [116]. It is a platform for building network control applications written in either C++ or Python, which is loaded dynamically. NOX aims to provide abstraction layers for the network infrastructures (*e.g.*, the routers and switches), giving network applications an execution environment, while developing similar management concepts and multiplexing between them [38]. Gude et. al. (2008) give an overview of NOX by describing its main components, observation and control granularity,

Controller	Open Source	Language	Developer	Overview Description
NOX [47]	Yes	Python/C++	Nicira	The first OpenFlow controller in Python and C++ implementations.
POX [78]	Yes	Python	Nicira	A general open-source SDN controller written in Python.
Floodlight [41]	Yes	Java	BigSwitch	An OpenFlow controller written in Java, based on the Beacon implementation (supports v1.3) that works with physical and virtual OpenFlow switches
OpenDaylight [61]	Yes	Java/ C++	Linux Foundation	Open platform for network programmability to enable SDN and NFV for networks at any size and scale a combination of components including a fully pluggable controller, interfaces, protocol plug-ins and applications.
Beacon [15]	Yes	Java	Stanford	A cross-platform, modular OpenFlow controller written in Java, which supports threaded and event-based operations [36].
Maestro [22]	Yes	Java	Rice University	A Java-based network operating system that provides interfaces for implementing modular network applications and for them to access network state for control and modifications.
Ryu [89]	Yes	Python	NTT, OSRG group	An SDN controller that provides a logically centralised control and APIs to develop new network control applications. Ryu fully supports OpenFlow v1.0, v1.2, v1.3 as well as the Nicira Extensions.
Trema [76]	Yes	Ruby/C	NEC	A framework for implementing OpenFlow controllers based on Ruby and C.
SNAC [98]	No	C++	Nicira	An OpenFlow controller based on NOX-0.4, which uses a web-based and user-friendly policy manager for network management, device configuration and events monitoring.
MUL [67]	Yes	C	Kulcloud	An OpenFlow controller that has a C-based multi-threaded infrastructure at its core, which supports a multi-level north-bound interface for application development.
Helios [21]	No	C	NEC	An extensible OpenFlow controller written in C that supports a programmatic shell for performing integrated experiments.
Jaxon [13]	Yes	Java	Independent Developers	An OpenFlow controller written in Java based on NOX implementations.
NodeFlow [72]	Yes	Java Script	Independent Developers	An OpenFlow controller written in JavaScript for Node.JS [73].
Flowvisor [94]	Yes	C	Stanford/ Nicira	Special purpose controller implementation.
RouteFlow [70]	Yes	C++	CPqD	Special purpose controller implementation.
ovs-controller [114]	Yes	C	Independent Developers	A basic OpenFlow controller reference implementation with OpenVswitch for managing any number of remote switches using OpenFlow protocol. As a result, the switches function as L2 MAC-learning switches or hubs.

Table 2.5.: Current SDN Controller Implementations Compliant with the OpenFlow Standard

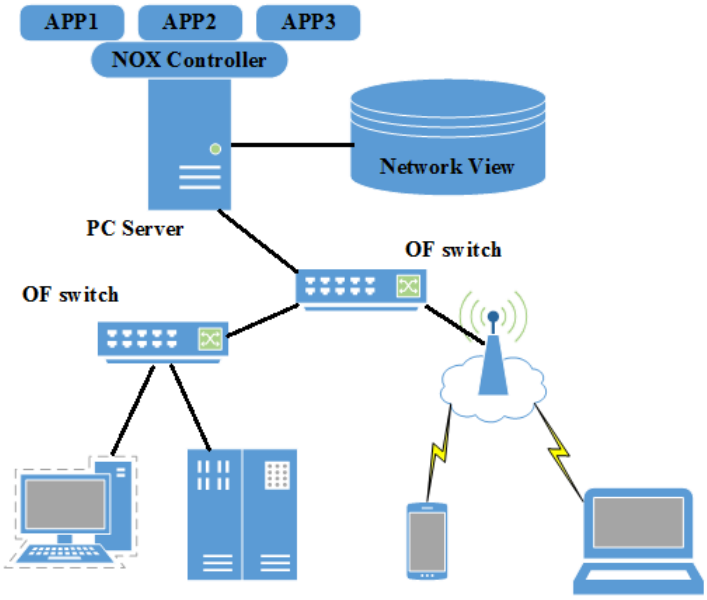


Figure 2.9.: Components of a NOX-based network: OpenFlow (OF) switches, a server running a NOX controller process, and a centralised database containing the network view [47].

switch abstraction, basic operation, scaling, and implementation status [47].

Components – The main components of an NOX-based network are visualised in Figure 2.9, which consists of a set of switches and one or more network attached servers. These servers are run by the NOX software and its management applications. The NOX software involves several different controller processes and a single network view. The network view is a centralised database containing the results of NOX's network observations such as network topology, abstractions like users and host names, and low-level information like MAC and IP addresses. Applications use this state information to make management decisions. In order to control network traffic, NOX instructs the OpenFlow switch to forward all packets from certain protocols such as Dynamic Host Configuration Protocol (DHCP) and Domain Name System (DNS) to its controller. Then, this controller processes the packets to filter out information regarding a topology change in which it updates the database [47], [116].

Granularity – This is an important design issue in which NOX can provide observation and control. NOX's observation granularity can provide adequate information for many network management tasks, as well as changing slowly enough that it can be maintained in sizable networks in terms of scalability. The intermediate granularity for NOX is *flows*. Once the control is exerted on some packets, subsequent packets that have the same header portion are treated in the same way. In regards to this flow-based granularity, it is possible to develop a large-scale system while still giving flexible control.

Switch Abstraction – NOX uses management applications to control network traffic by passing the instructions to an OpenFlow switch. These switches should support flow-level control granularity mentioned above, and also should be independent of the particular switch hardware. To meet the requirements, the OpenFlow switches are presented by flow tables containing flow entries with the form:

< Match Fields: Counters, Instructions>

Each packet matches a specified header, the counters are updated, and an appropriate instruction is taken. If a packet is matched to multiple flow entries, the entry with the highest priority will be chosen.

Operation – NOX is primarily responsible for handling unknown packets and often the first packet of a flow (flow-initiations). NOX controller processes may also choose to receive all the packets from certain protocols like DNS and Link Layer Discovery Protocol (LLDP), and therefore will not insert a flow entry for them. The NOX controller uses these flow-initiations as well as other forwarded packets to create and update the network view, and determine whether to forward the packets, and if so, along which path (control). Gude et. al. (2010) have developed routing applications and access-control to determine whether a flow should be allowed, process an appropriate L2 route, and install flow entries in all OpenFlow switches.

Scaling – NOX can process at three different rates in terms of timescales: (1) Packet arrivals - scale of millions of packets per second for a 10 Gbps link, (2) Flow initiations - typically one or more orders of magnitude less than the packet arrival rate, (3) Changes in the network view - scale of tens of events per second for a network of thousands of nodes. In regards to consistency, the network view is the only global application state information. This is because applications draw information from the network view to make control decisions such as policies to be enforced in the network. Gude et. al. (2010) claim that a single NOX controller running on a generic PC can handle up to 100,000 flow-initiations per second, which is sufficient to accommodate large campus networks.

Implementation status – The NOX controller runs in user-space on the network servers. At present, NOX has two separate lines of development. The "NOX-Classic" line is written in Python and C++ with many applications, whereas the "new NOX" only contains support for C++. For those who prefer Python, they can either stick with the classic NOX or might be interested in POX, a younger sibling of NOX using Python. However, developers have no current plan to do any other significant development on the classic NOX. The NOX's core infrastructure and speed-critical functions are implemented in C++ (around 32,000 lines) [47].

Werner (2010) separates the NOX's interaction with network traffic into a three layer model. The highest layer is the changes in the network view. It is determined by the controllers, and by means of a centralised database, propagates to the entire system. A topology change is considered as an expensive operation in terms of overhead, but can still be manageable due to their rarity. The middle layer is

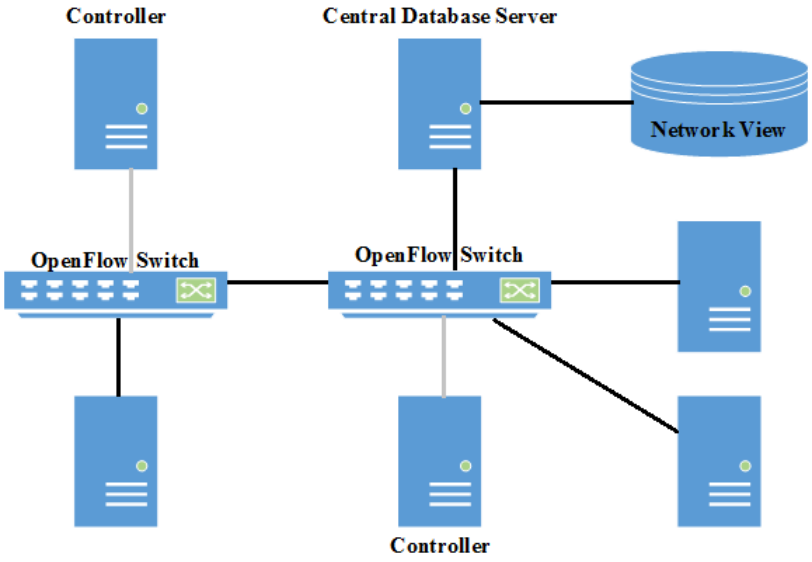


Figure 2.10.: Example NOX-based network setup. Each switch has its own controller but network state is stored centrally [116].

termed flow-initiations. A flow is typically a single end-to-end application transmission, which is identified by a flow table entry. The first packet of a flow is always forwarded to the controller. The controller then makes management decisions on how to process that flow. When a path is specified, NOX inserts appropriate entries in the flow tables of associated switches, so that similar packets for this specific flow can be processed by switches themselves without having to bother the controller again [99]. The lowest layer of the NOX model represents all packets within the network. They are typically regarded as part of a configured flow, so the processing occurs entirely inside the switches without any interactions from the controller. The traffic is processed by fast, hardware-accelerated devices, with only a small fraction that constitute flow-initiations being handled by the NOX software. Figure 2.10 illustrates a possible small NOX-based network set up.

2.4.5. OpenFlow Channel and Protocol

The connection between the controller and the OpenFlow switch is through an interface of OpenFlow channel. Through this interface, the controller exchanges messages with the switch in order to configure and maintain the switch, receive events from the switch, and finally send packets out through the switch. The interface is implementation-specific between the data path and OpenFlow channel, but all messages exchanged between the controller and the switch must follow the format indicated by the OpenFlow protocol. Transport Layer Security (TLS) is the cryptographic protocol usually used to encrypt the OpenFlow channel; however it may be run directly over TCP.

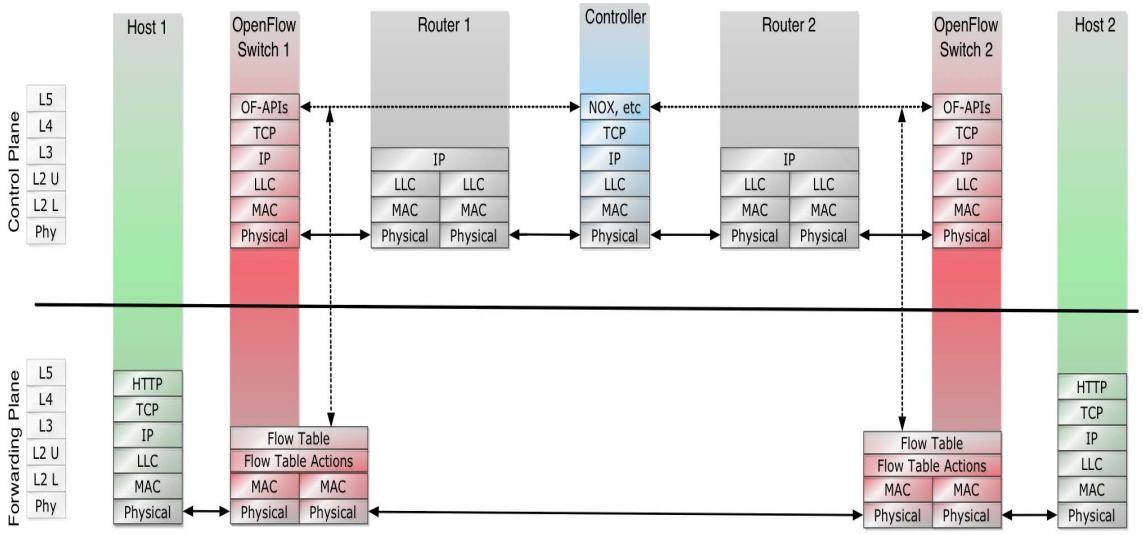


Figure 2.11.: Mapping of OpenFlow Network Protocol Layers

The detailed requirements of an OpenFlow switch are defined in the OpenFlow switch specification, published by the ONF. It is important to note that the OpenFlow protocol in this thesis is describe based on the OpenFlow specification, Version 1.0 (Wire Protocol 0x01), December 31, 2009. This original specification 1.0 was developed at Stanford University and was widely implemented. OpenFlow 1.2 was the first release from ONF after inheriting the project from Stanford University. OpenFlow 1.3 significantly expands the functions of the specification and is likely to become the stable base upon which future commercial implementations for OpenFlow will be built. However, fine-grained security settings in regards to scenarios with multiple OpenFlow controllers are outside the scope of the present specification.

Figure 2.11 shows the mapping of OpenFlow control plane and forwarding plane with network protocol layers. The NOX controller, for example, can be seen as an operating system performing at Layer 5, which provides a northbound API (*e.g.*, RESTful, Java and Python API) for network applications. On the other hand, OpenFlow is considered as a standardised southbound API used for the communications between the controller and switches. In the forwarding plane, an OpenFlow switch can provide header matching rules from Layer 1 (Ingress port), Layer 2 (L2 switching), Layer 3 (L3 routing) and up to Layer 4 (L4 firewalling). For example, it can match the header fields with MAC source/destination address, IP source/destination address, and TCP/UDP source and destination port.

The OpenFlow protocol is defined into three message types including Controller-to-switch, Symmetric, and Asynchronous, each with multiple sub-types.

2.4.5.1. Controller-to-Switch

Controller-to-switch messages are initiated by the controller and used to directly manage or inspect the state of the switch. These types of messages may or may not need a response from the switch and are categorised as follows:

Features: Once the TLS session is established, the controller sends a features request to request the capabilities of a switch. The switch must reply with a features reply message specifying the capabilities and the features that are supported by the switch.

Configuration: The controller is able to set and query configuration parameters in the switch. The switch only responds to a query from the controller.

Modify-State: These messages are sent by the controller to manage state on the switches. Their main purpose is to add, delete and modify flow table entries in the OpenFlow tables as well as to set switch port properties.

Read-State: The controller uses Read-State messages to collect various information from the switch flow tables, ports and each flow entries such as current configuration, statistics and capabilities.

Packet-out: The controller uses this type of message to send packets out of a specified port on the switch. Packet-out messages must contain the whole packet or a buffer ID referencing a packet stored in the switch. Also, the message must contain a list of actions to be applied in the order they are specified.

Barrier: Barrier request/reply messages are used by the controller to assure message dependencies have been met or to receive notifications for completed operations.

2.4.5.2. Symmetric

Symmetric messages are initiated by either the switch or the controller and sent without solicitation in either direction. There are three symmetric message types in the OpenFlow protocol as following:

Hello: Upon connection start up, Hello messages are exchanged between the controller and the switch.

Echo: Either the switch or the controller can send Echo request/reply messages, and must return an echo reply. These messages are primarily used to verify the liveliness of a controller-switch connection, and are also used to measure its latency and bandwidth.

Vendor: Vendor messages provide a standard way for OpenFlow switches to offer additional functionality within the OpenFlow message type space. It is the staging point for features intended for future OpenFlow revisions.

2.4.5.3. Asynchronous

Asynchronous messages are initiated by the switch and used to inform the controller of network events and changes to the switch state. These messages can be sent without a controller soliciting them from a switch. Asynchronous messages are sent by the switch to controllers to denote a packet arrival, changing switch state or an error. The four main asynchronous messages are described as follows:

Packet-in: A packet-in message is sent to the controller for all packets that do not match a flow entry or if a packet matches a flow entry with an associated 'send to controller' action. For switches that have enough memory to buffer packets that are sent to the controller, the packet-in message will include just a fraction of the packet header (default of 128 bytes) and a buffer ID of the stored packet to be used by the controller when it is ready for the switch to forward the packet. In the case of switches that run out of buffer space or do not support internal buffering, the packet-in message must contain the encapsulated complete packet to be sent to the controller.

Flow-Removal: The primary purpose of this message is to inform the controller regarding the removal of a flow entry from a flow table. A Flow-Removal message is generated as the result of a controller flow delete request or the switch flow expiry process when one of the flow timeouts is exceeded. When a flow entry is added, an *idle timeout* is configured to indicate when a flow entry should be removed due to the lack of activity, whereas a *hard timeout* is also configured to specify when a flow entry should be removed, regardless of activity.

Port-status: The switch uses this type of message to inform the controller of a change in a port. The switch is expected to send port-status messages to controllers in case of port configuration or port state changes. These events include changes in port configuration events such as port disabled directly by a user, or a change in the port status events, for example if the link goes down.

Error: The switch is able to notify controllers of problems using error messages.

2.4.6. SDN Development Tools

SDN has been introduced to facilitate network innovation and evolution by allowing rapid deployment of new protocols and services. Currently, not many network simulators support the OpenFlow protocols. This subsection provides an overview of currently available emulation and simulation environments for developing SDN-based services and protocols. There are several recent simulation tools for developing and testing the OpenFlow protocol and applications such as OpenFlowVMS, NS-3, Mininet, OMNeT++, and EstiNet. This subsection also explains the efforts related to these tools.

Simulation tools can provide more suitable task of designing, building, and testing for users with practical feedback when developing real-world systems. This will

allow system designers to determine the correctness and efficiency of a design before the system is actually deployed. It is also useful to explore the behaviour of these protocol models, capabilities and shortcomings further, by making use of simulations [82].

OMNeT++ is a discrete event network simulator based on C++. The primary goal of OMNeT++ is simplifying the integration of new modules as well as changing those already implemented [110], [111]. The OMNeT++ environment has the INET framework, which is an open-source communication networks simulation package. This framework contains many models for wired and wireless networking protocols such as UDP, TCP, SCTP, IP, IPv6, Ethernet, and several application models [109].

A variety of new routing protocols focused particularly on the distributed environment have been developed, but little performance information and no realistic performance comparisons between them are available. Until now, very few performance evaluations of OpenFlow architectures using OMNeT++ exist. Recently, the INET framework has a new extension for the OpenFlow model, which is a new toolbox for the SDN simulation environment. However, this OpenFlow packet-level simulator is still in the early development of the INET framework, currently based on switch specification version 1.2 [54]. The implementation model uses the *openflow.h* header file to develop the protocol and defined messages as closely as possible. The nodes implemented include the OpenFlow switch, OpenFlow controller and the most important messages use to communicate between switch and controller via secure OpenFlow channel [10]. In addition, utility modules were also implemented to enhance required functions like controller placement and spanning tree modules.

A prior study that aims to simulate OpenFlow networks was OpenVMS started in 2009 [118]. This work was designed to emulate OpenFlow enabled devices based on using virtual machines. However, virtual machines have significant limitations in regards to large resource overhead and thus do not scale very well. Moreover, it was developed to test real-time functionalities of OpenFlow rather than simulating and evaluating arbitrary scenarios [54]. This also applies to Mininet, in the sense that Mininet is an emulation platform for the functional testing of OpenFlow protocols and applications. It uses lightweight OS containers to separate and emulate hosts and switches in a network, and hence reduces the overhead.

The widely used NS-3 simulator has a project for OpenFlow protocol simulation as well. However, this approach only supports OpenFlow specification version 0.89, which is quite old compared to the latest version 1.4 [115], [113]. Furthermore, the NS-3 simulator is a user-level program, the same as the real OpenFlow controllers like NOX/POX. Therefore, the real OpenFlow controller program cannot be compiled and linked together with the NS-3 program to form a single executable program [115]. For example, it is compulsory to create C++ code from scratch to build new modules of OpenFlow switches or controller. Consequently, a real OpenFlow controller cannot be readily run in NS-3 simulations without code modification. According to the NS-3 official website, developers found it too difficult to upgrade NS-3

	NS-3	**OMNeT++**	**Mininet**	**EstiNet**
OpenFlow version	0.8.9	1.2.0	1.0.0	1.0.0/1.1.0
Programming language	C++	C++	Python	C/C++
Operating system support	GNU/Linux, FreeBSD, OS X	OS X, Windows, Linux distributions	OS X, Windows, Linux distributions (VM image)	Linux Fedora 17 32-bit
Supporting simulation	Yes	Yes	No	Yes
Supporting emulation	No	No	Yes	Yes
Ability to use real controller	No	No	Yes	Yes
Result repeatable	Yes	Yes	No	Yes
Scalability	High By single Process	Middle By single Process	Middle By multiple Processes	High By single Process
Performance result correctness	No Spanning Tree Protocol and no real controller	No real controller	Performance depend on resources	Yes
GUI support	Yes -Monitoring Only -Configuration by C++	Yes - Configuration - Monitoring	Yes -Monitoring Only -Configuration by Python	Yes - Configuration - Monitoring

Table 2.6.: A Comparison of NS-3, OMNeT++, Mininet and EstiNet [115]

for OpenFlow version 1.0, and NS-3 still cannot support a real external OpenFlow controller.

Other efforts like EstiNet combine the advantages of both the network simulator and emulator without their respective limitations. As for emulator, EstiNet uses a real controller run on real devices and applications that can control simulated OpenFlow switches without any modification. EstiNet uses a new method of kernel re-entering to support multiple hosts in a single kernel [115]. Regarding scalability, the EstiNet simulation engine has the capability to simulate a large number of OpenFlow switches. Moreover, as EstiNet simulation engine can generate time-related OpenFlow performance measurements accurately, the results are repeatable [81]. The EstiNet GUI can also show the playback of OpenFlow control packets once the simulation is finished. Table 2.6 shows a comparison of the capabilities provided by NS-3, OMNeT++, Mininet and EstiNet with respect to their latest developments.

Mininet needs to run a shell process to emulate virtual hosts and start up a user-space or kernel-space (OpenVswitch) to emulate each OpenFlow switch. Therefore, Mininet is less scalable compared to EstiNet, NS-3 and OMNeT++. Mininet can

only be used to study the behaviour of virtual hosts, but cannot be used to study the time of network/application performance. Mininet's GUI can be used for observation purposes such as observing the packet playback of a simulation run, and users need to write Python scripts to set up and configure the emulation case. In contrast, OMNeT++ has a GUI which can be used for observation of results, where users need to write C++ code to set up and configure the simulation case. Overall, it is better to use OMNeT++ even if it takes time and effort to create simulations, though once modules are created, it is much easier to create new ones [57], [11], [115], [113].

2.5. SDN Scalability Issues

The software-defined networking field has been quite active in the past few years and is growing at a rapid pace. Still, there are key development and research challenges that need to be addressed. One of the major limitations of a centralised OpenFlow controller is the lack of scalability. The fundamental feature of an SDN controller is that it is responsible for establishing every flow in the network. As the size of production networks deploying OpenFlow increases, so will the number of flows that need to be processed. If the controller does not have the capacity to handle all these flow setups, it can present a scalability bottleneck.

In regards to enterprise data center networks which have 100 edge switches, the controller can expect to see around 10 million flow requests per second [22]. This can create significant challenge for deploying centralised OpenFlow controller in a large-scale data center. Cai (2010) further notes that each flow request is processed individually, and all packets created accordingly are forward individually by the OpenFlow controller. In addition, sending out messages individually takes about 80% of the flow request processing time. This can cause overhead of multiple socket write operations to forward each packet to the same destination individually instead of a single batch. The NOX controller, for example, did not give the application developer enough flexibility to achieve scalability, nor adequately address reliability as the control platform must handle equipment (and other) failures gracefully [55].

A single domain can also lead to scalability issues where the computing power is loaded to a single point, whereas the distance can affect the higher Round Trip Time (RTT) for requesting unknown packets to be processed. Tootoonchian and Ganjali (2010) also explain that relying heavily on only one centralised controller for the entire network might not be feasible for several reasons. Firstly, the number of control traffic destined towards the controller increases according to the number of switches. Secondly, despite where the controller is placed, if the networks have a large diameter, some switches will face long flow setup latencies. Thirdly, since the network is bounded by the processing capacity of the controller, flow setup times can rapidly increase as demand grows in terms of network size and complexity [106].

As a result, improving the performance of the SDN controller to keep up with the rising demand becomes a significant challenge.

On the other hand, Foster et. al. (2010) highlight the difficulties of writing programs for the OpenFlow-based NOX platform as follows [43]:

• Interactions between concurrent modules – Networks often process multiple tasks such as routing, monitoring, and access control. These functions cannot be processed independently unless they perform on non-overlapping portions of the traffic since a rule (un)installed by one module could undermine the proper functioning of other modules

• Low-level interface to switch hardware – OpenFlow maintains a low-level interface to the switches. Applications must establish rules that match on bits in the packet header. Because rules can have wild-cards, a packet may match different overlapping rules with multiple priorities. This may translate high-level policy into multiple low-level rules.

• Two-tiered programming model – Controllers only receives packets that switches do not know how to process. This can limit the controller's visibility into the underlying traffic. Essentially, the execution of an application is split between the controller and the switches. Applications must avoid installing rules that hide vital information from the controller.

Early SDN deployments are managed by a centralised domain, which often leads to a single point of failure. Many solutions can be applied to address this issue such as control plane architectures with multiple controllers connected to a switch, which could allow backup controllers to take over in the event of a failure [39]. Other approaches have been made to physically distribute controllers but this is logically centralised by synchronising all network information and balancing the load between several controllers (can distribute network applications, or just synchronise events between distributed controllers). These proposals provide a simplified central view of the network as well as reduce the lookup overhead by allowing communication with local controllers. However, many research projects are still in their research phase where such solutions produce more complexity on the current OpenFlow architecture.

Furthermore, SDN can simplify network management and also lead to more flexible and faster operations. It depends on abstracting the control functions as network applications. In addition, the controller has a wide view of the entire network. The SDN architecture has the control layer and the forwarding layer, this architecture relies on a central point, which manages the forwarding layer. Hence, network bottleneck can occur at both interfaces, in the controller as well as the switches. For example, traffic bottleneck within switches can happen in the forwarding table memory. Thus, the current SDN architecture is not scalable for large and extended networks, which include all the networks used in transferring data from distant places and in a wide geographical area (several kilometres to thousands of kilometres). Any such links between network devices in places far away from each other

need to connect the branches of their institution within or outside the country to each other in order for the users to exchange information and e-mail.

The bottleneck of request and response times is another limitation of OpenFlow. Conventional networks are able to handle events locally and perform actions according to those events, whereas in an SDN architecture, the forwarding plane of each device is not able to handle some events that need to be sent to the controller for managing those events. The controller receives these events and calculates them, and finally sends the command back to the corresponding switch. In this case, sending many requests from a large number of switches to a single controller could lead to a traffic bottleneck. Also, the same thing can happen if many commands of setting many flow rules are sent to one certain switch.

For an OpenFlow switch, the requested queues are processed by scheduling algorithms that allow the provisioning of different Quality of Service (QoS) levels for different types of packets. However, enforcing Quality of Service in OpenFlow protocol, particularly version 1.0, is quite difficult. This version only provides very basic and not sufficient support for QoS. This is because the main focus of OpenFlow is based on static configurations.

2.6. Related Work to Solve OpenFlow Scalability Issues

In the last few years, network technologies have been increasing significantly in performance, complexity and functionality, driven by the needs of the modern world. This has given rise to a new network paradigm called Software-Defined Networking (SDN). SDN separates the forwarding plane from the control plane, and in so doing enables the creation of a standardised programming interface [44]. Flow computation is done in a centralised controller with the switches only performing simple forwarding functions. OpenFlow is very efficient at moving the computational load away from the forwarding plane and into a centralised controller. This centralisation brings optimality but creates additional problems of its own including single-domain restriction, scalability, robustness, and the ability for switches to act autonomously.

There are three approaches attempted to resolve the limitations of OpenFlow such as optimising the centralised controller, empowering the switches to process some control functions, and distributing the control platform [22]. However, the first approach still relies on only a single controller and sends batching messages to the switches not in a run-time configuration. This can significantly affect the packet forwarding process within large-scaled networks. Furthermore, OpenFlow is more flexible since its control logic can realise behaviours that cannot be easily achieved by a set of policy rules installed in authorised switches [34], [63], [120]. Finally, synchronisation within distributed controllers should be in a run-time mode in order to provide reliability and robustness [106], [95].

2.6.1. Optimisation Techniques

A large number of OpenFlow applications have been developed to solve the limitations of an OpenFlow control plane. Maestro is a multi-threaded Java-based controller from Rice University designed for scalable OpenFlow control. Maestro has a central controller for a flow-based routing network with an increasing of flow processing, and thus can enhance network scalability [37]. Maestro can achieve overall good performance and scalability by coordinating between centralised controllers and distributed routing protocols. The Maestro approach works as a hybrid control plane, which is more robust than the centralised control plane. It keeps the simple programming model for programmers and uses the technique of parallelism with additional throughput optimisation techniques to alleviate packet processing. This includes the use of multi-threading to handle the flow requests from OpenFlow switches and batches sending when the controller needs to send configuration control messages back to the switches [107]. Maestro can solve traffic bottlenecks for SDN-based controllers by using several applications, such as "routing" or "learning switch" [22].

Maestro has a programmable environment with a high-level language, which can deal with distribution and concurrency without the need to involve the operator. Maestro has a common user interface to control the hardware, and it also includes analysis tools. It can auto-detect the attached hardware including its features. Moreover, Maestro can make local decisions without involving a controller, and can synchronise certain actions by using its master machine [22], [5]. The operator is not involved in the controller configuration, where Maestro is able to insert events automatically. As a result, Maestro has been accepted as a high-level program application for protected structures [52]. However, Maestro still relies on only a single controller and sends batching messages to the switches not in run time configuration. These challenges can significantly affect the packet forwarding process in OpenFlow.

2.6.2. Devolving Some Control Functions Back to the Switches

- **DevoFlow** (Devolved OpenFlow) decreases the interactions between OpenFlow switches and controller by allowing an aggressive use of wild-carded OpenFlow rules and new mechanisms to detect significant flows. DevoFlow allows switches to make their routing decisions locally and forward flows that do not need the vetting by the controller. DevoFlow modifies the Open Flow model to redistribute as many decisions as possible to the switches, in ways to enable simple and cost-effective hardware implementations [34]. DevoFlow can solve the bottleneck of an OpenFlow switch within high-performance networks, where unknown packets are usually forwarded to the controller. Assuming there are several thousand flows being forwarded per second, DevoFlow proposes to tackle the problem by addressing short-lived (mice) and long-lived

(elephant) flow separately. Switches only need to notify the controller of specific flows, which require more security or any other policies [63].

DevoFlow is forcing the switches to use OpenFlow wild-carded rules in order to reduce the interactions between the controller and the switch. DevoFlow uses mechanisms that will allow the switches to make local decisions for routing when they do not require per-flow checking by the controller. DevoFlow involves the OpenFlow controller for computation power but puts too much load on the control plane. Therefore, forcing the wild-cards flow matching rule can reduce that load, but will prevent the controller from managing the traffic effectively [34], [63].

- **DIFANE** [120] provides a way of achieving effective rule-based policy enforcement in the network and aims to resolve the centralised issue by distributing the rules across *authority switches* and performing policy matching rules at the switches themselves. DIFANE is a distributed flow-based architecture built on OpenFlow switches. The division of labour is changed by DIFANE between the centralised management system and the switches, by pulling some rule processing functions back to the switches, to achieve better scalability. DIFANE can solve the bottleneck of the current centralised OpenFlow controller by distributing some of the controller functionalities across authority switches. This authority can handle unknown packets received from other switches instead of sending the packets to the controller [120].

  DIFANE downgrades simpler tasks from the controller by translating high-level policies to low-level rules, and distributing the rules and processing all the packets in the switches. DIFANE reduces the memory usage for rules at the switches. Therefore, DIFANE builds a distributed rule directory service among the switches by partitioning the rules between switches. DIFANE can be easily implemented with smaller software "medications" to commercial OpenFlow switches [64].

DIFANE architecture is composed of a controller and a number of authorised switches. The controller provides rules and installs them in these authority switches. The authority switches will receive the packets and forward them according to the rules or encapsulate them and send them to other authority switches. Minlan Yu (2011) indicates that DIFANE has three sets of wild-card rules, which can keep packet processing within the forwarding plane instead of sending to the controller. Wild-card rules include cache rules, authority rules and partition rules as shown in Table 2.7.

However, devolving control functions back to the switches is not easy; it requires the deployment of a set of relativity rules and configurations installed in OpenFlow switches in terms of security perspectives.

Wild-card Rules	Description
Cache rules	The rules are cached by the ingress switches so that most data traffic stays in the cache and is processed by the ingress switches. Authority switches install the cache rules in the network.
Authority rules	The controller can update and install the authority rules in all authority switches. When a packet matches this rule, it triggers a control plane function to install rules in the ingress switch.
Partition rules	Partition rules are installed in each switch by the controller to ensure a packet will always match minimum one rule in the switch and stay in the forwarding plane.

Table 2.7.: DIFANE Wild-Card Rules

2.6.3. Designing a Distributed Control Platform

- **FlowVisor** [93] is a similar design as HyperFlow to distribute the OpenFlow control plane. However, FlowVisor tackles a slightly different problem as it enables multiple controllers in an OpenFlow environment by slicing network resources and delegates the control of each slice to a single controller and also supports isolation between slices. FlowVisor is referred to as one of the first approaches to promote a network virtualisation architecture for infrastructures and resembles the hypervisor model for storage and computation. This will allow multiple controllers to manage overlapping sets of hardware switches simultaneously [94].

  FlowVisor is an early development to visualised SDN, which allows different networks to share similar OpenFlow network infrastructure. For this purpose, FlowVisor offers an abstraction layer that can easily slice a data plane of an OpenFlow-enabled switch. This will provide multiple and diverse networks to co-exist. FlowVisor has five primary slicing dimensions including device CPU, topology, bandwidth, traffic, and forwarding tables with each network slice supporting multiple controllers that can co-exist on top of the same physical network infrastructure. Each controller is allowed to act only on its own network slice [95]. In general terms, a slice is defined as a particular set of flows on the data plane. From a system design perspective, FlowVisor is a transparent proxy that intercepts OpenFlow messages between switches and controllers. It partitions the link bandwidth and flow tables of each switch. Each slice receives a minimum data rate, and each guest controller gets its own virtual flow table in the switches [93], [94].

- **HyperFlow** [106] aims to improve the performance of the OpenFlow control plane by extending the NOX and distributing the event-based control plane for OpenFlow. By sharing the same consistent network-wide state among dis-

tributed controllers, HyperFlow assures that the process of a particular flow request is localisable to an individual NOX controller. It proposes a physically distributed control plane, but it is logically centralised by sharing network information and balancing the load between many controllers. HyperFlow seeks to have multiple controllers to manage the entire network, and each controller is responsible for its portion of the network [106]. HyperFlow is an application implemented on top of the NOX controller, where the implementation is changed operations, and allows reuse of existing NOX applications with minor modifications [47].

Also, each switch makes a local decision (relying on its flow table and its controller) using the HyperFlow to synchronise passively upon the whole network state of OpenFlow controllers. This can provide a local server by the controller to all packet flows, and thus significantly reduce the response time of the control plane for data plane requests. With HyperFlow, each controller synchronises its global network view using a distributed file system. It uses the Publish/Subscriber messaging communication to support cross-controller communication [106]. Therefore, HyperFlow is capable of processing more flow events while keeping the flow setup latency minimal [69].

- **Onix** [55] approach attempts to maintain a reliable control platform by having a logically centralised controller but physically distributed control plane. Onix aims to cope with large-scale networks using a distributed management architecture. It provides a useful common programming API to build network applications. Onix contents are distributed to high-level network services and applications to ease distributed data coordination. Onix deals with per-packet registers instead of per-packet events to reduce the interactions between Onix and the network forwarding elements. Onix was developed to resolve SDN scalability purposes, which has an in-memory network graph of entities (network state) called Network Information Base (NIB). The NIB contains two duplicated data storages for exchanging information within the NIB.

  Control applications in Onix can also divide the workload and adding instances to reduce work without only replicating it. In Onix, network state is managed by a cluster of Onix nodes, which appears as a single node in a separate cluster's NIB. This can enhance the hierarchical and federated structuring of Onix clusters, thus decreasing the overall amount of information needed within a single Onix cluster. Moreover, Onix provides applications with control over the durability and consistency of the network view.

  Onix also supports OpenFlow protocol, where the controller indirectly modifies the NIB in the data plane. The controller has to guarantee that changes in the data plane are reflecting the NIB before the NIB can make any changes in the data plane configuration. Each Onix can manage a subset of a certain data plane independently as well as exposing the entire network state using NIBs. Thus, Onix can scale large networks and provide flexibility for production

deployments.

- **Kandoo** [48] proposes a hierarchical distributed control platform by defining two main layers of control. The top layer contains a centralised root controller, which is responsible for maintaining the desired network behaviour as well as running non-local applications. The bottom layer comprises several local controllers to manage local applications that have no knowledge of the network wide state. Consequently, Kandoo can decrease the number of events received at the control plane of the network.

  Kandoo [48] is a hybrid implementation that uses local controllers with local applications for network management. Kandoo will redirect a flow to the root controller only if the decisions require a centralised network state. By filtering the number of new flow requests, Kandoo can reduce the traffic on the root controller. It can also provide the data plane with a faster response for flow requests that are handled by a local controller application.

All these approaches are physically distributed but logically centralised. They propose a different architecture and layer of SDN. Multiple controllers managing a network can bring many benefits such as enabling backup controllers to take over in the case of a failure, simplifying central view of the network, and reducing the look-up overhead by allowing communication with local controllers. However, distributing the control platform in FlowVisor, HyperFlow, Onix and Kandoo are not adapted to large data centres with several Autonomous Systems (AS), and they need extensive traffic among controllers to maintain a global network view [16]. Moreover, there is also a potential downside related to trade-offs between staleness and consistency when distributing network state among the controllers. This may cause applications that believe they have an accurate view of the network to operate incorrectly [58].

2.7. Autonomic Communications

2.7.1. Background of Autonomic Communications

Over the past twenty years, network technology has been improved rapidly in terms of speed, performance, component and functionalities. A number of different types of network devices have been developed and hence have led to an increase in the complexity of network systems. Autonomic communications (ACs) is a concept for managing the ever-growing complexity of distributed networks; this term gives systems the ability of self-management, which mean each component in ACs can adapt itself to changing conditions of the dynamic environment. ACs can refer to “distributed communication systems with the learning and adaptation capability to cope well with dynamic, uncertain and complex environments – that is, immediately adapt their strategies in accordance with high-level business objectives and rules in order

to maximise service satisfaction within available services and managed resources" [27]. ACs are systems that can create their own strategies for adaptation according to the system's objectives to meet service satisfaction within managed resources. It is currently in the area of considerable research and industrial interest. ACs adopt the analogy of human autonomic nervous system, which regulates homeostatic functions without conscious intelligent control, and seeks to simplify the management of complex communication systems and reduce the needs for human intervention [35]. It deals with a number of existing disciplines. For example, protocol design, pervasive computing, network management, artificial intelligence, biology, semantics, context-aware systems, and sensor networks.

ACs aim to provide systems with the ability of self-management, which means each component in ACs can adapt itself to changing conditions of the dynamic environment. ACs can also free networks managers from some management tasks especially low-level tasks, and in the meantime bring better system behaviour. Moreover, ACs are considered as a universal concept of self-x management properties such as self-configuration, self-protection, self-healing, and self-adaptation. These self-x functionalities are the key characteristics of autonomic communication described in the IBM blueprint. Because of the increasing difficulties of managing complex networks, ACs are playing a major role and are also regarded as a promising solution to address the challenges in such distributed electronic environments.

The concept of ACs was built on the employment and extension of Autonomic Computing principles, initially introduced by IBM in 2001, in order to cope with the rapidly growing requirements of distributed system management. ACs have been proposed as a promising solution to managing the increased complexity of the distributed electronic systems, adopting the automation and minimising the need for manual operations. In this regard, IBM has introduced a framework of the advanced mechanism for autonomic decision making and self-x properties such as self-configuring, self-optimising, self-healing, and self-protecting. ACs have a very wide scope, which aims to address all aspects of communication networks by empowering network elements to best suit communication intentions, and providing self-management behaviour without explicit interventions [66].

Figure 2.12 compares the four fundamental aspects of self-x properties in autonomic communication and current computing. Each of the self-x property definitions is also clarified. Despite their evident similarities, it is worth pointing out that autonomic communications have slight differences from autonomic computing. According to Dobson et. al. (2006), they both share the same goals. However, autonomic computing is more directly oriented on the IT application software, middleware, and managing the computing resources while autonomic communication is more likely oriented towards distributed systems and services. This includes network resources management at both the user levels and the infrastructure.

The theoretical goals of ACs are to understand how desired element's behaviours are learned, changed or influenced, as well as how these behaviours will affect other

Concept	Current computing	Autonomic computing
Self-configuration	Corporate data centers have multiple vendors and platforms. Installing, configuring, and integrating systems is time consuming and error prone.	Automated configuration of components and systems follows high-level policies. Rest of system adjusts automatically and seamlessly.
Self-optimization	Systems have hundreds of manually set, nonlinear tuning parameters, and their number increases with each release.	Components and systems continually seek opportunities to improve their own performance and efficiency.
Self-healing	Problem determination in large, complex systems can take a team of programmers weeks.	System automatically detects, diagnoses, and repairs localized software and hardware problems.
Self-protection	Detection of and recovery from attacks and cascading failures is manual.	System automatically defends against malicious attacks or cascading failures. It uses early warning to anticipate and prevent systemwide failures.

Figure 2.12.: Self-X Functions [53]

elements, network and groups in return. While the ultimate goal of ACs research is that networks and associated devices and services will be able to operate in a completely unsupervised manner, and to maintain self-x properties [4]. This will also deliver networks the capability to adapt their behaviours dynamically to meet the changing specific needs of end users. Moreover, it will dramatically reduce the complexity and associated costs currently associated with the effective and reliable deployment of networks and communication services.

Furthermore, it is important to point out that the term “autonomic” is not equivalent to “automatic”. This is because ACs work at a high-level network function, which can create their own adaptation strategies, driven by system requirements, to support a self-management functionalities for the network [31]. Besides, human intervention and management is entirely released from the lower level of operation work, whereas the term “automatic” is a simple self-managed process, meaning that simple functions are reacting according to predefined rules to manage the network.

ACs aim to assign swarm intelligence into local components in order to improve the efficiency of applications and services for the next generation networks. In recent years, intensive research on autonomic communication networks (ACNs) is actively engaged in by the European research union and The Autonomic Communications Forum (http://www.autonomic-communication.org/) also coordinates recent work. Current research mainly focuses on promising methodologies like multi-agent technologies and biomimetics, which are believed to be the best way to accommodate complexity in ACNs and meet the requirements of users, vendors and ISPs.

2.7.2. Overview of Self-X Properties

Advanced technologies have dramatically escalated over the past few decades, particularly in communication networks, and play an important role in providing control applications for complex and distributed networks. The business and modern needs force network technologies to be increased in performance, complexity, and functionality. Moreover, current network paradigms are lacking adaptability and are limited to a single domain management, which is managed by network operators. Also, networks have become massive and intractable due to the complexity and that has lead to challenges of scalability. Traditional operations struggle to cope, and thus a new management paradigm is required to fulfil the management of such dynamic infrastructures. Many attempts are introduced to cope with the higher demand on network management problems, but are still not suitable for large scale networks and could suffer from scalability issues. To deal with this problem, IBM introduced the term Autonomic Computing in 2001, which can create its own strategies so it can constantly adapt itself to dynamic conditions of the environment.

Autonomic computing is considered as a promising solution to cope with network complexity by providing a self-management capability based on high-level system objectives. In this regard, each Autonomic Computing System (ACS) should have two main capabilities; it should be able to adapt itself quickly to the dynamic environment, and it should have self-x management properties. The main purpose of the self-x management framework is to work at high-level goal driven functions to deal with the increasing challenges of managing distributed network environments. It requires the important capabilities of (1) gathering related information, (2) modifying the attributes of network nodes, and (3) managing its own functions and adapting itself to the ever-changing environment of the network, which is defined as network autonomy [29]. Scholars also believed that in order to establish a comprehensive Autonomic Communication, the systems should have a sustainable and maintainable information model, and have the ability to make local decisions when collecting information. The main properties proposed by IBM as the basics of AC are as follows [31]:

1. Self-optimisation: system software and hardware should use resources maximally to provide optimised functioning and performance of communications, as well as to detect optimal behaviours in order to improve the systems' performance.

2. Self-healing: This means that the system should be able to detect and recover automatically from a potential problem that might occur in the local software and hardware, for example, the system restarts or reboots a failed element.

3. Self-protection: This means that the system should be able to detect and prevent automatically any malicious attacks on the network, or take the resources offline in case of severe threats. This property also includes the ability to maintain the systems' overall security.

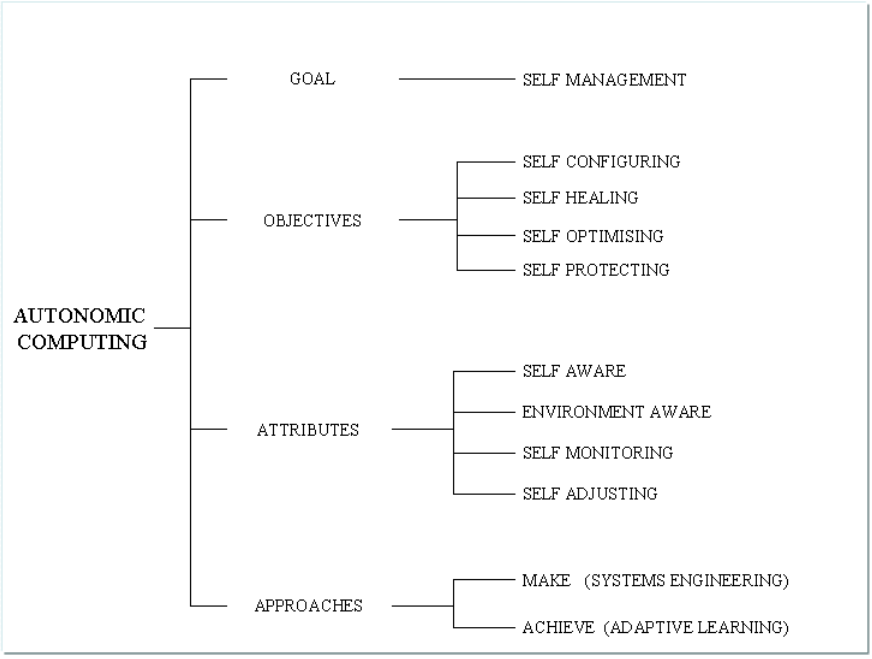


Figure 2.13.: Autonomic Computing Tree [102]

4. Self-configuration: This means that the system should be able to adapt automatically to the ever-changing environment. Moreover, the system should automate the configuration of components and systems where high-level objectives are defined.

Figure 2.13 illustrates the general properties of autonomic systems. The objectives describe the broad system requirements, whereas the attributes identify the primary implementation mechanisms. In order to meet these objectives, the network system must be aware of its actual state (self-awareness) and the current external operating flows (environment-aware). If the conditions change, it will be detected via self-monitoring and adaptations will be made consequently (self-adapting). In addition, this means that the network environment has some knowledge of its available resources, its components, its on-going status, and the status of communications with other systems [101].

In the autonomic research field, key researchers have defined many different attributes and properties for ACs. Each of the self-x literature on ACs is mainly categorised based on the working areas of the self-x functions themselves. Since 2001, the list of self-x properties has grown dramatically. Currently, the self-x framework includes functions such as self-definition, self-organisation, self-adjustment, self-monitoring, self-regulating and so on. For example, “Sabio” is a program that automatically classifies large numbers of documents demonstrating self-organisation and self-awareness properties [77]. IBM and other independent research centres have recently proposed a model to measure the degree of these autonomic systems. Examples of distributed systems and applications of distributed computing include the

following:

- Telecommunication networks.
- Telephone and cellular networks.
- Internet and other computer networks.
- Wireless sensor networks.
- Network applications and services.
- Peer-to-peer networks.
- Massive multi-player online games and virtual reality communities. Assuming every player is a node, autonomous computing can be mostly used for controlling interactions between these nodes.
- Real-time process control.
- Industrial control systems. Assuming the industrial system is an autonomous system consisting of many sensors, control system and actuators (e.g. robots), each actuator can be assumed as a node that can decide for itself in a way that the whole system reaches a specific goal.
- Robot control systems. Just like an industrial system, one can assume a robot is a system reaching a specific goal consisting of many nodes.
- Aircraft or train control systems. Assuming each train or airplane is a node that can decide for itself, the trafficking control centre or network manager can be freed from so many tasks.

3. Distributed Active Information Model Theory

3.1. Introduction

OpenFlow is defined as the first standard communications interface between the control and forwarding layers of an SDN architecture. The OpenFlow approach, however, has some significant limitations: it restricts its use to a single domain, it is not scalable, and it does not adapt well to changes in local environments. As a result of these limitations, network operators today are relatively constant as they seek to reduce the risk of interruption of services. To stay competitive, next generation networks must provide a higher value as well as the best customer service than ever before. There have been many interesting approaches proposed to resolve these limitations. One promising solution for these requirements is to deploy a highly distributed network architecture. Decentralisation is often used to compensate for the single point of failure, one main drawback that is presented when using a single SDN controller as a central node.

The important background based on the O:MIB structure and O:XML technique is presented in this chapter. In addition, a new information model called Distributed Active information Model (DAIM) is proposed to incorporate into the OpenFlow structure at the level of the switches to provide a distributed control plane that will effectively manage the flow tables. The candidate DAIM architecture is also introduced to allow the local decision-making process that will essentially contribute to complex distributed network environments. The DAIM model is a sustainable and maintainable information model that collects, maintains, updates and synchronises all the related information and data objects, in which some or all of the intelligence and management control is locally distributed within the network entities. DAIM provides an adaptable decision-making capability that requires smart agents to manage themselves autonomically and adapt to the ever-changing environments.

This chapter also presents one possible implementation of the DAIM model by using adaptation algorithms embedded with intelligent agents and information objects to be applied to such complex systems. By adopting the DAIM model and these adaptation algorithms, managing complex systems in any distributed network environment can become autonomous, adaptable and scalable. The autonomic approach of distributed systems leads to rapid innovation through the ability to provide network capabilities and new services without having to configure individual devices.

The DAIM model can enhance objects to make their own local decisions through its active performance, and thus significantly reduce the workload of centralised decision-maker. In order to achieve the system's goal, a large number of distributed objects needs to be highly integrated with the DAIM model. Details of the efficient DAIM model and its packet forwarding process are also described, which will hopefully address the schemes of some of the traditional network management protocols such as Simple Network Management Protocol (SNMP), Common Information Model (CIM), and the mechanisms like Policy-Based Network Management.

3.2. Theoretical Framework

Autonomic communication relies heavily on a functional information model that provides source data to drive both decision-making processes and information mining processes. The development of the DAIM model is derived from two major concepts (1) O:MIB, (2) hybrid O:XML. In this section, a new information model is required to cope with the dynamics in distributed ACNs. In addition, an active Object-oriented Management Information Base (O:MIB) is presented as a theoretical framework for the rest of the research with the hope of replacing the traditional management information base (MIB). The corresponding programming language hybrid O:XML is explored as a practical technology to implement O:MIB, with platform-independent Java agents (e.g. Jade and JadeX). Chiang and Braun [29] further found that a variation of XML called O:XML would allow developers to embed Java fragments into XML tags. In addition, they described the MIB as an XML structure, and could embed methods with properties of objects to give them the ability to modify themselves.

3.2.1. O:MIB Theory

The DAIM model can be applied to distributed communication networks for enabling autonomic functions. One of the most significant barriers when dealing with large-scale and complex network systems is the lack of decentralised service management. Because the development of agent-based in the field of Distributed Artificial Intelligence (DAI) has grown rapidly, Autonomous Decentralised Systems (ADSs) and multi-agent technology are by far the best solution for managing complex network environments. The DAIM model consists of adaptation algorithms for adapting the intelligent agents and information objects to be deployed to such distributed electronic systems. The main purpose of designing the DAIM model is to re-engineer the structure of the network information model so that this new structure can effectively cope with the next generation communication networks. It also aims to redesign the traditional MIB structure by adopting the object-oriented principles, which is required to fulfil management services such as configuration management, topology discovery, activating application process, and assigning resource

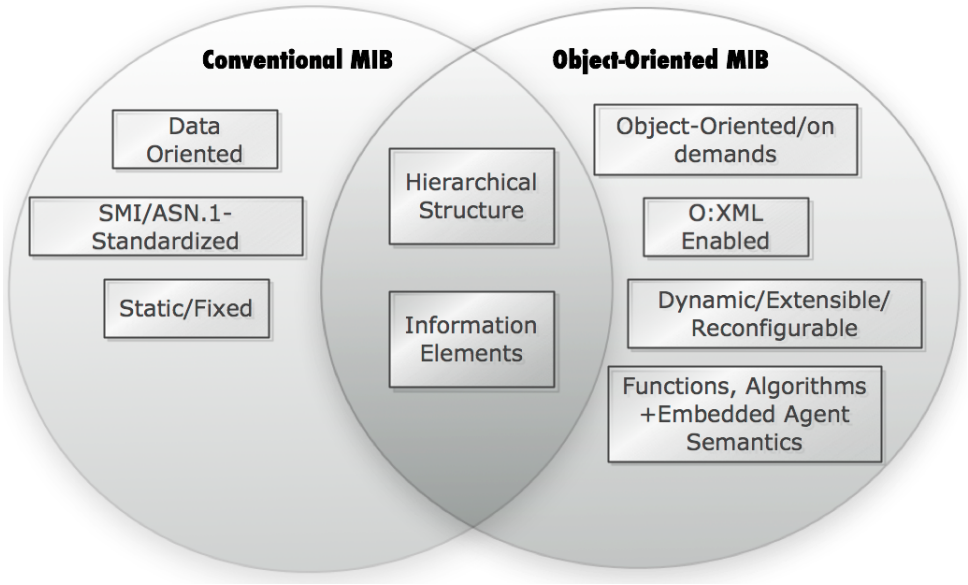


Figure 3.1.: Comparison Between Traditional SNMP MIB and O:MIB [84]

process. Figure 3.1 shows a characteristic comparison between the standard MIBs and proposed O:MIBs.

Furthermore, O:MIB can operate as a part of the distributed information model to enable autonomic software agents that act as the network elements (other routers, switches, hosts, etc.). The Autonomous Agent (AA) inherits the surrounding agent's behaviour and also makes local decisions based on the state of the network. The agents of distributed O:MIB technology will allow the richness of self-organised management. For example, it will allow dynamic software configurations, service activation, and service discovery. Therefore, it is possible to develop the DAIM model specifically with embedded smart algorithms for distributed elements to improve the efficiency of local execution abilities.

These large numbers of heterogeneous O:MIBs need to be well organised in a way to favour the distributed complex environments. This implies a distributed intelligent holonic system in order to manage and implement the O:MIBs in the hierarchical telecommunication system efficiently. The main characteristics of the holonic O:MIBs include; (1) they are object-oriented MIBs with methods embedded, and (2) this O:MIB exists on a holonic-level. For example, they are embedded into individual electronic devices such as mobile phones, printers, and even further in the sub-level of devices (chip-level). Braun and Chiang (2008) state that the holonic system is not only a component-based communication architecture but also a universal way to construct distributed MEs at various levels. The proposed holonic agent-based O:MIB consists of three parts: (1) conventional MIB; (2) user-accessible provisioning; (3) methods/operation. Intelligent algorithms and functions are embedded in each holonic subsystem to fulfil any network tasks for agents to cooperate together

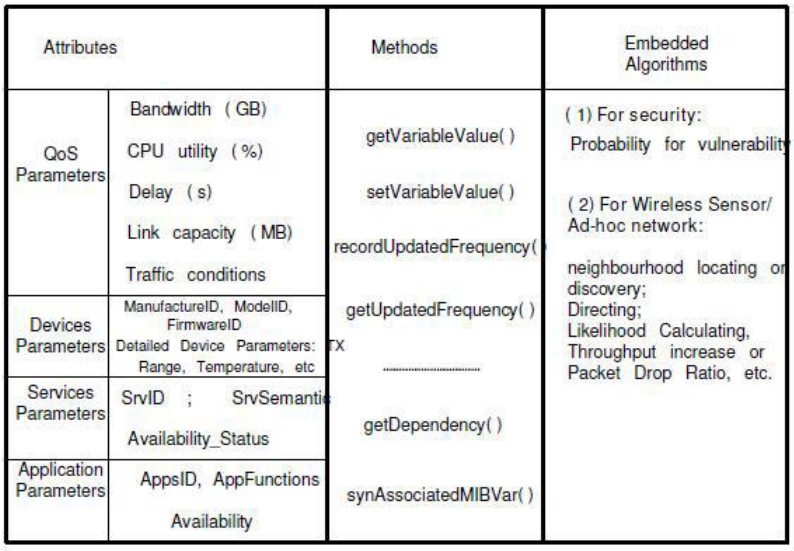


Figure 3.2.: Algorithms and Methods in O:MIB [29]

and share synchronised information.

Figure 3.2 shows the necessary contents of an O:MIB class. It should cover four divisions of information: QoS parameters; device information; service information; application information and dependency information related to devices and services. Chiang et. al. (2007) designed this O:MIB class for each category of network components. When an object is required for activation during run time, any instance of the O:MIB_Class is directly created by java codes via the keyword "new": OMIB_Class OMIB_object = new OMIB_Class( );

The O:MIB model is expected to be used in peer-to-peer networks, mobile technology, and Wireless Ad-hoc Sensor Networks (WASNs) as well as to address other complex issues. O:MIB adopts the object-oriented principles to manage the MIB objects. It has multiple distributed agents that remain in every network component and node, which functions with its own O:MIB as a way to activate applications when required. These network components can also analyse the important data, learning the systems environment, calculate situations, and perform adapting capability. Therefore, a full understanding of autonomic communication will be obtained. Object or element is the basic information unit of the O:MIB. Each important element comprises [20], [9], [79]:

- Attributes: It specifies the information values that represent the characteristics of the managed object identifiers (OIDs).
- Method behaviours: An action that helps to achieve autonomic communications. This can include the self-awareness function in real time and intensive and spatial data.
- Algorithms: These are the algorithms that will support a specific network task

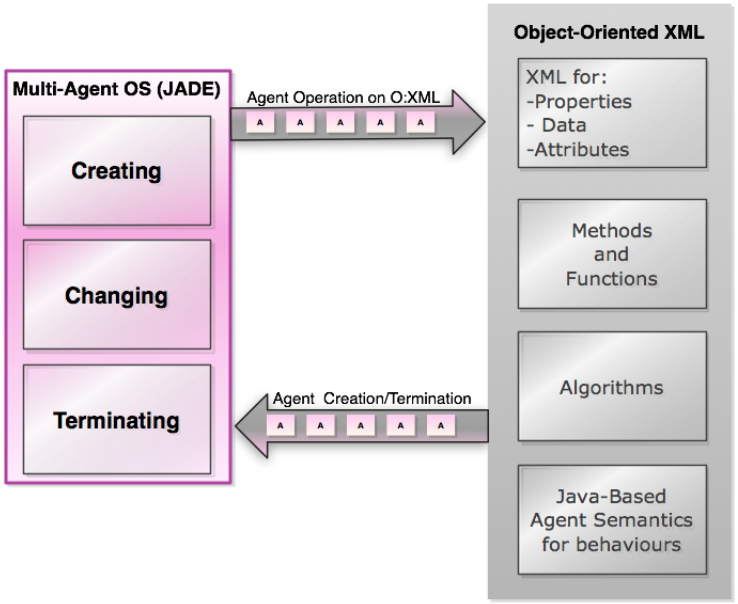


Figure 3.3.: Self-Maintained Process [20]

to be embedded into O:MIB domains. It also represents a set of predefined uses of the total available method calls. For example, humidity of the network environment, temperature monitoring, and predicting the level of raising alarms and risks in autonomic communication networks.

- Messages: In response to the on-demand requests, local messaging daemon action can invoke messages in order to obtain general information about network topology or mapping discovery.

Figure 3.3 indicates the implementation process of O:MIB via O:XML. The software agents remain on each node having O:XML employed to populate the recorded data to the corresponding agents. In addition, Java agents are also involved because it is platform independent, and due to other agent development tools being mainly based on Java technology as well. Each agent is defined from instantiated agents according to their electronic environment. The O:MIB algorithms are invoked by the instantiated agents, whereas information values are re-configured by Java-based agents. Ultimately, the agent's life cycle is accomplished while the program is operating.

The overall stages of this approach can be described as the follows [20], [79]:

1. Observing the current agents on distributed nodes and noticing the environments.
2. Generating new agents when a new environment is identified through the adaptation and learning strategies.
3. Functioning the local O:MIB by invoking the algorithms and methods instructed into the local O:MIB systems by default.

4. Adapting the node in regard to the awareness of the system-level objectives.
5. Wrap up the agent's life cycle until the next round of process is ready.

This efficient O:MIB-based DAIM model approach is introduced to cope with managing autonomic communications in terms of, for example, self-configuring, self-adapting, self-optimising, self-learning and self-awareness. This new information model scheme can also be applied to other self-x properties in ACNs. The attributes of each information object in the O:MIB-based DAIM model can be implemented in one O:XML file. This brings the possibility of embedding DAIM agents into portable communication devices as well as applying into real networks in the future such as wireless networks, including WASNs, Mobile Ad hoc Network (MANET), Peer-to-Peer networks, and Mesh networks [85], [79].

3.2.2. Use of O:XML

Essentially, XML is a general purpose markup language widely used for defining a set of rules for encoding documents, data structure recording, and messaging. XML has a hierarchical structure, which is suitable for most types of documents in the hierarchical networks. It can also be easily parsed by other parser algorithms due to its parsing requirements and strict syntax. Therefore, the XML semantics are widely used to describe the current MIB structure. Java agents can easily parse the document and apply read/write action with XML. However, with its current syntax, it is not feasible to have methods and functions or algorithms embedded into the XML structure. Chiang et. al. (2007) have proposed a hybrid O:XML language with object-oriented functionality, which combines the advantages of XML semantics and the advantages of O:XML language in order to achieve these goals and make the O:MIB structure available in practical use.

O:XML is a newly developed open source language with straight-forward syntax and is build on the basis of standard XML structural scripts. Chiang et. al. (2007) point out that O:XML has more features than traditional XML, which is the purpose for only data structure description, including functions like overloading, polymorphism, exception handling, and threads. All of these features make O:XML suitable for describing the functions and variables and in the O:MIB. Chiang further states that the latest O:XML technology is integrated with Spring application framework. This allows developers to embed the java code (termed as JavaBean) into O:XML while the JavaBeans contain algorithms that can function to find unified information. For example, the probability of neighbourhood availability; or self-reasoning of learning procedures; or even reconfiguring the data value of the component itself.

ObjectBox is the required compiler to interpreter the methods or algorithms in each O:XML file. XML can be parsed easily by Java agents and the XML contents can be integrated into the O:XML output from ObjectBox compiler, therefore, indicates that XML file is also recognisable in part by O:XML interpreter. This drives the idea of possibly combining the XML format with O:XML format. Chiang (2007)

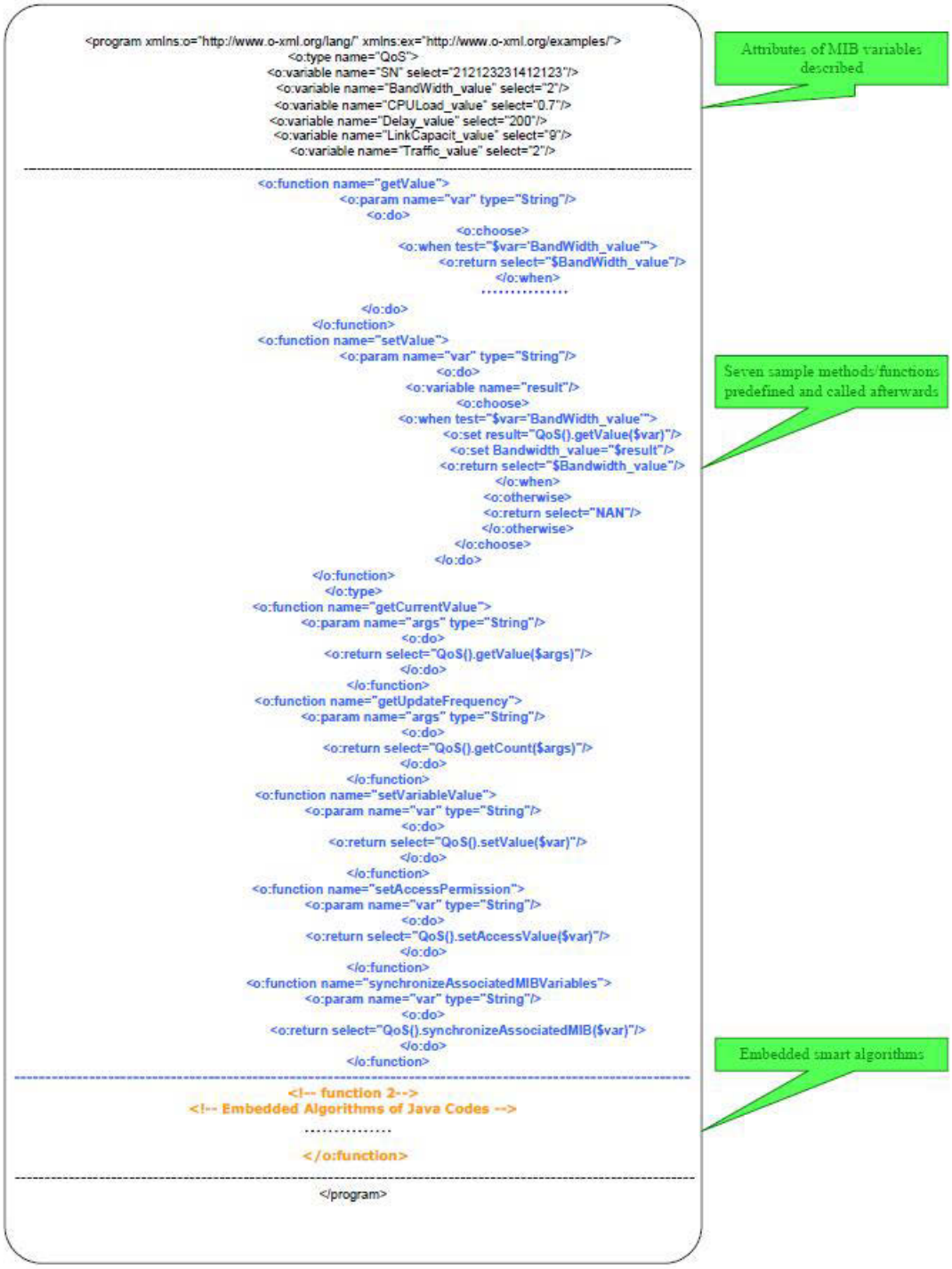


Figure 3.4.: Script Sample of Method Described O:XML Format for O:MIB [32]

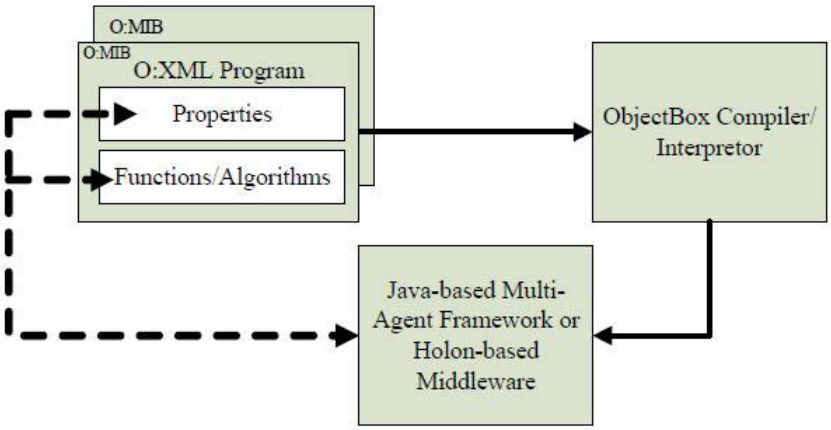


Figure 3.5.: Integration of Multi-Agent Framework with O:XML Implemented O:MIB [30]

combines their advantages and produces a markup structure that hybridises both XML format and O:XML syntax, where data structures like node attributes are described in XML format, whereas functions and embedded algorithms are stated in O:XML format. Figure 3.4 is an example of a sensor network where XML semantics are used to describe attributes of the MIB variables, and O:XML syntax is used to describe methods and algorithms.

- Use of Autonomous Agents

Figure 3.5 depicts the process of software agents (or holons in holon-based middleware), which reside on each node, call functions and algorithms in O:MIB during run-time. O:XML is deployed to populate the recorded information to corresponding agents. The specific agents can be instantiated from agent templates in accordance with diverse electronic environments. The O:MIB functions or algorithms can be invoked by the instantiated agents. Data attributes are able to be reconfigured by Java-based agents (dash line). As an interpreter of O:XML, the ObjectBox can expose the interpreted information in the O:MIB to the agent system (solid line). As a result, the agents' life cycles are fulfilled in the run-time of the program.

3.2.3. Using DAIM as a Logically Distributed Control Plane

In traditional network configurations, if the circumstances should change or the requirements should change, the network then requires re-configuration again. For example, we are applying the classical compiler to the executable paradigm according to the following method. Firstly, we should get a software requirement specification. Secondly, create the codes to meet the requirements. Finally, compile these codes into an executable program. In the case of OpenFlow-based SDN, the process would be as the following: (1) get the system requirements from business drivers; then

(2) create system configuration/code to meet the system requirements; finally (3) compile this into an SDN-enabled configuration.

Thus, the main challenge with complex networks is that if the circumstances or requirements change, the network requires repeating all the above methodology. For example, new bandwidth capacity, the performance requirement, traffic load and traffic requirements change within the network, and hence this would be the problem. In order to overcome this issue, SDN has decoupled the control plane and the data plan by developing components on top of the network operating system as network applications. However, these applications provide very low-level methods for interfacing with the network as the operators configure them. Examples of these are application for discovering the links in the network by sending LLDP packets out of every switch interface, application for mapping network topology, and application for routing.

Figure 3.6 shows the mapping of conventional networks and SDN on how they are constructed or implemented. Essentially, the conventional network deployments is shown on the left-hand side of the diagram in comparison to the SDN paradigm on the right-hand side. The box at the top is typically the business drivers or the requirements that the system and the infrastructure have to provide. For conventional networks, the vertical box on the left is the network artefacts such as switches, routers and links that make up the network, whereas the right vertical box represents the configuration parameters, which are traditionally stored in MIBs and other different ways to store these parameters, or simply on the routers and switches. Current network management paradigm should have the ability to control the configuration in order to manage the network as shown in the configuration column, which then reflex via the artefacts into network behaviours. In recent years, this traditional paradigm can not cope with the growing complexity of today's networks (old mainframe computer).

On the other side, SDN also has the same requirement specifications and business needs. The next block down represents the abstraction of the defined system requirements. Then those definition system codes get translated by a compiler into another abstraction, which is the next block down containing network parameters (e.g. OpenFlow switches) that cause the infrastructure to behave according to business drivers. The SDN approach uses intelligent abstractions to simplify the network including (1) forwarding – a common API for programming network infrastructure; (2) state distribution – a single state distribution algorithm for networking; and (3) global management – where programs interact with the entire network instead of individual nodes. Ultimately, that creates the infrastructure in the bottom horizontal rectangular box, which is providing the same services as the conventional networks would provide but through a different means.

Thus, the alternatives could be implementing the DAIM model as a logically distributed control plane to enable autonomic communications. The paradigm is very similar to other SDN approaches. However, this new application model has some

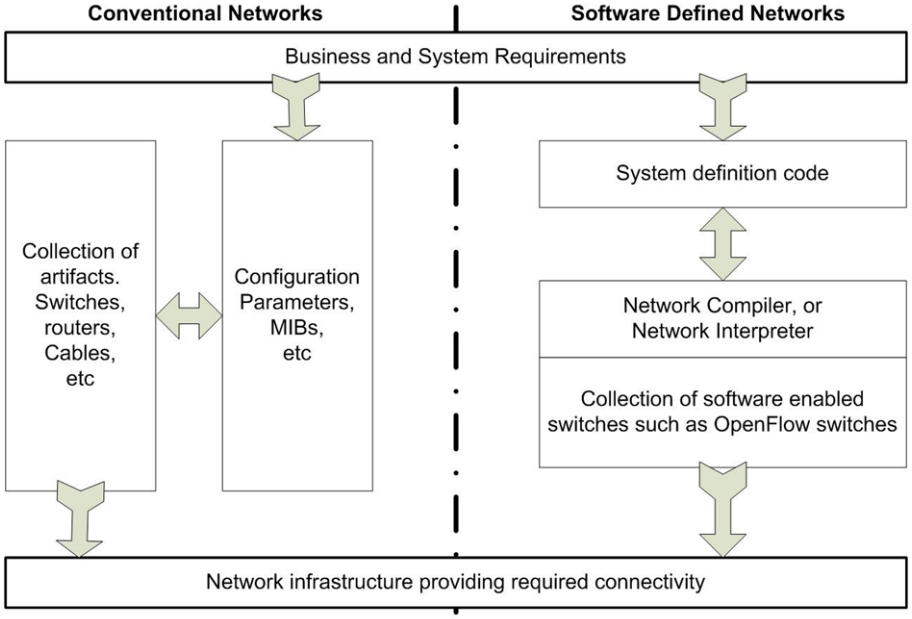


Figure 3.6.: The Mapping of Conventional Networks and SDN

significant differences, which are based on the intelligent DAIM agents for collecting and exchanging information from the system requirement database driven by the business needs. The similarity of the proposed model and OpenFlow is the process of compiling the system configuration into an intermediary binary code and then handing over to a run-time environment (*e.g.*, network operating system) for interpretation/implementation. This process is similar to how the NOX controller operates in an OpenFlow network environment.

The author is proposing to situate the DAIM model within the OpenFlow switches as a logically distributed control plane paradigm and have DAIM compute the flows from local information distributed across network elements. The proposal moves the computational load to the switches and effectively the DAIM model acts as a fully distributed control plane [83]. The DAIM controllers are physically distributed where each controller manages its connected switch and distributes useful information to other instances within its domain. Moreover, the decision-making ability will be local within each switch, on the basis of collecting information by intelligent agents, which allows it to adapt efficiently and react quickly to network dynamics. DAIM is a new way of viewing information objects that manage the behaviour of a network, or any other complex distributed electronic environment. The DAIM model is proposed with the hope of addressing the limitations of current approaches in OpenFlow and future distributed network systems aiming at an autonomic computing management strategy.

Developing the DAIM model can be the means of implementing a physically distributed control plane for SDN. By distributing the DAIM controllers on the network, and having these controllers compute the flows from local information and central-

ised requirements database, it moves the computational load to the switches and could give effect to the so-called fully distributed control plane. Hence, the network would be a truly distributed computing environment. The functionality of the controller migrates into a requirements database while its computational obligations migrate into the DAIM model on the switches. The behaviour of the network is then simply changed through changes in the requirements database. As a consequence, we can envisage OpenFlow being a candidate transport mechanism for applying DAIM.

In the case of an OpenFlow network, the DAIM agents will reside in the network elements such as the OpenFlow switches and would act as the "owners" of the flow table entries. In addition, the actual variables in the OpenFlow table's entries, embedded within the OpenFlow switches, would be the properties of DAIM agents. These agents would then modify or adapt their variables' values so as to implement the requirements of the network driven by the business needs. They would be reacting to changing network circumstances and requirements by changing their properties. Therefore, the DAIM model would extend across all these network elements and could be considered as a logically distributed control plane, which is managing the system requirements to enable the infrastructure according to the business needs. In effect, it is possible to develop a truly distributed communication network by applying DAIM.

3.3. DAIM Model Paradigm

A new information model named: Distributed Active Information Model (DAIM) is presented to allow the local decision-making processes that will essentially contribute to complex network systems like the SDN [20]. An implementation of the DAIM model is expected to enable the requirements of the autonomic components of the distribution networks such as self-management capabilities. An autonomic system in this context means that each distributed device can draw its own strategies for adaptation driven by the goals of the system [27]. The distributed autonomic system adapts the network for needs of dynamic changing in business and reduces operations and management complexities.

Scalability can refer to networks with the ability to increase the number of nodes and the length of links very widely without affecting the performance of the network. To ensure network scalability, it is necessary to use additional communication equipment and flexible network structures. For example, good scalability has multi-segment networks built by using switches and routers and has a hierarchical structure of relationships. Such a network may include several thousand computers while providing each user the right network service quality. Thus, this research notes that by deploying the DAIM model, the scalability of any system could be solved by using a distributed environment as the control platform.

Benefits that can be achieved through implementing the DAIM model include dynamic control of network elements, direct manipulation of new network services without having to configure each individual device, programmability by operators, and the ability to provide automated management that increases network security and reliability. This approach accelerates business innovation by allowing network operators of Information Technology (IT) to program the network in real time to meet the business needs and specific requirements of the users. Accomplishing such approaches is a challenging task. This is because the network control plane mechanisms take several years to be fully designed, and even longer to spread widely, a new control protocol. Moreover, it is important to consider the characteristics of incremental properties, the complexity of new network operators, and some of the missing functions in the network elements [40]. Another part of the approach is that complex distributed systems need to have a well-defined system requirement database (SRD), in order to maintain scalability, optimal service configuration, recovery and stability issues in the functional management domain [20].

3.3.1. Objectives of Designing DAIM

Implementing the proposed DAIM model can also simplify network management by distributing control functions to organise massively dispersed network elements and the ability to proceed with rapid configuration of network devices locally. In addition, the DAIM model could manage complex systems in any distributed network, which makes it possible to become adaptable and scalable to the changing network dynamics. The goals of designing DAIM is to address the research challenges such as managing the complexity of distributed electronic environment and to construct a distributed flow management architecture that will enable self-x functionalities for business needs. Thus, the design requirements to build the DAIM model according to [20], [79], [105] will be as following:

• Compatibility: Consideration is taken of the complexity and the future growth of the networks, where there are varieties of network devices and business needs. Therefore, DAIM will use OpenFlow-based SDN environment as a programmable network to meet different varieties of needs. DAIM can abstract the network management model and services as network applications.

• Model simplicity: DAIM allows switches to make decisions locally. That ability is called autonomic network management. This means that the distributed self-adaptation strategies can maintain the system in the face of changing requirements and unexpected threats to provide for the defined requirement. Hence, operators and programmers are no long required to handle any changes of the requirements either actively or reactively. Network management model and services will be abstracted as network applications.

• End hosts modification: The DAIM model does not require software or hardware changing of the end hosts, where DAIM mainly focuses in forwarding packets.

• Security: DAIM is supporting security by using network security protocols. For example, the messages between System Requirement Database and OpenFlow switches, are encrypted by using Transport Layer Security (TLS).

The new candidate system can enable the development of different network services and applications with embedded autonomic agents. This new paradigm can also be applied to other infrastructures or distributed environments that provide global services such as the National Broadband Network (NBN).

3.3.2. DAIM Model Architecture

This section presents the designed architecture of the candidate DAIM system. The DAIM model can be potentially implemented within each OpenFlow switch using a multi-agent operating system, which is supported by DAIM agents as a field of distributed active Artificial Intelligence (AI) to enable autonomic functionality. The DAIM model is composed of a system requirement database and smart DAIM agents. These components use augmented OpenFlow protocol called DAIM protocol for corresponding messages between them. Intelligent DAIM agents that reside in each switch as an independent computational environment are implemented in a Java Virtual Machine (JVM). In addition, they interact with the database and neighbouring switches to exchange information. Therefore, the DAIM agents can compute their own local decisions according to the business needs defined in the database. The DAIM system requirement database schema holds business needs and the network information such as host identifier, business requirements, topology discovery, QoS, connectivity, bandwidth, security policies, and global view of the entire network (see Figure 3.7). The DAIM model uses this information to make management and routing decisions. The monitoring data includes changing of network links, network topology, changing of host locations, so that would facilitate the calculation and installation of shortest route.

The basic information unit of each DAIM agent includes attributes, method behaviours, algorithms and messaging. These four characteristics can modify the value of its own property. This inverts the traditional *get*→*compute*→*set* process in network management. Instead, property values are changed by either calling methods, or by the modules proactively calling processes that compute the required values, and then call those methods.

To facilitate our approach of managing a distributed environment autonomically, it is necessary to have a well-defined system requirement database or a central database containing the global network view. These databases can support the DAIM agents to make local decisions in terms of forwarding, maintaining, and adapting to the unexpected changes. A local processing engine like DAIM could then be distributed across OpenFlow switches to allow the development of autonomous behaviours.

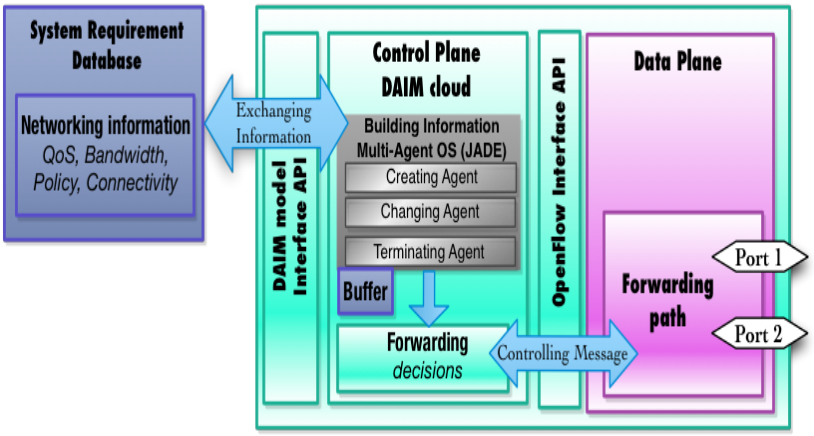


Figure 3.7.: DAIM Model Architecture as an Intelligent Computational Environment

3.3.3. DAIM Agents Implementation

One possible implementation of the DAIM model is to deploy a multi-agent operating system such as JADE/JADEx (Java Agent DEvelopment Framework) that can create, change, and terminate the intelligent DAIM agents. Essentially, these agents have the responsibility to maintain their own values, which they can adapt and modify their properties according to the collected information. The DAIM agents can make their local decisions based on the system requirements driven by the business applications. Moreover, these intelligent agents will also compute the forwarding destination for packets, exchanging data between agents, and update the flow table entries using its built-in methods and algorithms [9], [83], [84].

When the DAIM model receives an unmatched packet, it creates a unique DAIM agent that will query and access the system requirement database or interact with other agents to obtain the local network information needed to determine the forwarding rules. The DAIM agent should be able to check this packet flow against system requirements and other policies to see whether it should be allowed and if allowed, the DAIM agent needs to compute a path for this flow and install the flow entries on every switch along the chosen path. Finally, the DAIM model will send the packet itself back to the origin switch with an action to forward out a specific port. The DAIM agents could provide a distributed environment where the network information is the property (values) of software agents residing on virtual machines that are distributed throughout the network elements. The properties or values are the familiarity notions of object-oriented programming. These software agents not only have the responsibility for maintaining their own property or value but also for adapting and modifying them. Therefore, the DAIM agents have the ingredients to implement autonomic behaviours.

One promising way to effectively implement the DAIM model within each OpenFlow

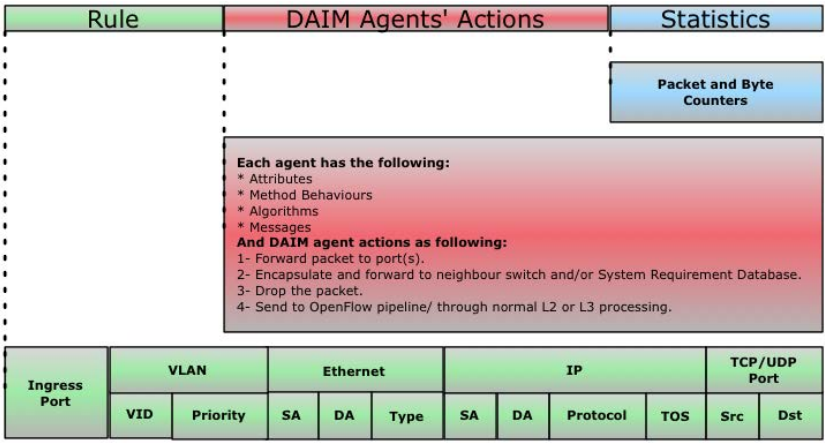


Figure 3.8.: DAIM Agent Owns a Flow Entry in the Flow Table

switch is to develop a multi-agent operating system, which will support the creation of the DAIM agents as a field of distributed active artificial intelligence to enable the autonomic behaviour. Figure 3.8 shows the basic information unit of the DAIM model where DAIM agents manage flow entries, and each of them includes [20]:

- Attributes: specific variables that represent characteristics of the flow entries such as header fields, counters and actions.
- Method behaviours: actions that provide the autonomic functionalities such as self-awareness instantly (temperature, humidity), and self-configuration (switch down).
- Algorithms: algorithms for fulfilment of a network task, can be embedded into DAIM agents, such as informing the DAIM model if any circumstances change within the network and synchronise information between databases.
- Messaging: messages that can be created by the DAIM model as a response of requests to get information (track host location, track topology changing and shortest route). The control databases are connected together and use the DAIM protocol for corresponding messages. At the same time, each switch must be connected to a system requirement database as well as a discover route database to ensure optimal performance of the network.

3.3.4. Uniqueness of DAIM Model

The DAIM model derives its design principles from previous SDN-based research approaches. That is, dividing the control functions, and implementing them on the data forwarding layer as a physically distributed control plane. In so doing, the switches will have more control functions and the routing decisions will be made locally. Furthermore, the DAIM model tries to solve current issues of the centralised management paradigm and difficulties in managing complex networks by enabling

autonomic behaviours for network management based on DAIM agents implementation [20], [79], [105].

The autonomic approach in SDN has not been introduced before. This is because, in the classical routing or switching, fast forward packets (data plane), and high-level routing decisions (control plane) occur on the same device. In addition, most vendors' devices are closed and not accessible. Moreover, if the device is not described by Management Information Base (MIB), the device does not exist. Therefore, each device has an MIB in the ASCII format that network operators can access and edit to achieve the new requirements [23]. From the vendors' side, they have a lack of standard and open interfaces, and there are limitations in the ability of network operators to design the network to meet different individual requirements. This makes a gap between market requirements and network capabilities.

There are five unique aspects of the DAIM model as follows:

- Firstly, the DAIM model is a programming framework for creating distributed control functions within the SDN environment. The DAIM model can be applied to a flow-based routing network such as OpenFlow.
- Secondly, the DAIM model provides clear and direct control over interactions with the system requirement database, and over network state synchronisation by using DAIM agents to gather information and set instruction.
- Thirdly, the DAIM model could solve the scalability issue of centralisation, by distributed control functions within OpenFlow switches.
- Fourthly, using the DAIM model in distributed network environment can solve the robustness and responsiveness issues of the current centralised paradigm. The adaptation algorithms can adapt the distributed nodes by synchronising the network state with the system requirement database.
- Finally, the DAIM model is not similar to the cloud computing model, where cloud computing typically has many separated computing entities, presented as a one computational infrastructure. The DAIM model has many network entities, which are distributed and working independently, where each is represented as an independent computing environment.

3.4. Packet Processing Within DAIM

Packet forwarding of DAIM in an OpenFlow environment would significantly depend on another structural component such as a centralised database to obtain all information regarding network graph and links for computing the shortest path. In addition, the DAIM model can query the registered events from this central database, so that other switches can actively reconstruct the entire network state. As an important advantage of distributing the DAIM model, it will provide some self-management capabilities if any local change happens within an individual switch.

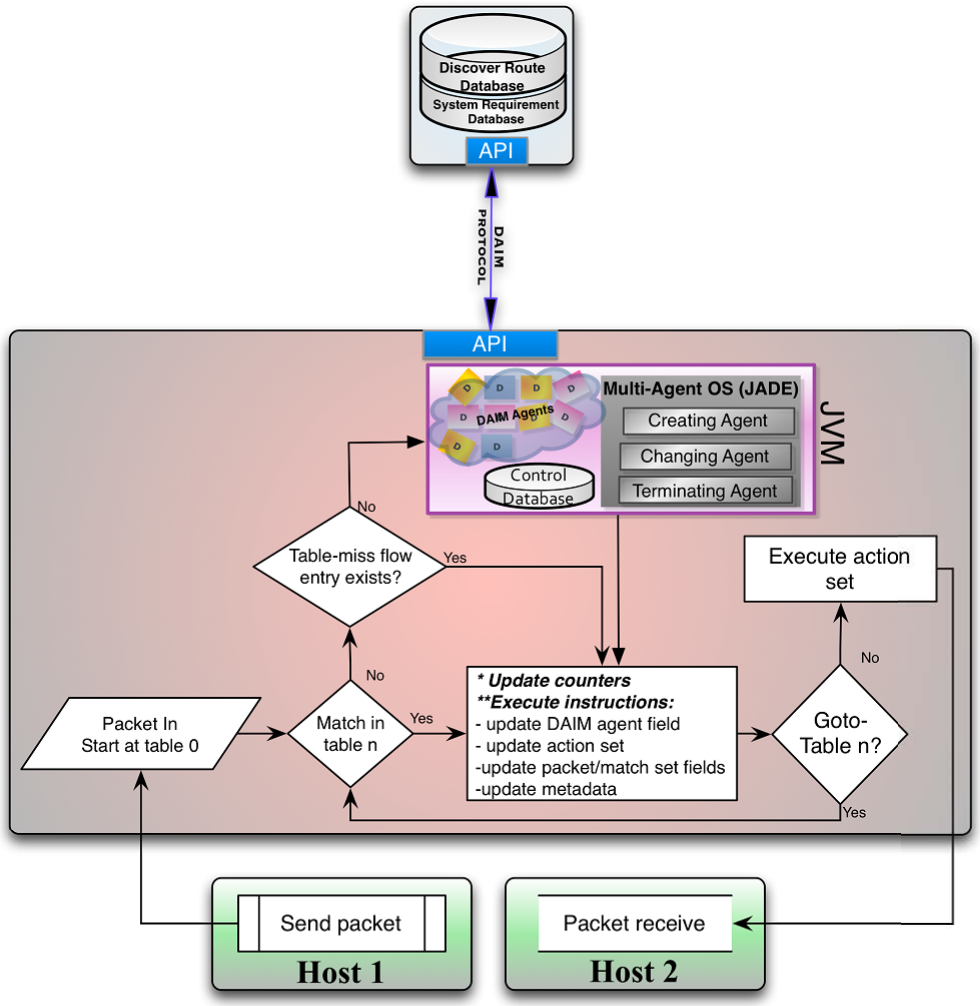


Figure 3.9.: Flow Chart Detailing Packet Processing Within DAIM Model

Each switch will be able to serve any coming packets from other switches locally and adapt to any changed conditions of the dynamic environment. Hence, it is feasible to deploy the DAIM model as a distributed system structure in OpenFlow, which has the ability to share information of the whole network and forward packets locally without support and vetting of the controller. DAIM can create an environment that provides the infrastructures to enable autonomous behaviour for future OpenFlow deployments.

When the packet arrives at an OpenFlow switch, it performs the operations shown in Figure 3.9. Packet headers are used for table lookups depending on the packet type, and typically include various packet match fields, such as source IP, destination IP and MAC destination address. The switch begins with performing a table look-up in the first flow table, and may perform table look-ups in other flow tables [33], [44]. For example, the flow tables are sequentially numbered, so the packet is matched against flow entries of flow table 0. Other flow tables may be used depending on

the outcome of the match in the table 0. If a flow entry is matched, the instruction set included in that flow entry is executed, and the counters associated with the selected flow entry must be updated. Those instructions may direct the packet to another flow table, where the same process is repeated again. On the other hand, the instructions could forward the packet to a Table-miss or the DAIM model in the case of flow entry not matched.

If there is a no matching rule in the flow table for that particular packet, the switch sends the packet to a Table-miss flow entry. The flow table configuration defines the behaviour of a Table-miss entry by using wild-carding rules. A Table-miss flow entry in the flow table specifies how to process unmatched packets with the options of sending packets to the DAIM model or passing them to a subsequent table. Moreover, the table-miss flow entry behaves similarly to any other flow entry, where it does not exist by default in a flow table. The DAIM model can support this Table-miss process by adding a Table-miss flow entry or removing it at any time, as it may also expire. However, the DAIM model is responsible for all unmatched packets, if there is no Table-miss flow entry.

Normal OpenFlow process	**Candidate OpenFlow + DAIM process**
Unknown packet arrives at a switch	Unknown packet arrives at a switch
Switch cannot match a flow table entry	Switch cannot match a flow table entry
Switch uses OpenFlow protocol to forward packet to controller	Switch instantiates a new agent, which "owns" a new row in the flow table.
Controller computes forwarding destination for the packet.	This is a unique Agent (which "owns" that row in the flow table). It computes the forwarding destination for the packet according to the system requirement database and exchanges information between agents
Controller uses OpenFlow protocol to update flow table entry on the switch, which now knows how to forward the packet, and similar ones	The agent updates the local flow table row using its built-in methods and eventually forwards the packet to the destination

Table 3.1.: Comparison of Normal and Candidate Processes

Table 3.1 shows a comparison between normal OpenFlow processing and the candidate DAIM model within OpenFlow which typically depend on the system require-

ment database and the intelligent DAIM agents. Therefore, an individual switch can serve any coming packets locally.

DAIM agents will be bounded to a particular variable such as a flow entry variable and have some level of self-adaptation strategy to manage the variables for forwarding according to the business needs. For example, when the DAIM model receives an unmatched packet, the switch creates a new agent, which "owns" a new row in the flow table. This unique agent is able to access and control network elements. It will compute a path for the unmatched packet, and install flow entries on every switch within the chosen path to forward that flow. For more details, DAIM agents are able to check this flow against system requirements and other policies to see whether it is allowed to be processed.

3.5. Risk Scenarios of the DAIM Model

The theoretical design of the DAIM model includes two main central databases: the system requirement database and the discovery route database. However, the DAIM model is logically distributed across OpenFlow switches, which needs to synchronise with both central databases in order to manage the network effectively. Thus, the switches are fully responsible for serving all packets within its site, unless a failure happens (*e.g.* port down). If any failure occurs, switches that are connected to the affected switch should be able to reconfigure themselves to their neighbour switch, forcing the flow modification to happen. This new switch will use its adaptation strategies to synchronise actively among all databases for determining the requirements and calculating a new path for the connected hosts.

Early SDN architecture relies heavily on a centralised controller, whereas the proposed DAIM model adopts the distributed management paradigm to ensure a high degree of network reliability. The potential consequences upon the failure of the system requirement database and discovery route database are the following:

Firstly, when the OpenFlow switch receives an unknown packet, it will not be able to calculate a path for forwarding to the destination. Furthermore, the switch will convert the flows to be handled by the normal Ethernet switching operation, for example, traditional Layer 2 switching capabilities, VLAN isolation and QoS processing. However, the system will not perform optimally because all autonomic functions will be disabled. Secondly, DAIM can no longer store the collected information in those databases to construct the central view of the network, and hence is incapable of performing autonomic actions such as self-adaptation, self-configuration and self-protection [26].

However, the above issues can be avoided by the design and functions of the database buffer as well as failover capability in DAIM to maintain network state in the case of any failure. For example, the switch can send a *port-status* message to the DAIM model when a link is broken. After receiving this information, DAIM will perform a

recalculation of the shortest path and sends the flow modification messages to install a new path. At the same time, flow entries that are configured in the flow table of the failed switch will be deleted.

Moreover, the DAIM model can actively synchronise with the rest of the system's components upon start-up by using DAIM agents. The network information collected by these DAIM agents serves as the heartbeat of the designed model. Intelligent DAIM agents can also share local information of managed elements and immediately synchronise them within both databases. Therefore, the collaboration of agents to manipulate network elements could enable autonomic services such as self-adaptation and self-learning.

Ultimately, the proposed DAIM model will hopefully address the limitations of current SDN approaches and future distributed network systems aiming at an autonomic management strategy. The invented DAIM model will also enable some requirements of autonomic functionality for distributed network components such as self-learning, self-adaptation and self-CHOP (configuration, healing, optimisation and security). Using DAIM, each system component can be adaptable according to any changing conditions of the dynamic environment without human intervention.

Part II.

Proving the “Propositions”

4. Integrating DAIM to OpenFlow-Based SDN Using Mininet Emulator

4.1. Introduction

In this chapter, DAIM implementation based on the theory presented in Chapter 3 has been described in three phases including the basic carrier, semi-distributed and fully distributed functionality. The purpose of developing DAIM controller is to conduct a logically distributed control plane by integrating DAIM into the OpenFlow architecture at the level of the switches. This enables the switches to process flows locally by distributing DAIM to operate the packets instead of a centralised controller in an SDN environment. In more details, the three main modules that construct the DAIM ecosystem are discussed with some sample code reviews and flowchart diagrams of the implemented algorithms (code available at Appendix A). The new DAIM structure aims to address the limitations of current OpenFlow deployments with respect to scalability, adaptability and robustness. Moreover, building the distributed control plane offered by the DAIM controllers could create the appropriate "hooks" that will allow the development of autonomic management strategies.

The Mininet emulator is used to set up network scenarios for testing and validating the DAIM controller. Mininet is a network emulation tool that provides researchers and developers with the ability to emulate OpenFlow network on a single computer. Mininet can create many virtual hosts and switches by using lightweight process-based virtualisation on a single OS kernel. It can also create user-space or kernel OpenFlow switches, controllers and hosts to communicate within the emulated environment. Virtual Ethernet (***veth***) pairs are used to connect hosts and switches in Mininet. Development processes such as testing, debugging and deployment can be simplified when using Mininet. In addition, new SDN-based applications can firstly be implemented and tested on an emulation of the anticipated deployment network. Once a prototype works on Mininet, it can be deployed on actual networks with real operational infrastructure for real-world use without any changes. Unlike a simulation tool, Mininet emulates an environment that runs real, unmodified code, application code, OS kernel code and control plane code. As a result, implementation of the DAIM controller in Mininet can be repeated in the real world.

Furthermore, various tasks and scenarios are demonstrated to validate the functionality of DAIM controller such as executing a ping command, streaming media and transferring files between hosts. Hence, the performance benchmarks of the DAIM model with distributed network architecture are believable, and can be proved to possess efficiency, robustness, flexibility and will improve the scalability needed in an OpenFlow environment.

4.2. DAIM Model Implementation

A new information model named: Distributed Active Information Model (DAIM) is presented to allow the local decision-making processes, which will essentially contribute to complex distributed network environments. An implementation of DAIM model is expected to introduce the requirements of the autonomic components of the distribution systems. An autonomic system in this context means each distributed device can draw its own strategies for adaptation driven by the goals of the system. The distributed autonomic system adapts the network for the needs of dynamic changing in business and reduces operations and management complexities. The DAIM model can provide distributed systems with a sustainable information model, which collects, maintains updates and synchronises all the related information. Each device has decision-making ability on the basis of information that was collected and can adapt autonomously to any changing environments.

Benefits that can be achieved by implementing the DAIM model include better control of any OpenFlow-enabled network device from any vendor, flexible network management and automation, and rapid configuration to update devices across the entire network. The approach accelerates business innovation by allowing network operators of Information Technology (IT) to program the network in real time to meet the business needs and specific requirements of the users. The distribution system approach leads to rapid innovation through the ability to provide network capabilities and new services without having to configure individual devices or wait for the launch from the seller. DAIM does not require any changes to the OpenFlow standard and guarantees loop-free forwarding as well as network resilience.

The initiatives and outcomes of the DAIM model can be applied to currently available communication networks to resolve complex issues. Deploying DAIM will also enable the development of autonomic management strategies. An autonomic system in this content means that each distributed device can draw its own strategies for adaptation driven by the goals of the system. The distributed autonomic system adapts the network for needs of dynamic changing in business and reduces operations and management complexities. As a result, enterprises get the programmability, automation and the control of the network, which enable them to build highly scalable and flexible networks that can be easily adapted to different changing environments. This section describes the three phases of developing the DAIM model as follows:

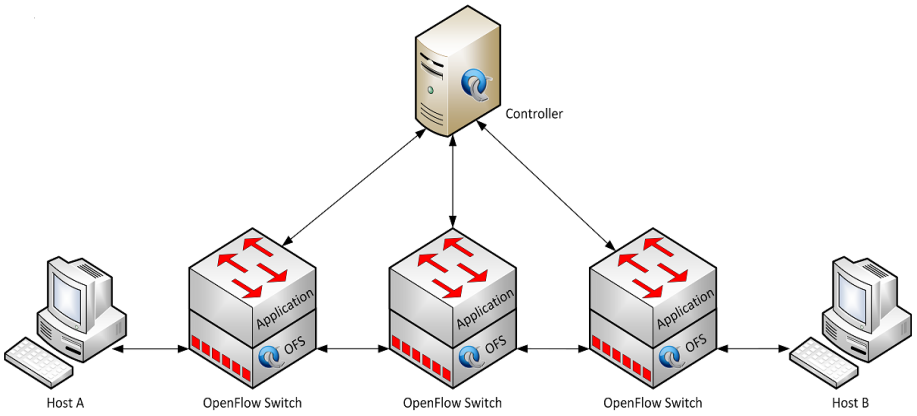


Figure 4.1.: DAIM Implementation Phase 1

4.2.1. Phase 1: Basic Carrier Functionality

In the first phase, DAIM model has been initially integrated to the SDN architecture by applying the implemented communication channel between the NOX controller and an OpenFlow switch. The DAIM model is developed in C++ using an open source NetBeans IDE and NetBeans Platform. It is programmed based on the object-oriented paradigm where data structures that contain data are created in the form of *fields* (attributes), and code in the form of *procedures* (methods). DAIM model is designed and developed based on the concepts of *objects*, which they can access and modify the data fields of the object with which they are associated and often interact with one another. Because C++ programming language is class-based, DAIM model is implemented as instances of classes, which essentially also defines their type. In addition, DAIM model is created in different classes to represent each type of OpenFlow messages such as Packet-In, Packet-Out, Set-Config and Flow-Modification. The *openflow.h* is included in the header file to facilitate the implementation of such messages.

At this phase, DAIM is defined as a basic application channel used for message transmission in an OpenFlow network. No control functions have been implemented. Essentially, this phase describes a simple communication channel based on a client-server model (see Figure 4.1). DAIM is developed using UNIX BSD socket programming API where the server socket connects to OpenFlow switch and the client socket connects to the NOX controller. The DAIM application listens on a particular port (default 6633) for messages from the NOX controller. The network architecture of OpenFlow is still the same, which has all of the high-level routing decisions made by the NOX controller, but will have the DAIM application processing and forwarding all OpenFlow messages (without any modification) from controller to switch and vice versa instead of the original secure OpenFlow channel. In this phase, the communication module of the DAIM model is implemented to accommodate the OpenFlow messages. Moreover, the DAIM model is able to perform *get* and

set process driven by the OpenFlow controller as well as updating the forwarding table via flow modification message.

DAIM application can also forward statistics and network information to the NOX controller. This can work in both pull-based (active probing) and push-based (publish/subscribe mechanism) mechanisms. DAIM application uses the OFPT_Port_Status protocol message to retrieve a set of statistics from that particular switch. Network applications on top of NOX can then view these statistics in a key-value format.

4.2.2. Phase 2: Semi-Distributed Functionality

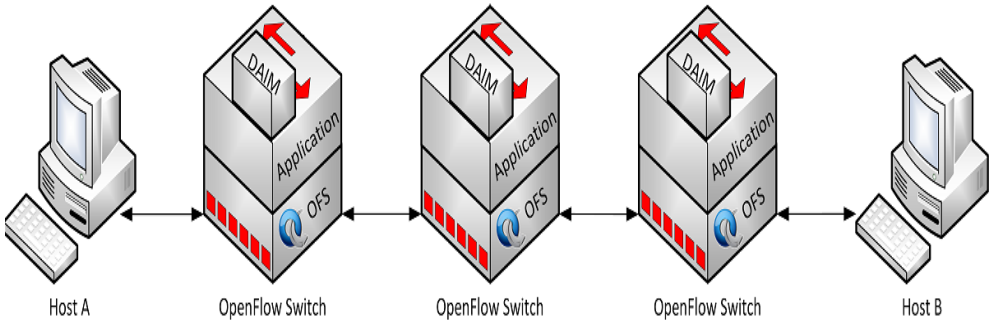


Figure 4.2.: DAIM Implementation Phase 2

The structure of phase 2 is similar to the previous phase but includes some level of distributed event-based control plane for OpenFlow by distributing the DAIM controller to each connected OpenFlow switch to perform its own functions locally (see Figure 4.2). A major distinction from the first phase is that the NOX controller gets replaced by each distributed DAIM controller. Now the DAIM controller will not only be a basic carrier for OpenFlow messages between switches and controller but also can gather information from the network and propagate its local MAC address table to act as an intelligent Layer 2 learning Ethernet switch. For example, it can store network information when connected nodes perform an ARP or ICMP session, and hence it is possible to forward flows directly from the switch according to the flow entries that are pre-defined by the DAIM controller. In more details, DAIM controller is also capable of updating the forwarding tables via flow modification messages on the switches and can execute some functions across the managed switches. The algorithm for building an intelligent Layer 2 learning switch functionality consists of the following steps:

```
1 Algorithm: Ethernet Learning Switch
2 For each packet from the switch,
```

```
(1) Use MAC source and incoming port number to update the data
  structure.
(2) if Ethernet frame type is LLDP(0X88cc)
(3) Drop the packet // Do not forward the link-local traffic
(4) else if MAC destination is multicast,
(5) Flood the packet
(6) else if the output port is same as the input port
(7) Drop the packets.
(8) else if the data structure contains a port for the MAC
  destination,
(9) Forward the packet to the destination address.
(10) else if the data structure does not contain MAC destination
  port
(11) Flood the packet
(12) else install the flow entry in the switch flow table.
```

At this stage, the DAIM controllers do not have a separate communication channel to exchange information between one another. The OpenFlow switches listen to its own connected DAIM controller on a specific port (*e.g.* 2000) for control messages. Another significant control function of DAIM in this phase is performing a certain process for querying network statistics (READ STATE protocol message). In addition, the DAIM controller is able to identify if a network problem happens, and also sends the corresponding message to the switch to update the ports and flow tables. In all cases, the connection is established per the TCP followed by, hello, and feature request/reply messages. This connection has to be developed in advance in order to process those functions.

Additionally, it is important to note that each DAIM controller may find it necessary to share information with each other for a number of reasons. For example, an internal control function may need to reserve resources across multiple domains of control or a "master" process may need to share policy information with a backup "child" process. This can also apply to hosts communicating across different domains, where source and destination information are shared and pre-installed among DAIM controllers. When the packets arrive at a new switch of another domain, this flow can be sent without controller interactions because of the installed entries in the flow table of the switch. As a result, distributed DAIM controllers can distribute their state across multiple running instances for fault tolerance.

4.2.3. Phase 3: Fully Distributed Functionality

Phase three aims at migrating all computational power to the DAIM model, which can manage each connected switch to produce some level of distributed computing network system. In addition, the implementation of this phase focuses on distributing the high-level decision making of traffic control to the DAIM controllers. Figure 4.3 shows a distributed networking architecture of OpenFlow embedded with DAIM controllers, supported by the core modules to manage the connected switches.

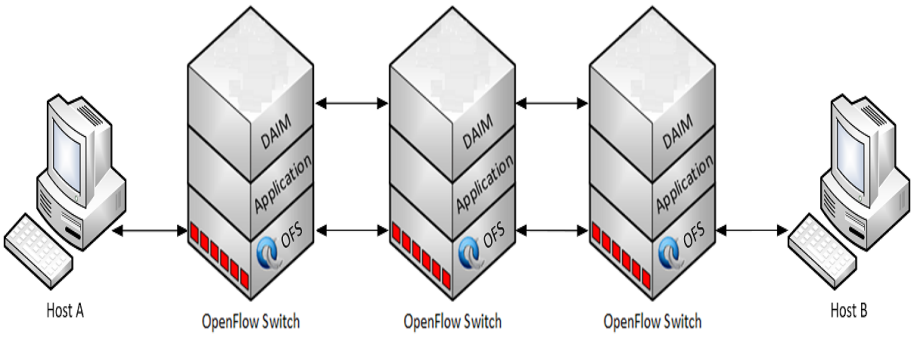


Figure 4.3.: DAIM Implementation Phase 3

Each DAIM controller can actively share all information regarding its portion of the network to ensure fine-grained network wide consistency. For coordination purposes, DAIM controllers can also publish events as well as actively synchronise its local information with other associated controllers in order to construct the global network view. By allocating one DAIM controller for each OpenFlow switch, the switches can serve any coming packets locally. This distributed traffic management allows multiple levels of redundancy as each site has the ability to perform wide area system functions. Thus, DAIM provides a logically distributed control plane, where each controller manages its affected switch and distributes useful information to other instances and if necessary communicates with the neighbouring domain. DAIM has the feasibility to be deployed in any distributed system structure.

Because there is no controller-to-controller communication protocol defined by OpenFlow, it is necessary for any type of distribution or redundancy in the control plane. Only one centralised controller may cause a single point of failure for the entire network. Therefore, distributing the DAIM controllers would allow backup controllers to take over in the case of a failure and become a truly distributed networking system. This can be achieved by each DAIM controller communicating with one another to share and exchange its local information such as flow tables, links, ports and hosts. This information can be used to indicate the network's latency and bandwidth, or the liveliness of a controller-controller connection. For example, topology discovery within DAIM will depend on the embedded messages in the LLDP frames sent through the OpenFlow protocol. In addition, network operators can also maintain multiple DAIM controllers and switches remotely from a single logical point.

To facilitate cross-controller communications, the mechanism for sharing messages between DAIM controllers is to mimic LLDP packet in/out and setting Ethernet frame type inside the OpenFlow message to LLDP. Since messages from the controller will mimic LLDP, the packets will propagate through the existing network

medium instead of dedicated controller medium. By using this method, the controllers have no control over the underlying network medium and hence have to manage how to forward messages to the destination controller. Inside each DAIM controller, this function is implemented using a separate thread to send messages independently (see section A.1). Note that the iteration of LLDP is every ten seconds.

4.3. DAIM Software Specification

This section describes the software specification of the DAIM model, the implemented OpenFlow messages and modules. The implemented modules are comprised of the Communication module, the local Storage module, and the Controller module as well as the most significant messages that are needed for the communication between the switch and the DAIM controller. More details are presented in the following subsections.

4.3.1. Overview of Model

The DAIM model is implemented as an application on top of the OpenVswitch running in Mininet. DAIM uses the OpenFlow protocol to update forwarding tables in the local memory and switches. In the current design of DAIM, the control application executes as a thread on top of the Linux sockets. DAIM model uses a separate control channel to invoke commands between the controller and switches. It is developed to support a cross-platform architecture, which has improved the variable types of running on both 32 and 64 bit CPU.

As can be seen in Figure 4.4, the DAIM ecosystem consists of three core modules namely *communication* module, *controller* module and *local storage* module working independently to achieve one single goal. The output from the entire ecosystem is communicated over the OpenFlow protocol, whereas the internal communication between the modules happens in one single process. The controller module is responsible for managing all the other modules so that the management and control can be traced to the controller module. Also, the controller module can actively support useful services for the other two modules. For example, the communication module provides routines for creating OpenFlow messages, and the local storage module provides storage information as well as the retrieval routines. Since all of the modules reside in a single process, the communication between each module is extremely fast.

The DAIM ecosystem uses OpenFlow protocol based on the OpenFlow switch specification version 1.0.0 (Wire Protocol 0x01). For the implementation of DAIM controller, the *openflow.h* is included in the header file to model the protocol and its defined messages as closely as possible. It is important to note that not all OpenFlow messages are implemented in the current state. However, the most important

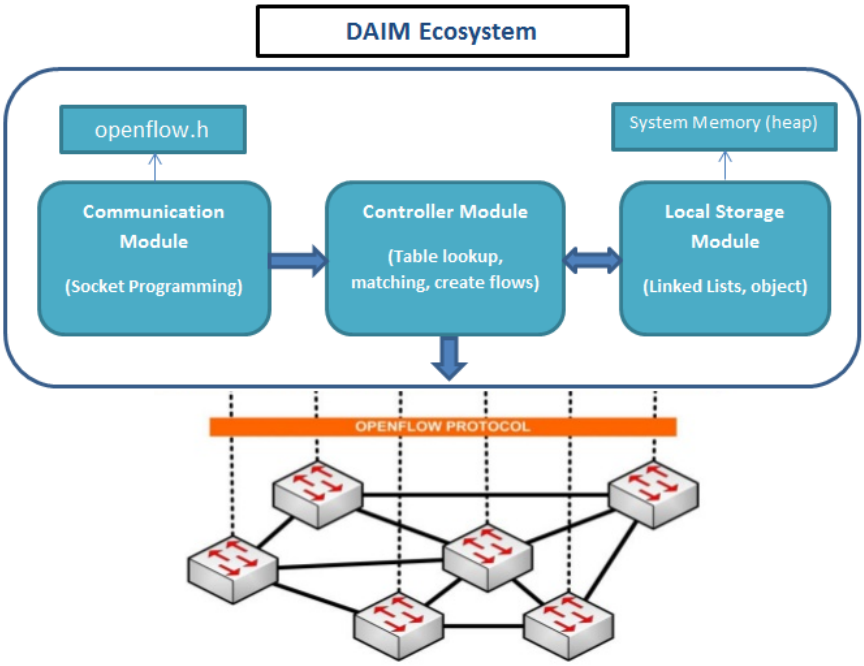


Figure 4.4.: DAIM Model Ecosystem

subsets are implemented to allow extensive emulations of OpenFlow enabled networks. The OFPT_Header message class contains all of the messages in sub-classes, which include the definition of OpenFlow message header and its corresponding C++ structures.

Inside the communication module, there are various routines for creating different types of OpenFlow messages. Figure 4.5 shows the implemented messages of the OpenFlow protocol in the DAIM model. The OFPT_Hello, OFPT_Features_Request, and OFPT_Features_Reply messages are implemented for the initialisation of the OpenFlow connection between DAIM controller and the switches. Echo request/reply messages are sent from either controller or switch and must return an echo reply. These messages are used to indicate the liveliness of a controller-switch connection and are repeated every 15 seconds. The controller uses an OFPT_Set_Config message to set the configuration parameters in the switch, whereas the OFPT_Packet_In message is used by an OpenFlow switch to notify DAIM of an unknown packet or to forward a packet to DAIM in the case of an associated action of a match. This message contains either the entire encapsulated packet or just the buffer ID of the buffered packet. The OFPT_Packet_Out message is a controller-to-switch message used by DAIM to forward a packet out of a specified port at the switch. Finally, DAIM can manage the flow table of a switch through the OFPT_Flow_Mod message type, which comprises the header match fields as well as the corresponding actions.

The DAIM application *(ani.cpp* file) establishes the linking of DAIM modules to

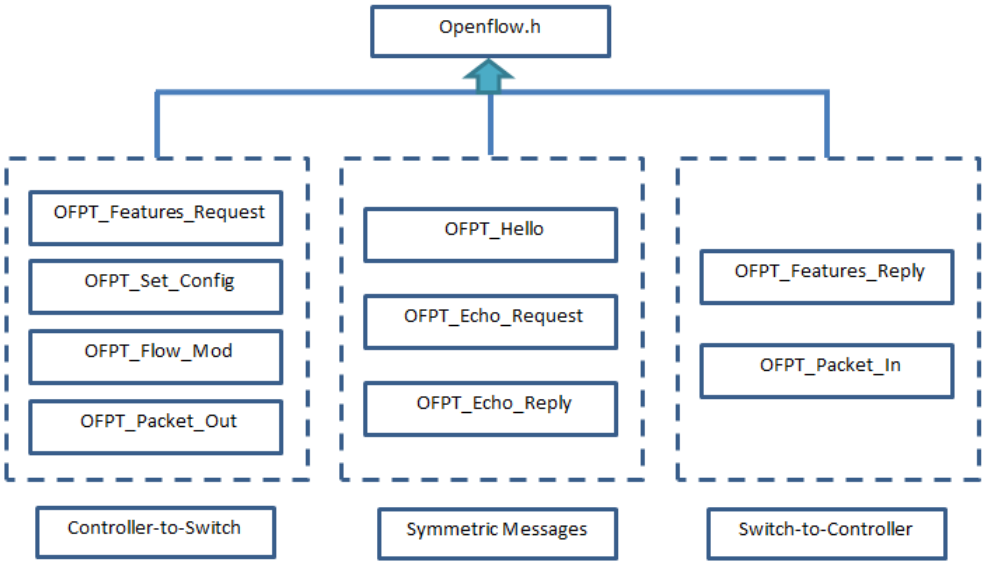


Figure 4.5.: Implemented OpenFlow Messages

Header Files	Description
openflow.h	defines the structure for OpenFlow protocol messages
arp.h	defines ARP message structure
eth.h	defines Ethernet header message structure
linked.h	defines the structure and class for ANI linked list implementation
ani.h	main program function declaration
com.h	OpenFlow messages creation and function declarations
con.h	function and structure definitions and declarations for the control module

Table 4.1.: DAIM Application Header Files

API name	Purpose
htons ()	To convert 2 bytes data from host byte order to network byte order, e.g. messages send from the DAIM to the switch are converted using hton function series
htonl ()	To convert 4 bytes data from host byte order to network byte order
ntohs ()	To convert 2 bytes data from network byte order to host byte order, e.g. messages received from the switch to the DAIM are converted using ntoh function series
ntohl ()	To convert 4 bytes data from network byte order to host byte order
memcpy	To copy n bytes of memory from one variable to other, e.g. used by the control and communication subsystem to copy MAC addresses into the DAIM tables objects
memset	To clear the first n bytes of the memory area in the variables, e.g. used by different sub-systems to give initial zero value to the variables
memcmp	To compare the first n bytes of one variable memory to other variable memory, e.g. used by the linked list and control module to determine host matching
strerror	To get the string describing the last system call error, e.g. used by different sub-systems to display error messages
exit	To exit from the main program

Table 4.2.: API Dependencies Used to Implement DAIM Modules

the local operating system. Therefore, this holds the entire DAIM application and produces the executable file for the DAIM controller. Moreover, the DAIM application relies on several related dependencies, for example, the Linux operating system routines as well as other own sub-systems routines. The header files (descriptive files) for these routines are listed in Table 4.1.

There are other important subroutines commonly called and used repeatedly during the executable of the program. This makes the program shorter and easier to write. Table 4.2 presents a list of all the API dependencies used to develop three modules of DAIM with a brief description including some examples of how these dependencies are utilised.

4.3.2. The Communication Module

The communication module is responsible for providing the communication mechanism and creating sockets of a two-way communication link between the controller and switch. In more details, it creates two processes for handling the communication between the switch to DAIM as well as DAIM to OpenFlow controller. The main

purpose of these two processes is to forward messages both ways among the controller and switch and create signal handlers for notification of errors to the processes as well as the exiting of processes upon user's request. Sockets facilitate TCP/IP communication between two separate systems. Initially, sockets are created and then transferred over to the respective processes.

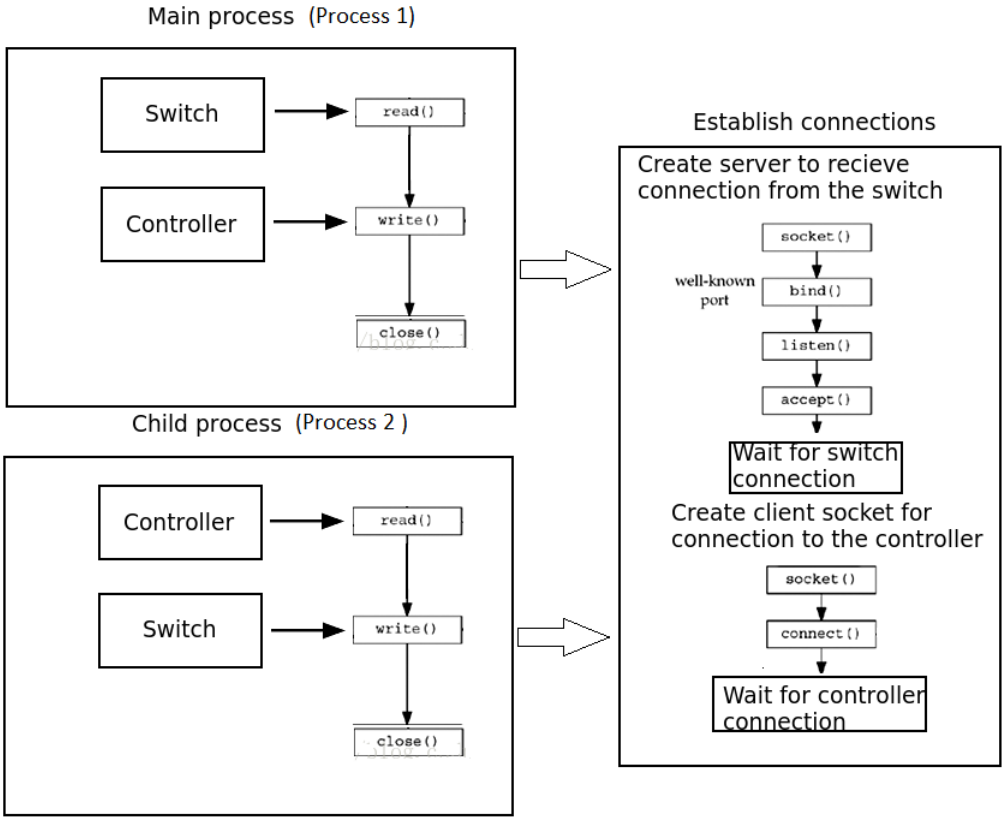


Figure 4.6.: Unix Socket Connection Setup

The client-server model is one of the most used communication paradigms in networking systems. Clients normally communicate with one server at a time. From a server's perspective and at any point in time, it is not unusual for a server to be communicating with multiple clients. A client needs to know the existence of the address of the server, but the server does not need to know the address of (or even the existence of) the client prior to the connection being established. Clients and servers communicate by means of multiple layers of network protocols in which this context will focus on the TCP/IP protocol suite.

The scenario of establishing connections between the server socket to receive the connection from the switch and the client socket to communicate with the controller is shown in Figure 4.6. Socket creation follows client and server model, where DAIM is the *server* socket for the switch and also is acting as the *client* socket to the OpenFlow controller.

The socket API and support for TCP and UDP communications between end hosts are described. Socket programming is the key API for programming distributed applications on the Internet.

4.3.2.1. System Calls Used for Creating DAIM Server

```
socket ()
// socket for server mode
app_sockfd = socket (AF_INET, SOCK_STREAM, 0);
    if (app_sockfd < 0)
    {
        fprintf (stderr, "XApp: Error creating server socket: %s\n",
           strerror (errno));
       return -1;
    }
```

The *socket* () function creates an endpoint for a process to read and write to a distant or a local process. All the system calls for reading and writing to the files are also applicable to the socket file returned by *socket* () function. *SOCK_STREAM* in the socket function establishes a socket with connection-oriented TCP/IP communication, which is essential for establishing OpenFlow communication. The third parameter 0 in the socket function tells the operating system to use a suitable protocol for the requested socket.

```
bind ()
// Bind server socket to the main process
memset (&app_addr, '\0', sizeof (app_addr));
app_addr.sin_family = AF_INET;
app_addr.sin_addr.s_addr = INADDR_ANY;
app_addr.sin_port = htons (app_port);
// Bind server socket to the main process
    if (bind (app_sockfd, (struct sockaddr *) &app_addr, sizeof (
      struct sockaddr)) < 0)
    {
        fprintf (stderr, "XApp: Error binding server socket: %s\n",
          strerror (errno));
        return -1;
    }
```

The *bind* () system call binds the Internet Protocol (IP) address and port to a specified process. Using bind system call, DAIM behaves like a controller to the switch by binding controller address and port to its process. By setting *s_addr* to *INADDR_ANY*, DAIM is able to accept connection from a switch with arbitrary IP address.

```
listen ()
// Set clients listening queue size
if (listen (app_sockfd, 1) == -1)
{
```

```
5     fprintf (stderr, "XApp: Error establishing listening: %s\n",
        strerror (errno));
6     return -1;
7 }
```

The *listen* () system call fixes the number of clients (switches) which can wait in a queue before receiving error connection message. By setting the second parameter to 1 in the listen system call, the pending switch connections are restricted to only one switch.

```
1 accept ()
2 // Accept connection from the switch
3 sw_addl = sizeof (sw_addr);
4 memset (&sw_addr, '\0', sizeof (sw_addr));
5 sw_sockfd = accept (app_sockfd, (struct sockaddr *) &sw_addr, &
    sw_addl);
6 if (sw_sockfd < 0)
7 {
8     fprintf (stderr, "XApp: Error accepting connection from switch:
        %s\n", strerror (errno));           return -1;
9 }
```

The *accept* () system call will return a socket describing the client socket file information and also fill in the client's IP address to the provided server structure. By using the socket returned from the accept system call, DAIM is able to read from the switch as well as write to the switch.

4.3.2.2. System Calls Used for Creating DAIM Client

```
1 socket ()
2 con_sockfd = socket (AF_INET, SOCK_STREAM, 0);
3 if (con_sockfd < 0)
4 {
5     fprintf (stderr, "XApp: can not create socket for the controller
        : %s\n", strerror (errno));             return -1;
6 }
```

Similar to DAIM server socket, this sample script shows a *socket* () function parameter that is also used for connecting DAIM client to the OpenFlow controller socket process.

```
1 connect ()
2 // Define Address for the controller
3 con_addr.sin_family = AF_INET;
4 bcopy ((char *) con_server->h_addr, (char *) &con_addr.sin_addr.
    s_addr, con_server->h_length);     con_addr.sin_port = htons (
    con_port);
5 // Connect to the controller
6 if (connect (con_sockfd, (struct sockaddr *) &con_addr, sizeof (
    con_addr)) < 0)
```

```
{
    fprintf (stderr, "XApp: can not connect to the controller: %s\n"
      , strerror (errno));
    return -1;
}
```

The *connect* () system call is used to connect to the remote controller based on the socket file returned by the socket function, which fills in the remote controller socket address into the *sockaddr* structure.

The *fork* () system call is used to create two processes to handle DAIM to controller and DAIM to switch communication. For example, inside process 2:

```
while (1)
{ memset (buffer, '\0', sizeof (buffer));
  buffer_len = 0;
  buffer_len = read (con_sockfd, buffer, sizeof (buffer));
    if (buffer_len < 0)
    {
        fprintf (stderr, "XApp: Can not read from the controller: %s
          \nExiting\n", strerror(errno));             close_sockets
          ();
        _exit (EXIT_FAILURE);
    }
    else if (buffer_len > 0)
    {
        sw_write_len = write (sw_sockfd, buffer, buffer_len);
        if (sw_write_len < 0)
        {
            fprintf (stderr, "XApp: Can not write to the switch: %s\
              nExiting\n", strerror(errno));
              close_sockets ();
            _exit (EXIT_FAILURE);
        }
    }
}
```

In the above sample script, the read and write system calls are used to receive and send data from the controller to the switch respectively.

Furthermore, the implementation code for the signal handler function inside the main process is as follows:

```
// Set main process exit handler
memset (&sigIntHandler, '\0', sizeof (sigIntHandler));
sigIntHandler.sa_handler = &exit_handler;
sigaction (SIGINT, &sigIntHandler, NULL);
```

The *sigaction* function is used to bind a signal to a specified action. The parameter *SIGINT* (Ctrl+C) will trigger *sigIntHandler* function and close the process appropriately. The closing of the process requires freeing of allocated memory by the application as well as closing of open sockets.

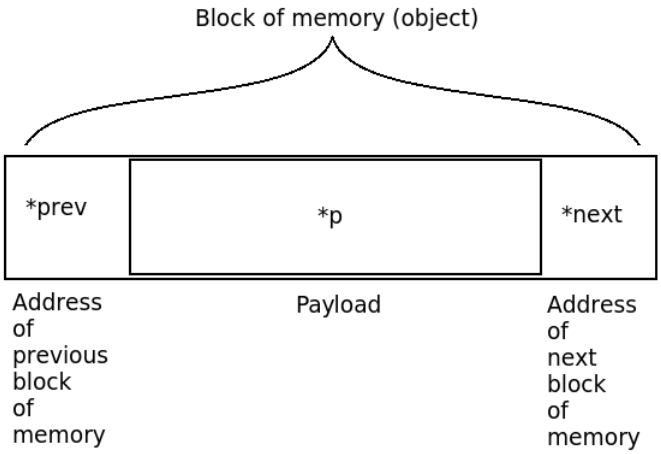


Figure 4.7.: DAIM Storage Block of Memory (Object)

4.3.3. The Local Storage Module

The DAIM local storage module requires two main components including the **Hosts** and the **Ports** table. The information of network devices is stored in these tables, which are implemented using *Linked list* based storage. Linked list is a way to store data with structures so that the programmer can automatically create a new place to store data whenever necessary. Specifically, the programmer writes a struct or class definition that contains variables holding information about something, and then has a pointer to a struct of its type. Each of these individual struct or classes in the list is commonly known as a node.

To store network information into the storage table, we created a list of objects and cached information using blocks of memories (objects). Memory for the objects is then allocated by using C++ dynamic memory allocation methods. In linked list based storage, each object keeps the address of its preceding and subsequent objects. Therefore, each object can refer to its next and previous objects. The system's block of memory (object) is depicted in Figure 4.7.

The ** prefix* represents a pointer that holds the address of a memory block. Hence, the payload **p* can point to any block of memory storing single or a combination of information such as host MAC address, switch data path ID, switch port, and IP address. The structure for the block of memory is defined by:

```
struct object
{
    void *p;
    struct object *next;
    struct object *prev;
};
```

To store objects, we have created the object_lists (tables) structure in the storage module. The object_list is a representation of a table, and each of them includes the host entries object_list and the port entries object_list. Each object_list structure stores information such as the object size, address of the first object, address of the current object and address of the last object. Each object_list has a number of functions to manipulate the object_list itself. These functions can be used to add a new object, remove an object, free the memories used by the objects and retrieve an object from the list.

In addition, the objects in the hosts table object_list are used to store the following data (payload):

- Switch data path ID
- Mac address of the host
- Switch port for host
- IP address of host

Therefore, the objects for the host table entries will have the above payloads which are defined by:

```
struct switch_host
    { uint64_t datapath_id;
      uint16_t port_no;
      uint8_t mac_addr [MAC_ADDR_LEN];
      uint32_t ip_addr;
    };
```

Furthermore, the class object_list defines an object that is related to a table or the object list. This object_list class has the following structure shown in the code below:

```
class object_list
{
    struct object *first;
    struct object *current;
    struct object *last;
    long size_of_object;
public:
    object_list ();
    void *add_object ();
    long get_list_size ();
    int remove_object (const void *ob);
    int free_list ();
    void *get_object ();
    void list_rewind ();
    void set_object_size (long s);
    long get_object_size ();
};
```

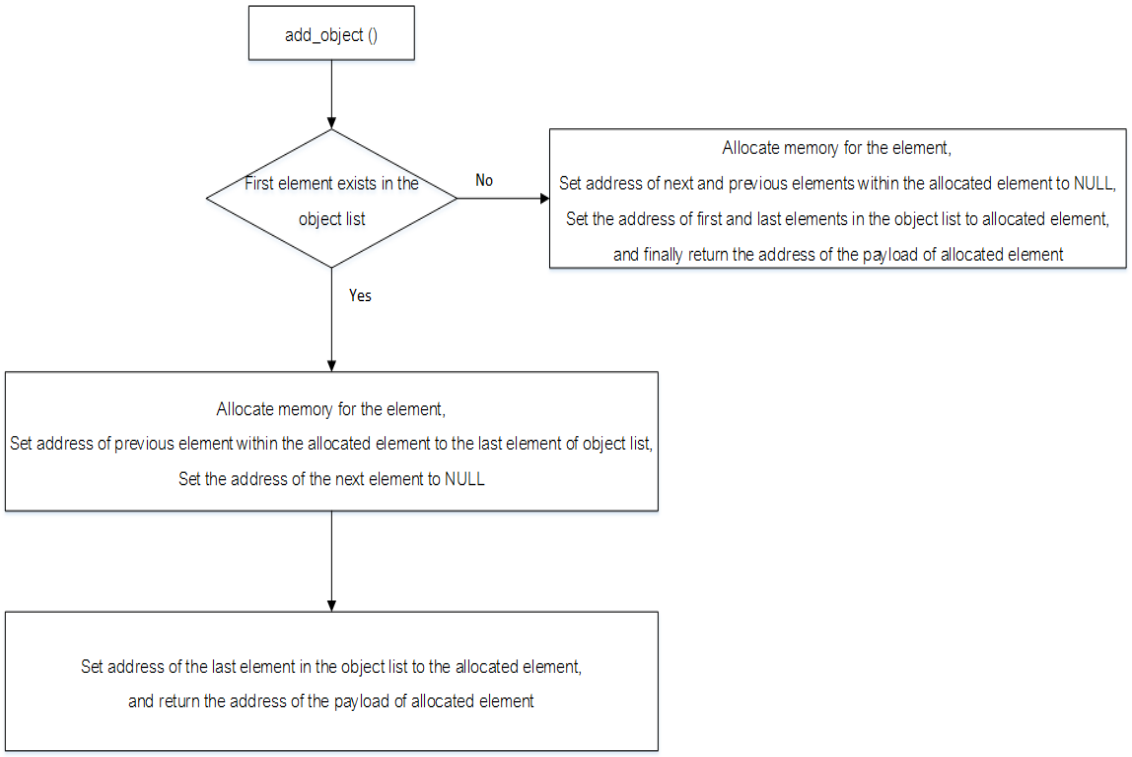


Figure 4.8.: Flowchart Detailing the Process of add_object ()

Each object of object_list type holds the address of first element, the current element, the last element in the list as well as stores size of each element in the list. Each object_list object has some methods defined as public functions to perform some operations on the list such as adding the new elements, removing of existing elements, and getting an element from the list. These public functions implement the memory allocation as well as freeing of memory for the elements from the system's heap memory. By using these functions, we can interact with the object list and modify its properties. The actions of public functions in the object_list class are listed in Table 4.3.

The most important methods in the storage module (Linked list) are add_object, remove_object, and free_list. The examples of the program flow charts for add_object, remove_object, and free_list are mostly self-explanatory as shown in Figure 4.8, Figure 4.9, and Figure 4.10 respectively. As can be seen in Figure 4.8, the add_object () method first allocates memory for the object, and then the method will check whether there is an element existing in the table. If there is no element in the table, then set the first and last element to be the allocated object. Furthermore, the method will set the previous and next element of the allocated object to be NULL (non-existing value). If there is already an element which exists in the table, set the previous element of the allocated object to the last element in the table, and set the last element in the table to be the allocated object. Also, the method will set the next element in the allocated object to be NULL.

Methods	Actions
add_object	Allocates memory for a new element and returns the pointer of allocated element payload
get_list_size	Returns number of objects in an object list
remove_object	Removes a matching object from an object list
free_list	Frees the memory allocated to each element in the list
get_object	Returns a pointer to the next element in the list (useful for reading through elements in the list)
list_rewind	Sets the current element in the list to be the first element
set_object_size	Sets the size of each object(element) in the list
get_object_size	Returns the size of each element in the list

Table 4.3.: Public Functions with Associate Actions in the Object List Class

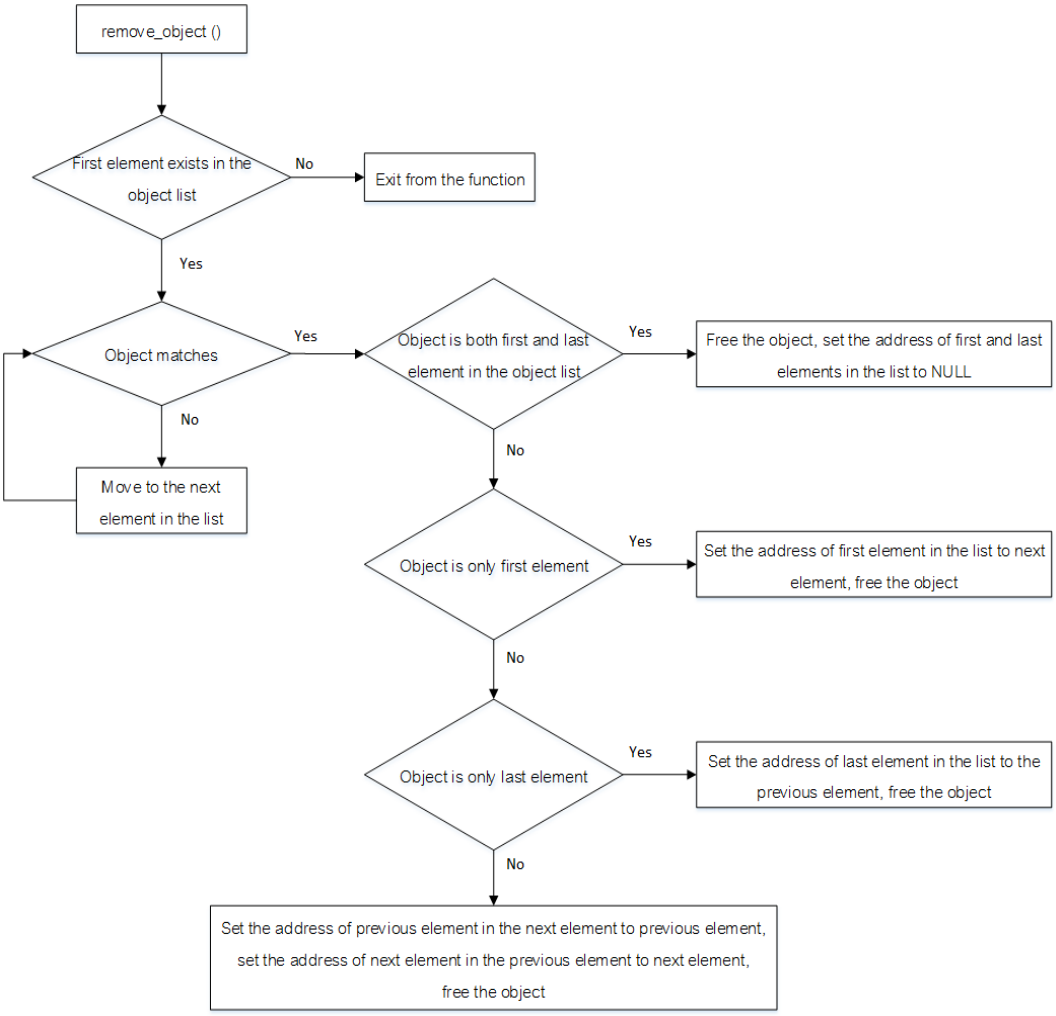


Figure 4.9.: Flowchart Detailing the Process of remove_object ()

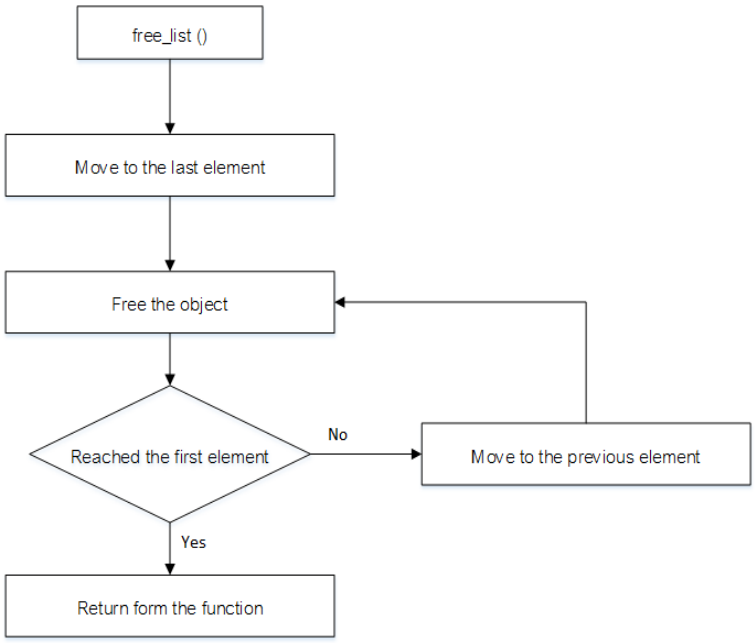


Figure 4.10.: Flowchart Detailing the Process of free_list ()

The remove_object method matches every element in the table to the object that needs to be removed. The actions on the matching element are determined by four conditions: (1) if the match found is both first and last element in the table (in other words the only element); (2) the match is first element; (3) the match is last element; and (4) the match does not fall into the first three conditions. The flow chart detailing the process of remove_object method is shown in Figure 4.9.

The free_list method moves to the last element in the table and frees the element from the table. After freeing each element, the method moves to the previous element of the last removed element. This way the method continues freeing each element in the table until it reaches the first element and finally removes the first element. Figure 4.10 illustrates a simple flow chart detailing the process of the free_list method.

4.3.4. The Controller Module

This section summarises the major functions that the controller module must perform when a packet arrives at an OpenFlow switch. Detailed descriptions are provided to elaborate these functions including high-level summary such as algorithms and example code reviews. This will help organising the functions to make them understandable. A picture of the major groups of related requirements and how they relate such as a top level data flow diagram is also described. In addition, the major services provided by the controller module are demonstrated in this section by mode of operation, object class, and functional hierarchy, which

OpenFlow Messages	Purpose
OFPT_Hello	Symmetric Establishing message
OFPT_Features_Reply	Features support report from the switch to the controller
OFPT_Packet_In	Packet forwarding decision
OFPT_Echo_Request	Checking of liveliness of the controller

Table 4.4.: OpenFlow Messages Handled by the Controller Module

makes the most logical sense for the DAIM model.

The controller module is responsible for maintaining the connectivity with the switches as well as allowing the switches to decide what actions to apply when a particular combination of network request is queued to the switch.

Firstly, the controller module establishes the connection to the switch using TCP/IP protocol stacks. To be able to receive the connection from an arbitrary OpenFlow switch, the controller module creates a server socket that listens for any incoming connections from the switch. The process of waiting for a switch connection requires three actions:

- Establish a server socket (Passive socket, ANI controller socket)
- Binding of server socket to the controller application process
- Accepting connection from remote sockets (Switch socket)

The above tasks are accomplished by using the following two methods inside the ANI (DAIM) controller:

```
1 int create_ani_socket ()
2 int create_switch_socket ()
```

The create_ani_socket method combines the system call functions that are required for establishing a listing of server sockets, whereas the create_switch_socket () combines the system calls for creating an active socket for the switch, which will be eventually used by the ANI controller for sending and receiving messages from the switch.

Upon successful connection to the switch, the controller module goes in a sequential mode of operations by using the "communicate_with_switch" method, which performs the two functions in order, read a message from the switch and send a respective reply message to the switch. The methods that reside within the "communicate_with_switch" are *read_from_switch* and *send_information*.

Depending on the nature of the messages received from the switch, the controller then decides and directs the switch to perform the follow-up actions. The OpenFlow message types used by the controller module are listed in Table 4.4. In addition, the implementation code for the matching of OpenFlow packets is shown below:

```
pheader = (struct ofp_header *) read_bufferp;
if (pheader->type == OFPT_HELLO)
{
    if (action_hello (read_bufferp) == -1) return -1;
}
else if (pheader->type == OFPT_FEATURES_REPLY)
{
    if (action_features_reply(read_bufferp) == -1) return -1;
}
else if (pheader->type == OFPT_PACKET_IN)
{
    if (action_packout(read_bufferp) == -1) return -1;
}
else if (pheader->type == OFPT_ECHO_REQUEST)
{
    if (action_echo(read_bufferp) == -1) return -1;
}
else if (pheader->type == OFPT_PORT_STATUS)
{
}
```

When the switch receives a flow's first packet, it will be sent to DAIM application because there is no flow entry in the switch's flow table to match this flow. The DAIM controller module determines the action of switch packet forwarding upon receiving the OpenFlow OFPT_Packet_In message from the switch (see Figure 4.11). From the Packet_In OpenFlow message, the controller module first checks whether the packet is an ARP type by analysing the encapsulated Ethernet II frame type.

From the Ethernet II frame, the controller module then parses the MAC address of the source host as well as its IP address and adds this information into the respective tables if they are not already stored in the tables. The code for retrieving the host MAC and IP addresses is displayed below:

```
p_packet_in_eth = (struct of_ethernet *) p_packet_in->data;
src[0] = p_packet_in_eth->src[0];
src[1] = p_packet_in_eth->src[1];
src[2] = p_packet_in_eth->src[2];
src[3] = p_packet_in_eth->src[3];
src[4] = p_packet_in_eth->src[4];
src[5] = p_packet_in_eth->src[5];
dst[0] = p_packet_in_eth->dst[0];
dst[1] = p_packet_in_eth->dst[1];
dst[2] = p_packet_in_eth->dst[2];
dst[3] = p_packet_in_eth->dst[3];
dst[4] = p_packet_in_eth->dst[4];
dst[5] = p_packet_in_eth->dst[5];
p_packet_in_eth++;
p_packet_in_arp = (struct arp *) p_packet_in_eth;
ip_src = ntohl (p_packet_in_arp->arp_ip_source);
ip_dst = ntohl (p_packet_in_arp->arp_ip_dest);
```

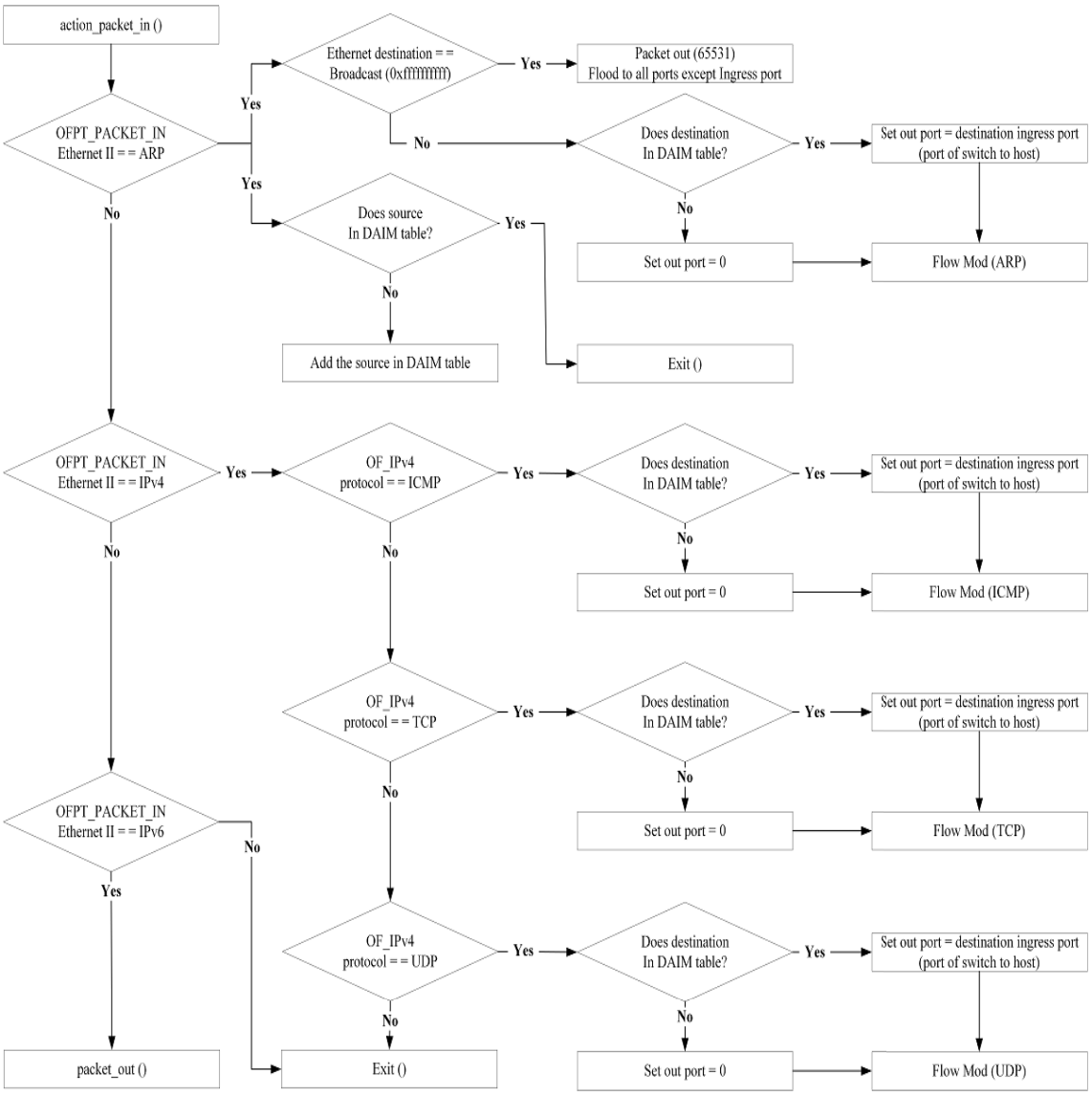


Figure 4.11.: Packet Flow in an OpenFlow Switch Controlled by DAIM

The functions for checking whether the host has been added to the local storage table and creating a new entry in the host table are given in the code example below:

```
while (hostp != NULL) {
    if (memcmp (hostp->mac_addr, src, sizeof (uint8_t) *
      MAC_ADDR_LEN) == 0)
    {
      if (verbose == true) cout << "Info: host already in the table"
         << endl;
      add_host = false;
      break;
    }
    hostp = (struct switch_host *) hosts.get_object();
}
if (add_host == true)
{
    struct switch_host *p = NULL;
    memset (&new_host, '\0', sizeof (struct switch_host));
    new_host.datapath_id = switch_id;
    new_host.port_no = in_port;
    new_host.mac_addr[0] = src[0];
    new_host.mac_addr[1] = src[1];
    new_host.mac_addr[2] = src[2];
    new_host.mac_addr[3] = src[3];
    new_host.mac_addr[4] = src[4];
    new_host.mac_addr[5] = src[5];
    p = (struct switch_host *) hosts.add_object ();
if (p == NULL) cerr << "Error: can not create object in host table"
  << endl;
    else {
    ((struct switch_host *)p)->mac_addr[0] = new_host.mac_addr[0];
    ((struct switch_host *)p)->mac_addr[1] = new_host.mac_addr[1];
    ((struct switch_host *)p)->mac_addr[2] = new_host.mac_addr[2];
    ((struct switch_host *)p)->mac_addr[3] = new_host.mac_addr[3];
    ((struct switch_host *)p)->mac_addr[4] = new_host.mac_addr[4];
    ((struct switch_host *)p)->mac_addr[5] = new_host.mac_addr[5];
    ((struct switch_host *)p)->port_no = new_host.port_no;
    ((struct switch_host *)p)->datapath_id = new_host.datapath_id;
    ((struct switch_host *)p)->ip_addr = ip_src;
    }
}
```

After the MAC addresses are learned, if the destination MAC address in the Ethernet II frame is a broadcast type, then the controller module sends a packet_out message to the switch with the forwarding action of flooding the packet to all switch ports except the ingress port. If the destination MAC address is not broadcast, the controller module then looks up for the host MAC address in the table, and if found it will send an ARP flow modification message with an action to create a new flow entry in the switch flow table. This flow entry will regulate the forwarding of all future packets by matching the incoming packet's source and destination addresses.

The implemented code for sending packet out messages to the switch is as follows:

```
p_packet_out = (struct ofp_packet_out *) write_buffer;
p_packet_out->header.version = 1;
p_packet_out->header.type = OFPT_PACKET_OUT;
p_packet_out->header.xid = htonl (transaction_id);
p_packet_out->buffer_id = htonl (buffer_id);
p_packet_out->in_port = htons (in_port);
p_packet_out->actions_len = htons (sizeof (struct ofp_action_output)
  );
p_action_output = (struct ofp_action_output *) p_packet_out->actions
  ;
p_action_output->type = htons (OFPAT_OUTPUT);
p_action_output->max_len = htons (0);
p_action_output->len = htons (8);
p_action_output->port = htons (OFPP_FLOOD);
p_packet_out->header.length = htons (sizeof (struct ofp_packet_out)
  + sizeof (struct ofp_action_output));
write_buffer_len = send(sw_sockfd, write_buffer, sizeof (struct
  ofp_packet_out) + sizeof (struct ofp_action_output), 0);
```

Furthermore, if the packet_in message of Ethernet II frame is an IPv4 type, the controller module parses the protocol type from the IP header frame and then follows the same procedure as ARP Ethernet type. For TCP and UDP protocol types, the controller module creates flow modification messages incorporating the TCP and UDP source and destination ports.

Some of the implemented code for sending ARP flow modification message to the switch is reviewed below:

```
p_flow_mod = (struct ofp_flow_mod *) write_buffer;
p_flow_mod->header.version = 1;
p_flow_mod->header.length = htons(sizeof (struct ofp_flow_mod) +
  sizeof (struct ofp_action_output));
p_flow_mod->header.type = OFPT_FLOW_MOD;
p_flow_mod->header.xid = htonl (transaction_id);
p_flow_mod->match.in_port = htons(in_port);
p_flow_mod->match.wildcards = htonl(0);
memcpy (p_flow_mod->match.dl_src, src, sizeof(uint8_t) *
  OFP_ETH_ALEN);
memcpy (p_flow_mod->match.dl_dst, dst, sizeof(uint8_t) *
  OFP_ETH_ALEN);
p_flow_mod->match.dl_vlan = htons(65535);
p_flow_mod->match.dl_vlan_pcp = htons (0);
uint8_t op = ntohs (p_packet_in_arp->arp_op);
p_flow_mod->match.nw_proto = op;
p_flow_mod->match.nw_dst = htonl (ip_dst);
p_flow_mod->match.nw_src = htonl (ip_src);
p_flow_mod->match.dl_type = htons (ARP_DATA);
```

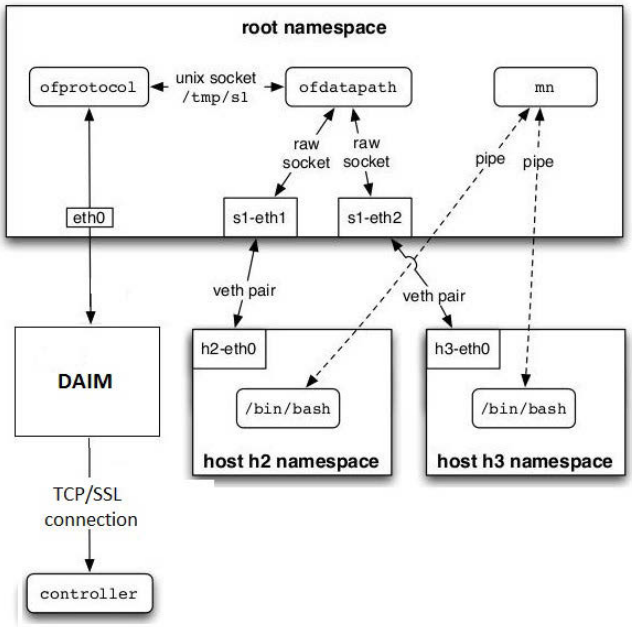


Figure 4.12.: DAIM Model Integration with Mininet

4.4. Setup Requirements for Testing DAIM

There are a few reasons taken into account when choosing to test the implemented DAIM model with Mininet. Firstly, Mininet is open and accessible to researchers and developers for SDN-based experiments. Secondly, Mininet provides a realistic setting for the SDN-based environment. Figure 4.12 shows the structure of Mininet with the integration of DAIM model phase 1 (subsection 4.2.1 on page 97). In this stage, DAIM only operates as a middlebox between the controller and the OpenFlow switch. More importantly, Mininet emulator includes a simple graphical user interface editor called MiniEdit. It is an experimental tool developed to illustrate how Mininet can be extended. MiniEdit is used to create a custom network, configure network elements, save the topology and run the simulation for testing DAIM. There are several significant requirements to conduct the scenarios for verifying the functionality of DAIM, which can be summarised below:

- **Apparatus Requirements**

Networking devices such as a software-based OpenFlow switch (OpenVswitch), host computers and an SDN controller that supports higher-level applications. An operating system such as Linux, which supports OpenFlow network model. A network emulation platform such as Mininet for testing and experimenting the functionalities of DAIM. A packet analyser software such as open-source Wireshark used for capturing network traffic and packets, troubleshooting and analysis. A programming language based on C/C++ to develop and compile the DAIM model. An integrated development environment (IDE) to ease and debug program coding.

- **DAIM System Requirements**

A network channel to communicate with the OpenFlow switches. A storage module to store information of dispersed network elements. A controller module to manage the installation of flow entries in the switch flow table. OpenFlow protocol version 1.0 to facilitate the communication between DAIM controller and the OpenFlow switch. Explicitly allocating one dedicated DAIM controller to connect and control only one OpenFlow switch.

4.4.1. Scenarios for Testing DAIM

This section presents the scenarios used to verify the functionality of the implemented DAIM controller. Using Mininet tool to test the system's functionality is considered as the primary goal. In terms of analysing the performance results correctly, the parameters and configurations such as links, bandwidth, delay, and network size for the testing topologies have been defined. Furthermore, MiniEdit is used as a simple user interface that presents a canvas with a row of tool icons on the left side of the window and a menu bar along the top of the window, which simplifies the creation of any custom network topology. There are a number of different types of network topologies that can be categorised into the following basic types:

- **Simple linear topology**

A linear network topology establishes a two-way link between one host and the next. In this content, it is a simple topology with two switches connected to each other with a link and has one host on each switch. Figure 4.13 shows a basic linear network topology used to test if an UDP port is open by creating a simple UDP server and client.

- **More complex network with ring and tree topology**

Each host in a ring network topology is connected to exactly two other hosts and forms a single continuous data path for signals through each host (a ring). Data travel from host to host with each host along the path processing every packet (see Figure 4.14). In contrast, a tree network topology is essentially a multiple level topology with N levels and two hosts per switch. Figure 4.15 shows a tree network topology created in Mininet for testing the DAIM controller.

- **Heavily interconnected network (full mesh topology)**

A mesh network topology has a direct link between all pairs of hosts, where each host relays data for the network. This scenario consists of four switches and four hosts cooperating in the distribution of data in the network (see Figure 4.16). A mesh network can relay messages by either using a routing technique or a flooding technique. Because there are at least two hosts with two or more paths between them to provide redundant paths, the mesh topology is used to test the shortest path calculation of DAIM.

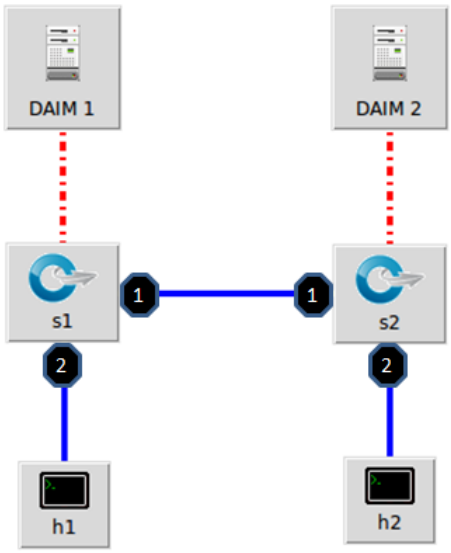


Figure 4.13.: Simple Linear Topology Setup

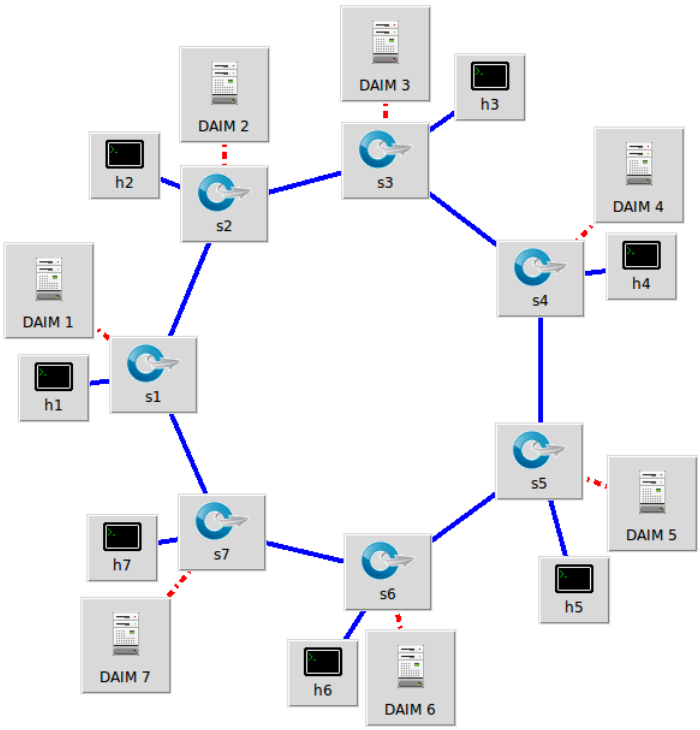


Figure 4.14.: Ring Network Topology Setup

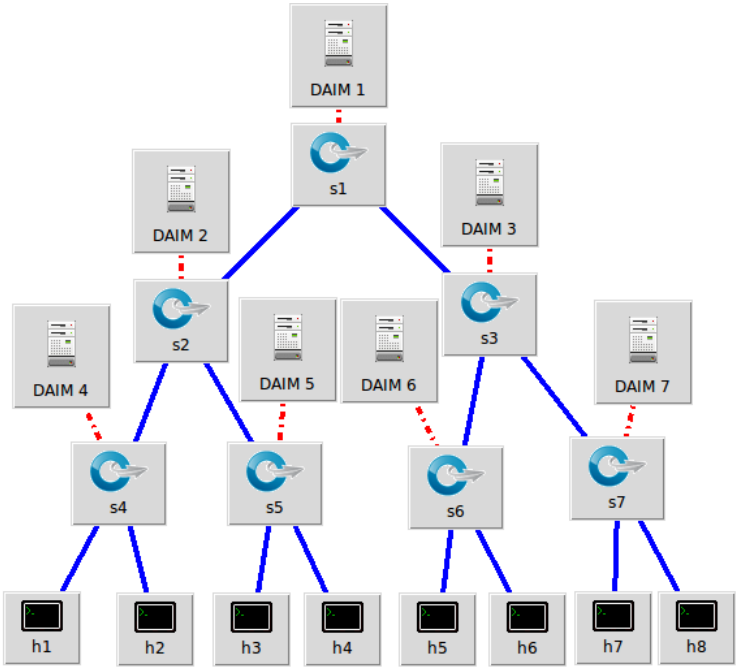


Figure 4.15.: Tree Network Topology Setup

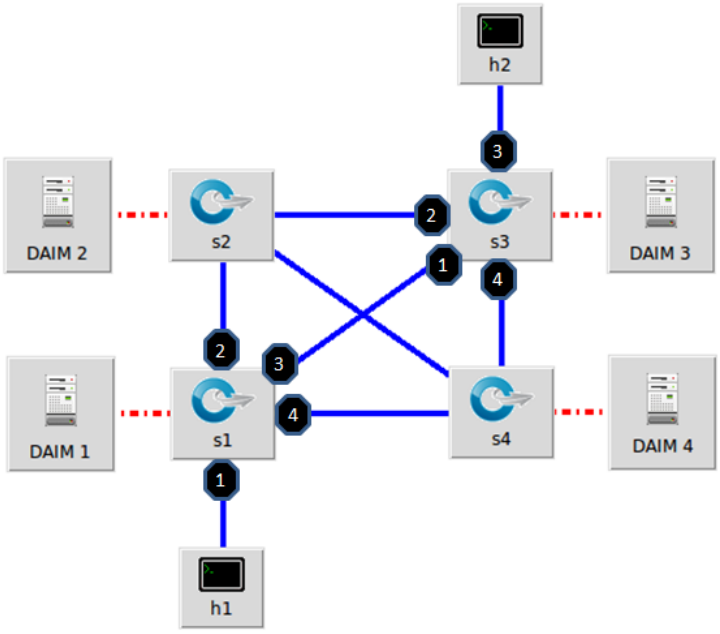


Figure 4.16.: Fully Mesh Network Topology Setup

4.5. DAIM System Validation

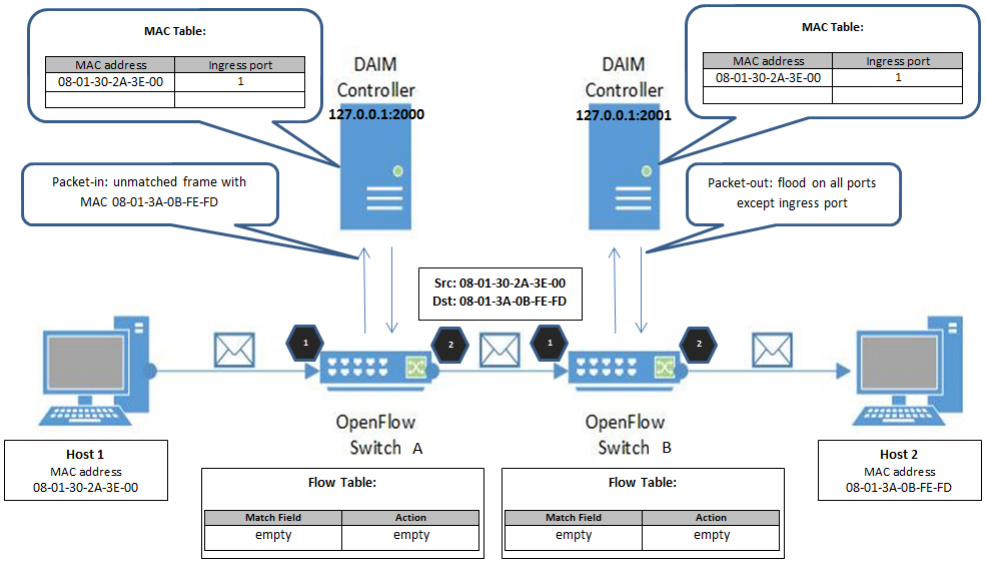
To validate the DAIM controller, a set of actions are used to check the compliance of the overall system with its purpose and functions. For example, using the ping command between hosts, creating Linux command line chat server, streaming video via VLC media player, and running a simple web server and client to verify network connection can be achieved. This section describes a simple communication scenario of the DAIM system in an OpenFlow-enabled network. It also focuses on a basic OpenFlow setup to test that DAIM operates exactly as it is designed to do in a consistent and functional manner, and to demonstrate the underlying protocol in action through various network services and applications. The objective of DAIM validation is to produce documented evidence, which provides a high degree of assurance that all components of the system will consistently work correctly when brought into use.

4.5.1. Communication Example

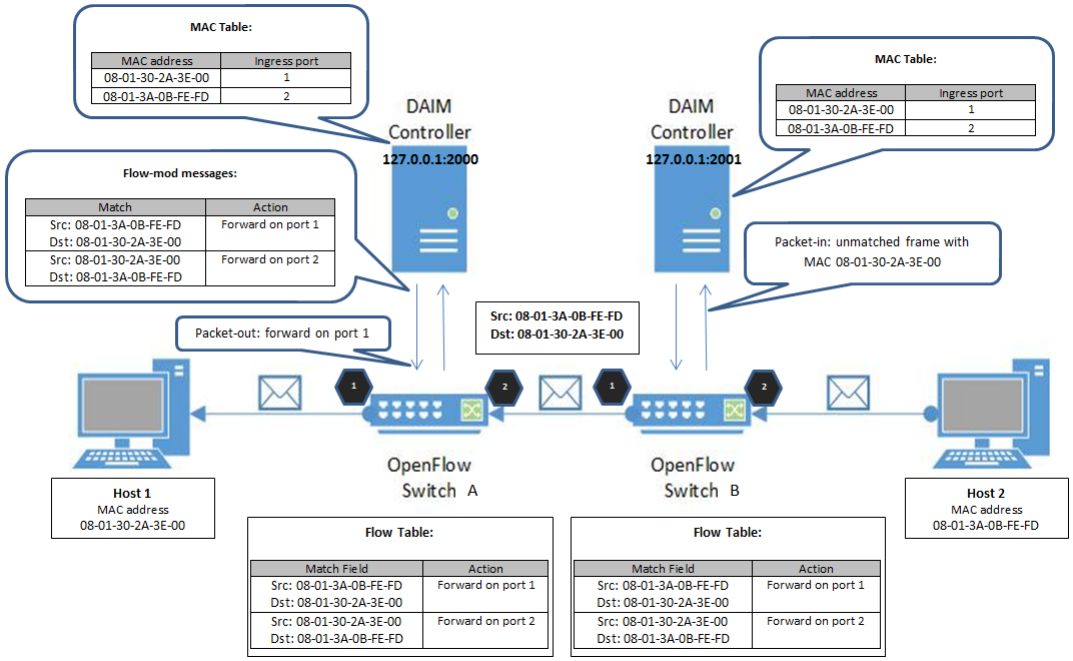
The following example is a basic OpenFlow network, which consists of two OpenFlow switches, two DAIM controllers and two hosts. The DAIM controllers in this example implement an intelligent L2 learning switch behaviour. Switch A is connected to DAIM via loopback port 2000, host 1 via port 1 and switch B via port 2. In contrast, switch B is connected to DAIM via loopback port 2001, host 2 via port 2 and switch A via port 1. In addition, the corresponding MAC addresses are presented for both hosts. In Figure 4.17a, host 1 (MAC address *08-01-30-2A-3E-00)* would like to send a packet to host 2 (MAC address *08-01-3A-0B-FE-FD)*. When the data packet from host 1 arrives at switch A, which has no flow entry yet, it thus sends an OFPT_Packet_In message to its DAIM controller. The controller determines the action that should be applied for this first flow and how it is handled. In this example, the DAIM controller implements L2 switch behaviour and hence answers with an OFPT_Packet_Out message with an action to flood on all ports of switch A except the ingress port. The data packet gets sent to switch B, which also has no flow entry yet. Therefore, the signalling process between switch B and its DAIM controller is repeated. Finally, the packet arrives at host 2.

Furthermore, the source MAC address and the ingress port are mapped and learned by the DAIM controller. However, there is no flow entry installed in the switch's flow table because the destination MAC address has not been learned yet. If the mapping of both addresses has been learned, the flow entries are then installed via the *flow-mod* messages.

Figure 4.17b shows the reverse direction for this communication. Host 2 now forwards a reply packet to host 1 which again arrives at switch B. Either the packet header or the entire packet gets sent to the connected DAIM controller as there is still no flow entry installed in the flow table. This controller learns the mapping



(a) First Packet in Flow from *Host* 1 to *Host* 2



(b) Response Packet from *Host* 2 to *Host* 1

Figure 4.17.: Communication between two nodes in an OpenFlow network managed by DAIM.

from source MAC address to the ingress port and confirms the mapping for the destination MAC address. Hence, both mappings are established, and the dedicated DAIM controllers now use the *flow-mod* messages to install the corresponding flow entries in the flow tables of switch A and B. Both DAIM controllers trigger the forwarding of data packet on the same port 1 via *packet-out* messages to switch A and B. Because the flow entries are installed in the flow tables, similar packets of this flow can be sent without the vetting from the DAIM controllers.

4.5.2. Flow Table Buildup with Example of Ping Traffic

This setup consists of four switches with a fully mesh connection between them, and each switch is reachable locally by its own DAIM controller on the management network (see Figure 4.16 on page 122). In order to show a simple flow installation on an OpenFlow switch, a ping example has been executed where *host 1* would ping to *host 2*. The process is explained by showing how the data and control packets are sent to each host. This is done by sending an ARP request/response and ICMP echo request/response between the two ends. The ping command sends ICMP Echo_Request from *host 1*. The ping utility of *host 2* uses the ICMP protocol's mandatory Echo_Request datagram to elicit an ICMP Echo_Response. Both hosts have Echo_Request datagrams (pings) that contain an IP and ICMP header, followed by a "struct timeval" and then an arbitrary number of "pad" bytes used to fill out the packet. If the results show a series of replies, the connection is working properly. The time indicates how fast the connection is. If there is a "timed out" error instead of a reply, there is a breakdown somewhere between the hosts.

In this example, DAIM implements a normal learning switch, which learns the MAC address to port binding by analyse incoming packets and look up at the source address. It uses this information to forward incoming packets, and if the destination is unknown, the packet will be flooded.

The tables below are defined as follows:

Data	Ctrl	Source	Destination	Content
Sequence number of packet sent between hosts. If marked with a,b,c.. this means it is the same packet but in a different stage of forwarding.	Sequence number control packets sent between OpenFlow switch and DAIM controller.	Source of packet	Destination of packet	Description of the content of the packet.

When a ping command is executed from *host 1* to ping *host 2* on 10.0.0.2, the mechanism to forward packets with an unknown destination is demonstrated by the process of an ARP request.

ARP request

Data	Ctrl	Source	Destination	Content
1a		Host 1	Broadcast	ARP: Who has 10.0.0.2?

A normal ARP request is generated by host 1 to broadcast MAC address.

	1	Switch 1	DAIM 1	OFPT encoded packet 1 (ARP request)

Switch 1 has no flows yet and does not know how to forward this packet. It sends a packet-in that will have the ARP request encapsulated in it to DAIM 1 controller.

	2	DAIM 1	Switch 1	Flood packet out all ports (except ingress port)

DAIM 1 controller responds by directing switch 1 to send a packet-out to all edge switches with an action to flood this packet out of all ports (except ingress port).

1b		Switch 1	Flood	ARP: Who has 10.0.0.2 ?

Switch 1 indeed floods the packet. This process is then repeated exactly the same way for switch 2, switch 3 and switch 4 until the packet reaches its destination (host 2).

	3	Switch 3	DAIM 3	OFPT encoded packet 1 (ARP request)
	4	DAIM 3	Switch 3	Flood packet out all ports (except ingress port)

As switch 3 floods the packet, it is received by host 2.

1c		Switch 3	Flood	ARP: Who has 10.0.0.2 ?

ARP reply

2a		Host 2	Host 1	ARP: I'm 10.0.0.2 my MAC is *08-01-3A-0B-FE-FD*

Host 2 is now aware of host 1 MAC address and sends an ARP reply back.

	5	Switch 3	DAIM 3	OFPT encoded packet 2 (ARP reply)

Switch 3 does not know how to forward the packet and sends an ARP reply in the form of packet-in to DAIM 3 controller.

	6	DAIM 3	Switch 3	Flow-mod: Prot: ARP, Src *08-01-3A-0B-FE-FD,* Dst *08-01-30-2A-3E-00* → IN = 3, OUT = 1

DAIM 3 controller knows where the destination MAC address locates in the network, and hence instructs switch 3 to install a flow and on how to forward similar packets in the future.

2b		Switch 3	Switch 1	ARP: I'm 10.0.0.2 my MAC is *08-01-3A-0B-FE-FD*

Switch 3 sends the packet to switch 1.

	7	Switch 1	DAIM 1	OFPT encoded packet 2 (ARP reply)

The process is repeated, DAIM 1 controller knows where the destination MAC address locates in the network, and hence instructs switch 1 to install a flow and on how to forward similar packets in the future.

	8	DAIM 1	Switch 1	Flow-mod: Prot: ARP, Src *08-01-3A-0B-FE-FD,* Dst *08-01-30-2A-3E-00* → IN = 3, OUT = 1

Finally, the packet is delivered to host 1.

2c		Switch 1	Host 1	ARP: I'm 10.0.0.2 my MAC is *08-01-3A-0B-FE-FD*

Host 1 can populate its ARP table after receiving ARP reply from *host 2*. Now that *host 1* has the ARP address of *host 2*, it will send the ICMP echo request.

ICMP echo request

Data	Ctrl	Source	Destination	Content
1a		Host 1	Host 2	ICMP echo request

Host 1 sends a ICMP echo request packet to host 2.

	1	Switch 1	DAIM 1	OFPT encoded packet 1 (ICMP echo request)

Again, switch 1 does not know how to forward the packet and sends an ICMP echo request in the form of packet-in to DAIM 1 controller.

	2	DAIM 1	Switch 1	Flow-mod: Prot: ICMP, Src 10.0.0.1, Dst 10.0.0.2 → IN = 1, OUT = 3

During the ARP session, DAIM 1 controller has learned where the destination host exists and can now direct switch 1 to install a flow for this specific traffic.

1b		Switch 1	Switch 3	ICMP echo request

Switch 1 sends the packet to switch 3.

	3	Switch 3	DAIM 3	OFPT encoded packet 1 (ICMP echo request)

For the incoming packet, switch 3 also sends an ICMP echo request in the form of packet-in to DAIM 3 controller.

	4	DAIM 3	Switch 3	Flow-mod: Prot: ICMP, Src 10.0.0.1, Dst 10.0.0.2 → IN = 1, OUT = 3
1c		Switch 3	Host 2	ICMP echo request

Again, the DAIM 3 controller instructs the switch 3 to install a flow for traffic forwarding.

The reply of the ICMP packet will be sent in the same way as the request packet. The entries are not annotated in this table for better readability.

ICMP echo response

2a		Host 2	Host 1	ICMP echo reply
	5	Switch 3	DAIM 3	OFPT encoded packet 2 (ICMP echo reply)
	6	DAIM 3	Switch 3	Flow-mod: Prot: ICMP, Src 10.0.0.2, Dst 10.0.0.1 → IN = 3, OUT = 1
2b		Switch 3	Switch 1	ICMP echo reply
	7	Switch 1	DAIM 1	OFPT encoded packet 2 (ICMP echo reply)
	8	DAIM 1	Switch 1	Flow-mod: Prot: ICMP, Src 10.0.0.2, Dst 10.0.0.1 → IN = 3, OUT = 1
2c		Switch 1	Host 1	ICMP echo reply

4.5.3. Creating a Linux Command Line Chat Server

Many people have been using chat services for a long time such as the popular Google Chat, Hangout, Facebook chat, Whatsapp, Hike and several other application and integrated chat services. One of the Linux command line tools used to send data between networked computers is netcat (nc). This Linux nc command can turn a Linux box into a chat server with just one line of command, and has the ability to use any unused local port and any local network source address. The nc utility is often known as Swiss army knife because of the number of its built-in capabilities. Netcat is prominently used for port scanning, file transferring, port listening or media streaming. It is a powerful network utility used as an investigation tool, debugging tool, reading and writing data to network connections using TCP or UDP protocol. Netcat is used to create a server and client chat system and test the TCP/IP connection between two hosts.

This subsection demonstrates a simple UDP chat session based on a linear topology presented in Figure 4.13 on page 121, where the text from one terminal can be echoed to the remote terminal. There are two instances of netcat required in order to create this chat: one to listen for incoming connections (the server); and another one to start the connection. Use nc command (on a server with IP address: 10.0.0.1) to create a command line messaging server instantly.

```
1 $ nc -l -u -v -p 6871
```

Explanation of the above command switches.

-l : instructs netcat to listen

-u : specifies to specifically use UDP

-v : is for verbose output

-p : the local port number

You may replace 11119 with any other local port number.

Next on the client machine (IP address: 10.0.0.2) run the following command to initialise chat session to machine (where messaging server is running).

```
1 $ nc -u 10.0.0.1 6871
```

Any text that is entered on one screen is echoed to the other screen over the UDP port specified. Figure 4.18 shows how it will look on the screen. A useful utility like *dpctl* that comes with the OpenFlow reference distribution is used to enable visibility and control of the switch's flow table. It is particularly useful for verifying that all necessary flow entries are properly installed by viewing flow state and flow counters.

Note that you can terminate chat session by hitting the ctrl+c key and also nc chat is one-to-one service.

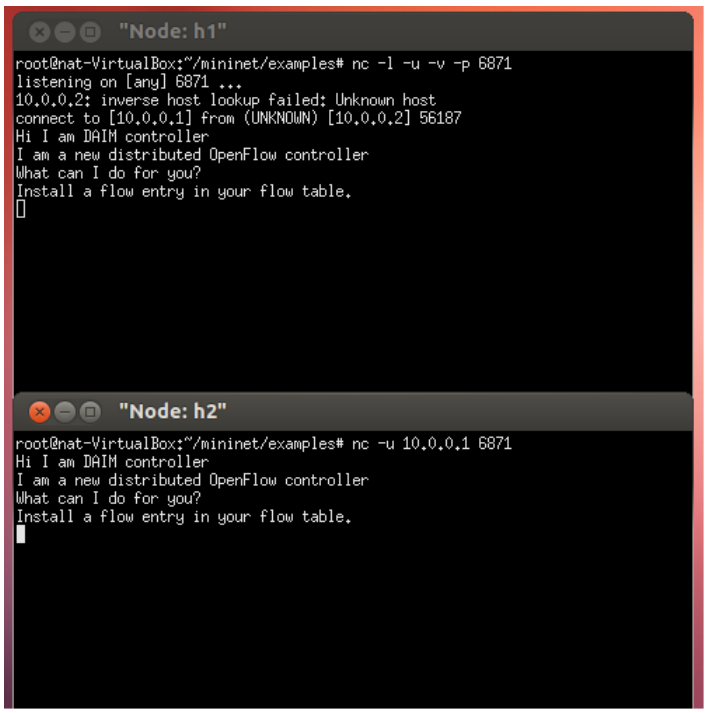


Figure 4.18.: Screenshot of netcat UDP Chat Session

After running a *dpctl dump-flows* from *switch 1*, the flow stats and flow entries installed by DAIM controller can be seen below:

```
1 ***s1
    ------------------------------------------------------------------------
    NXST_FLOW reply (xid=0x4):
2 cookie=0x0,duration=5.722s,table=0,n_packets=1,n_bytes=55,
    idle_timeout=60,priority=65535,udp,in_port=2,vlan_tci=0x0000,dl_src
    =c2:ee:c7:4b:d8:0f,dl_dst=66:17:bc:fe:87:3d,nw_src=10.0.0.1,nw_dst
    =10.0.0.2,nw_tos=0,tp_src=6871,tp_dst=41033 actions=output:1
3 cookie=0x0,duration=20.392s,table=0,n_packets=1,n_bytes=53,
    idle_timeout=60,priority=65535,udp,in_port=1,vlan_tci=0x0000,dl_src
    =66:17:bc:fe:87:3d,dl_dst=c2:ee:c7:4b:d8:0f,nw_src=10.0.0.2,nw_dst
    =10.0.0.1,nw_tos=0,tp_src=41033,tp_dst=6871 actions=output:2
4 cookie=0x0,duration=0.702s,table=0,n_packets=1,n_bytes=42,
    idle_timeout=60,priority=65535,arp,in_port=1,vlan_tci=0x0000,dl_src
    =66:17:bc:fe:87:3d,dl_dst=c2:ee:c7:4b:d8:0f,nw_src=10.0.0.2,nw_dst
    =10.0.0.1,arp_op=0 actions=output:2
```

As can be seen, the flow entries are installed using UDP protocol type with the corresponding actions. Moreover, an *idle_timeout* option for each entry is configured, which defaults to 60 seconds. This indicates that the flow will expire after 60 seconds if there is no incoming traffic. ARP sessions are also completed in order to populate the ARP tables.

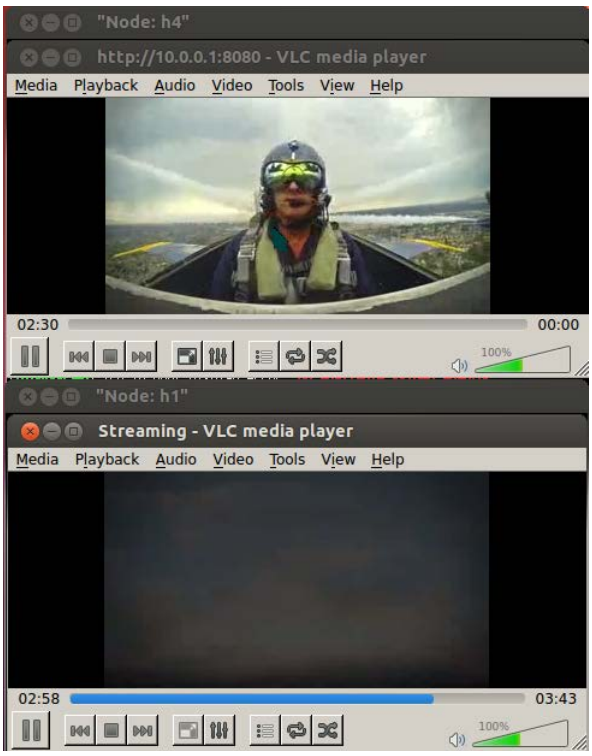


Figure 4.19.: Screenshot of VLC Video Streaming Session

4.5.4. Network Streaming via VLC Media Player

When testing OpenFlow networks and flow forwarding, it is useful to be able to send a number of multicast streams across the network. In this part, a VLC streaming session is performed to demonstrate that the DAIM controller can manage traffic through each switch in the network based on data and by setting up and removing flows from the switch. VLC is an open source cross-platform multimedia player and streamer. Streaming traffic from the server to client through OpenFlow is conducted based on the ring network topology presented in Figure 4.14 on page 121. The streaming is done via HTTP protocol on port 8080. This streaming method is used with the MP4 encapsulation. For the output method, the IP address and TCP port number on which to listen has been specified. To stream in HTTP:

- on the server (*host 1*) run:

```
vlc -vvv input_stream --sout '#standard{access=http,mux=ogg,dst
  =10.0.0.1:8080}'
```

- on the client (*host* 4) run:

```
vlc http://10.0.0.1:8080/the_file.mpg
```

Figure 4.19 shows how it will be seen on the screen. VLC is used to stream a local video file from *host 1* to *host 4*. In this setup, the DAIM controller can add the

respective flows which allow the client to receive the video successfully. Thus, the DAIM controller can provide efficient support for scalable video streaming over an OpenFlow network. Also, the ping has an RTT average of 4-5 ms. due to the dynamic routing capability using the shortest path. To verify that the flow entries are being implemented correctly, *dpctl dump-flows* is executed to print OpenFlow flow table entries of *switch 3* below:

```
1 ***s3 ----------------------------------------------------------
  NXST_FLOW reply (xid=0x4): cookie=0x0,duration=36.53s,table=0,
  n_packets=962,n_bytes=2097023,idle_timeout=60,priority=65535,tcp,
  in_port=3,vlan_tci=0x0000,dl_src=ca:8e:3b:a2:f0:33,dl_dst=86:85:db:
  ec:eb:0e,nw_src=10.0.0.1,nw_dst=10.0.0.4,nw_tos=0,tp_src=8080,
  tp_dst=50368 actions=output:2
2 cookie=0x0,duration=36.539s,table=0,n_packets=612,n_bytes=40530,
  idle_timeout=60,priority=65535,tcp,in_port=2,vlan_tci=0x0000,dl_src
  =86:85:db:ec:eb:0e,dl_dst=ca:8e:3b:a2:f0:33,nw_src=10.0.0.4,nw_dst
  =10.0.0.1,nw_tos=0,tp_src=50368,tp_dst=8080 actions=output:3
```

For *switch 3*, these flow entries show that TCP is used as the transmission protocol, which VLC listens on all the network interfaces of the server on port 8080. In addition, TCP guarantees delivery of data packets on port 50368 in the same order in which they were sent. DAIM controller also instructs the switch how to forward the packets using IP addresses and associated actions.

4.5.5. Run a Simple Web Server and Client

Ping is not the only command in Mininet that can run on a host. Virtual hosts can run any command or program that is available to the underlying Linux system (or VM) and its file system. Users can also enter any bash command, including job control (*e.g.*, jobs, kill, etc.)

This subsection creates a simple HTTP web server on host 1 and makes a request from host 8 based on a tree network topology presented in Figure 4.15 on page 122. This can verify that in an OpenFlow network with DAIM controllers, files from the web server can be consistently transferred to the requested host. The following command starts a web server on one host and makes an HTTP request from another:

```
1 mininet> h1 python -m SimpleHTTPServer 80 &
2 mininet> h8 wget -O - h1
```

Figure 4.20 illustrates how this will look on the screen. The Mininet CLI allow users to control and manage the entire virtual network from a single console. After using the *dpctl dump-flows* command built into the CLI, multiple flow entries can be seen in switch 1, 2, 3, 4 and 7 of the forwarding path. However, nothing will print out for switch 5 and 6 because they are not on the path and thus these switches have no flows added. Some sample flow entries installed in switch 1 by DAIM controller are as follows:

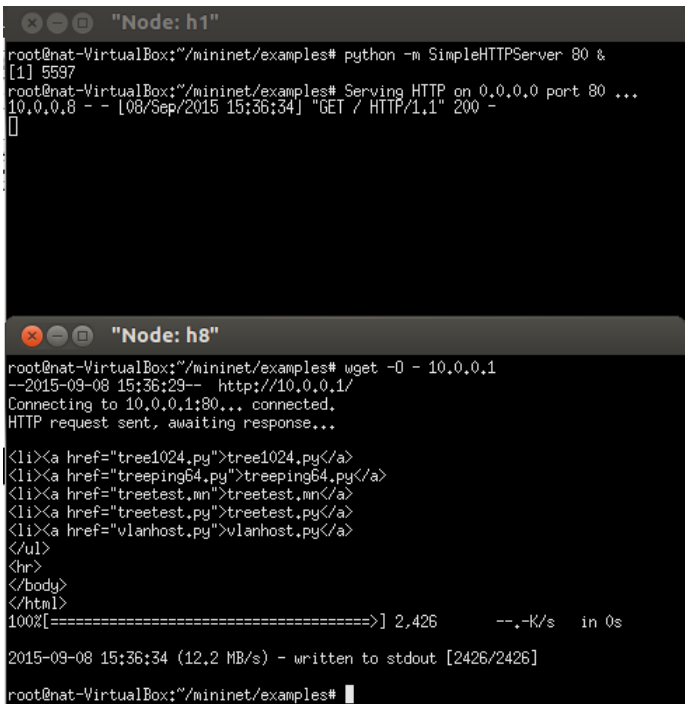


Figure 4.20.: Screenshot of HTTP Web Server Session

```
1 ***s1
    ----------------------------------------------------------------
    NXST_FLOW reply (xid=0x4): cookie=0x0,duration=11.892s,table=0,
    n_packets=12,n_bytes=3381,idle_timeout=60,priority=65535,tcp,
    in_port=1,vlan_tci=0x0000,dl_src=d6:6e:39:f2:fa:d4,dl_dst=a6:e2:5b:
    a0:df:25,nw_src=10.0.0.1,nw_dst=10.0.0.8,nw_tos=0,tp_src=80,tp_dst
    =41391 actions=output:2
2 cookie=0x0,duration=11.898s,table=0,n_packets=12,n_bytes=908,
    idle_timeout=60,priority=65535,tcp,in_port=2,vlan_tci=0x0000,dl_src
    =a6:e2:5b:a0:df:25,dl_dst=d6:6e:39:f2:fa:d4,nw_src=10.0.0.8,nw_dst
    =10.0.0.1,nw_tos=0,tp_src=41391,tp_dst=80 actions=output:1
```

For switch 1, these installed flow entries can verify that the server host (IP 10.0.0.1) can communicate with the requested host (IP 10.0.0.8) via TCP protocol port 80 and 41391. Incoming packets on port 1 are set with an action to forward out port 2, whereas packets on port 2 will be sent out port 1.

Lastly, shut down the web server after data transfer session with the following command:

```
1 mininet> h1 kill %python
```

5. DAIM Performance Results and Evaluation

5.1. Introduction

This chapter explains the technical details and testing methodology of the experimental evaluation. The performance analysis of the DAIM model is also conducted based on OpenFlow Operations Per Second (OFLOPS) open benchmarking tool [88]. In addition, a built-in utility called Cbench (controller benchmarker) is used as a generic software framework that allows the development of tests for the DAIM controller. It can measure the network capabilities and bottlenecks between the remote control application and the forwarding engine of an OpenFlow switch. Cbench is often considered as a standard evaluation tool for OpenFlow controller performance. It is a useful tool for evaluating current controller implementations that make the following observations: (1) the performance of flow setup depends on applied actions and firmware; (2) current OpenFlow implementations differ substantially in flow updating rates as well as traffic monitoring capabilities; (3) accurate OpenFlow command completion can be observed only in the data plane. These observations are crucial for understanding the applicability of OpenFlow in the context of specific use-cases, which have requirements in terms of forwarding table consistency, flow setup latency, flow space granularity, packet modification types, and traffic monitoring abilities.

In order to quantify DAIM controller performance, the ping command is executed to measure the average RTT values. Moreover, the Cbench tool is used to measure the number of flow setups per second that the DAIM controller can handle in terms of throughput and latency. At the same time, Iperf is served as a benchmarking tool for testing the network's maximum TCP bandwidth utilisation. The performance results of DAIM are compared with other OpenFlow controllers including NOX, POX and NOX-MT. The evaluations illustrate that the DAIM controller produces comparable results to those existing ones. The physical OpenFlow test lab, as well as other parameters that can affect the performance evaluation, are also described in this chapter.

5.2. Test Bed Description

The main target of this chapter is to give a dumping ground for side by side comparisons between multiple OpenFlow controllers and the candidate DAIM controller. In the SDN community, the focus has largely been on the control plane and has seen a rise of numerous controllers in different languages (C, C++, Python, Java and Ruby for example). Furthermore, while many times raw performance numbers are published, currently there has been no central repository for performance comparisons using the same methodology. The performance comparisons only cover open source controllers due to their primary usage in research and development; hence they tend to be single instances. Based on the study of available materials on sixteen SDN controllers, the following three controllers are tested:

NOX: A C++/Python controller used as a baseline and does not support multi-threading. It can only be tested in the single CPU case.

POX: A general single-threaded, open source OpenFlow controller written in Python.

NOX-MT: A multi-threaded and highly optimised implementation of NOX written in C++, which can be found in the destiny-fast branch on NOX repo.

For the evaluation of mean RTT, the default configuration for the traffic control of links between nodes is 100 Mbps for bandwidth and 5 ms for propagation delay. Each ping command is performed in Mininet with various packet sizes in order to obtain accurate and reliable results. Mininet is also used to measure the maximum TCP bandwidth utilisation of the network by running Iperf. Cbench is an evaluation tool for the DAIM controller performance. For all experiments, TCMalloc is used for better performance in C/C++, and test setups were run using as follows:

- Machine: Dell Optiplex 780

  - CPU: 1 x Intel® Core(TM)2 Duo CPU E8400 @3.00 GHz 3.00 GHz

  - RAM: 4.00 GB

  - Network: Intel(R) 82567LM-3 Gigabit ports

  - OS: Ubuntu 12.04.5 LTS x86_64 (VirtualBox image)

  Kernel: 3.5.0-54-generic #81~precise1-Ubuntu SMP Tue Jul 15 04:05:58 UTC 2014 i686 i686 i386 GNU/Linux

  Boost Library: Version 1.48.0.2 (libboost-all-dev)

  malloc: Google's Thread-Caching Malloc version 1.7-1ubuntu1

  Java version: Sun Java 1.6.0_35

- Controller configuration:

  NOX (destiny 0.9.1): must be configured with "–enable-ndebug" passed to the configure script.

NOX MT: must be configured with "–enable-ndebug –with-python=no" passed to the configure script. tcmalloc loaded before launch export

LD_PRELOAD=/usr/lib/libtcmalloc_minimal.so.0

POX: must be configured with *git checkout dart*, dart is the most recent release branch (Until July 2014).

- Network application used: Layer 2 learning switch functionality. This switch application is well representative of the controller flow processing performance with tunable read and write ratio (*e.g.*, number of unique MAC addresses).
- Running controllers: disable debugging and verbose output mode.

NOX:

```
./nox_core -i ptcp:6633 switch
```

NOX MT

```
./nox_core -i ptcp:6633 switch -t 1
```

POX:

```
./pox.py forwarding.l2_learning
```

DAIM:

```
./aniv21 -c -q -p 6633
```

For all experiments, each controller implements a normal L2 learning switch application provided by the controller. For every switch on the chosen path, the switch application performs MAC address learning. The packets get sent out of the last port where the traffic from the destination MAC address has arrived. Packets with an unspecific destination are flooded. Regarding the switches managed by NOX, the mapping between the port number and the MAC switch tuple is stored in a hash table. The switch application is mainly read-only workload. This means that only requests with newly observed source MAC address trigger an update in the hash table. In addition, the number of such events is proportional to the product of the number of switches and the number of hosts in the network.

5.3. Experiment Setup and Methodology

Cbench is specifically used to quantify controller performance with respect to throughput and latency by generating Packet-In events for new flows. It is an OpenFlow testing platform that primarily focuses on providing a set of basic measurement tests for developers to examine the capabilities of OpenFlow-enabled devices as well as detect any possible bottleneck. Cbench emulates any number of OpenFlow switches

that connect to a controller, send Packet_In messages, and watch for Flow_Mods to get pushed down. Cbench gives the ability to measure various performance aspects of the OpenFlow controller such as minimum and maximum response time, maximum throughput, and the latency and throughput of the controller with a bounded number of packets. Using Cbench, the flow setup throughput and latency are evaluated according to three publicly available OpenFlow controllers including NOX, POX and NOX-MT. The latest available version of each controller is used for the evaluation as of May 2015.

The Cbench operation is very simple, pretending to be N switches that create N OpenFlow sessions to the controller. Cbench can run in two modes: throughput and latency mode. In latency mode, each emulated switch maintains exactly one outstanding new flow request, waiting for a response before soliciting the next request. Latency mode measures the OpenFlow controller's request processing time under low-load conditions. In contrast, the throughput mode has each switch maintaining as many outstanding requests as buffering will allow until the local TCP send buffer blocks. Therefore, throughput mode measures the maximum flow setup rate that a controller can maintain. Cbench also supports a hybrid mode with n-new flow requests outstanding, to explore between these two extremes. A simple algorithm that Cbench used to measure the throughput and latency is shown below:

```
Algorithm:
    pretend to be an OpenFlow switch that can
    create number of unique source MAC addresses to the controller
    if latency mode (default):
        for each session:
            1) send up a Packet_In
            2) wait for a matching Flow_Mod to come back
            3) repeat
            4) count how many times #1-3 happen per sec
    else in throughtput mode (with '-t' option):
        for each session:
            while buffer not full:
                queue Packet_Ins
                count Flow_mods as they return back
```

Each Cbench run consists of 10 test loops of 10 seconds duration, each is run three times and takes the average number as the result. The first loop (first 10 seconds) is considered as controller warm-up, and hence the results are not counted. The total responses received from each test are averaged to compute the responses per second result. In addition, the last 8 tests results are averaged to compute the final result. Because DAIM controller must be connected to only one switch, a fixed number of one emulated switch is tested to avoid a large number of traffic to be stressing the network simultaneously. 82 byte sized OpenFlow Packet_In messages are used to ensure that the bandwidth is not the bottleneck as well as to maximise the stress on the controller end. The latency is also measured with one switch, which sends requests that have to wait for a reply from the controller before sending the next

request.

Furthermore, Iperf is a benchmark tool used for active measurements of the maximum achievable TCP bandwidth on IP networks, allowing the tuning of various parameters and UDP characteristics. For each test, Iperf can report the bandwidth, delay jitter, loss and other parameters. However, Iperf tool in this context is merely used to measure maximum TCP bandwidth utilisation.

To measure throughput and latency, the Cbench tool is used to test and compare the performance of a system or subsystem, different computer architecture and different software. The benchmark test consists of executing a computer program or a set of programs with the aim of evaluating the performance of a system. It is necessary to define and explain the network performance parameters, definitions needed for a correct evaluation and to successfully perform the evaluations.

5.3.1. Network Performance Metrics

Performance evaluation of an OpenFlow controller is defined by four characteristics: round-trip-time, throughput, latency and bandwidth. The aim is to obtain the lowest average RTT, maximum throughput (number of outstanding packets, flows per second), minimum latency (response time, ms), and maximum TCP bandwidth (the rate of data transfer) for each controller. These main four parameters are more important and determinants for the performance of any application. The results show that DAIM performance tests are comparable to the results of the current OpenFlow controllers and that it is possible to run DAIM on conventional hardware.

5.3.1.1. Mean Round Trip Time

The first performance metric for the evaluation is the mean RTT for the hosts in the investigated network. The average RTT is measured from the received echo reply messages. The experimental results will need the recording of a large number of samples. To describe the measurement of the numbers, the standard deviation and mean value of RTT are computed. The mean of the RTT is assumed to be the average value and the standard deviation is approximated using the following equation [17]:

$$T = N/R \tag{5.1}$$

Where T is the packet transmission time between segments sent and receives an acknowledgement arrival measured in millisecond (round trip transmission), N is the packet size (bits), and R is the data rate bandwidth (bit/second).

Then we need to calculate the round trip time RTT by the following equation:

$$\triangle \;=\; \alpha \times \tilde{\triangle} + (1-\alpha) \times T \tag{5.2}$$

Where $\triangle$ is the round trip time RTT (ms), α is the smoothing factor, $\tilde{\triangle}$ is the old round trip time RTT, and T is the packet's new round trip transmission time.

Test methodology: The mean RTT values are measured from the ping tool by α (value between 0 and 1) which is equal to the value 0.875 multiplying with the old RTT as the top equation, and then multiplying with T which is the new RTT between the segments sent and the acknowledgement arrival found from equation *(5.1)*. Finally, the mean RTT values are calculated from equation *(5.2)*. Each run is repeated 6 times with different random number seeds to exclude simulation artefacts. Each ping command is executed with increasing packet sizes of 56, 4000, 8000, 16000, 32000 and 65000 bytes.

5.3.1.2. Flow Setup Throughput

Each controller throughput is a significant factor in deciding the overall number of controllers required to handle the network control load. The goal of this performance parameter is to determine the maximum throughput of the system in various settings. Flow setup throughput is the number of flow transactions per second that an OpenFlow controller can handle. Measuring this parameter can come in two distinct forms, one would be using bits per second (bit/s or bps), and the other would be through data packets per second (p/s or pps) to measure the throughput performance. However, throughput in this content is measured in responses per second (rps) for which a larger value is better.

Test methodology: All experiments are performed with Cbench in order to take the maximum throughput measurements. The changes in the parameter when adding more hosts to the network where the controller runs show the scalability of the controller. The performance analysis is based on the correlation between controller's throughput and the number of unique hosts (10^3, 10^4, 10^5, 10^6, 10^7), having a fixed number of one simulated OpenFlow switch. For each session, the switch constantly forwards as many Packet_In messages as possible to the controller assuring that the controller always has messages to process, and then counts the Flow_Mod messages reply.

Running the benchmarker:

```
1 cbench -c $ctrladdr -p $port -s $nswitch -m $duration -l $loop -M
    $maddr -t
```

where $ctrladdr and $port are controller IP address and port number respectively, $nswitch is the number of emulated switches, $duration is the duration of the test, $loop is the number of times to repeat the test, and $maddr is the number of unique source MAC addresses. The *-t* option is for running the throughput test, omit it for the latency test.

5.3.1.3. Flow Setup Latency

Flow setup latency is a significant metric in OpenFlow network operation, particularly for transiting data from applications to delay. In regards to flow-based network, latency is the time delay to respond to flow requests. The delay can be caused by the physical environment or devices forming the network (switches and controller). For example, the first packet sent to the controller may increase the latency by at least the round trip to the controller. There are two ways to measure the latency: the first is to measure the time it takes for a packet from source to destination (One-Way) and the second is to measure the time it takes a packet to go and return (Round-Trip). This last form of measuring the latency, Round-Trip, is the most used because it allows a single device to measure the latency of a network.

For OpenFlow networks, 50 ms or less latency is required for a high-quality VoIP connection. Slightly higher latency can lead to some drops of frames, which could be admissible while maintaining a usable connection. In particular when streaming videos, less than 5% of some loss is acceptable for most codecs. If the frames are buffered, then 150 ms of average latency is adequate as well as a 5 seconds delay might also be acceptable. However, interactive videos can have a higher requirement to lost frames. Despite losses that are less than 1% are still acceptable, jitter on the other hand can have much more impact.

Test Methodology: The latency test is measured as the average time the controller takes to process each event in microseconds (s). Cbench is used to emulate one switch, which each session sends a single packet to the controller, and waits for a reply in a Flow_Mod. These two steps are repeated as quickly as possible and the total number of responses received at the end of a session can be used to compute the average time the controller took to process each event. Cbench is running via loopback locally, which emulates an OpenFlow switch and sends Packet_In events to the controller.

Running the benchmarker:

```
1 cbench -c localhost -p 6633 -m 10000 -l 10 -s 1 -M 100000
```

5.3.1.4. Maximum TCP Bandwidth Utilisation

While other network metrics are used to measure the performance of the DAIM controller, Iperf is written in C as a modern alternative for measuring TCP and UDP

bandwidth performance of IP networks. Iperf has a server and client functionality that can measure the throughput between two nodes. The output of Iperf typically contains the throughput measured and a time-stamped report of the amount of data transferred.

Iperf features:

TCP: Iperf can be used to test TCP capacity, which measures the throughput of the payload. It uses 1024 × 1024 for megabytes and 1000 × 1000 for megabits.

UDP: Iperf can be used to test UDP capacity, which allows the user to set the datagram size and provides results for the datagram throughput and the packet loss.

Test Methodology: Iperf [50] tool is used to create TCP data streams and measure the throughput of a network that is carrying them. Iperf offers the user to specify different parameters that can be used for testing a network, or for optimising and tuning purposes. In this experiment, Iperf is performed to measure only the maximum TCP bandwidth utilisation. Each Iperf experiment is run 15 times and the average number taken as the result. In addition, the link parameters are customised to 10, 20, 50, 100, 200, 500 and 1000 Mbps respectively. The Iperf tool is commonly utilised to optimise the network bandwidth, and the formula used for the calculation is as follows [7]:

$$B = ((V + S + I + U)/V) \times R \tag{5.3}$$

B = Actual link bandwidth in bits/second

V = Value given to Iperf's "-l" parameter

S = Size of Ethernet framing

I = IPv4 header size

U = UDP header size

R = Reported value in bits/second by Iperf.

Assumption: Run an Iperf server on *host 1* and run an Iperf client on *host 2* in order to parse the bandwidth achieved. Much lower TCP bandwidth is reported compared to Mininet's default topology because of the defined link parameters. The Iperf-reported TCP bandwidth with DAIM implementation running should be similar to OpenVswitch switch type as well as the OpenFlow kernel module running with NOX and POX, and possibly faster.

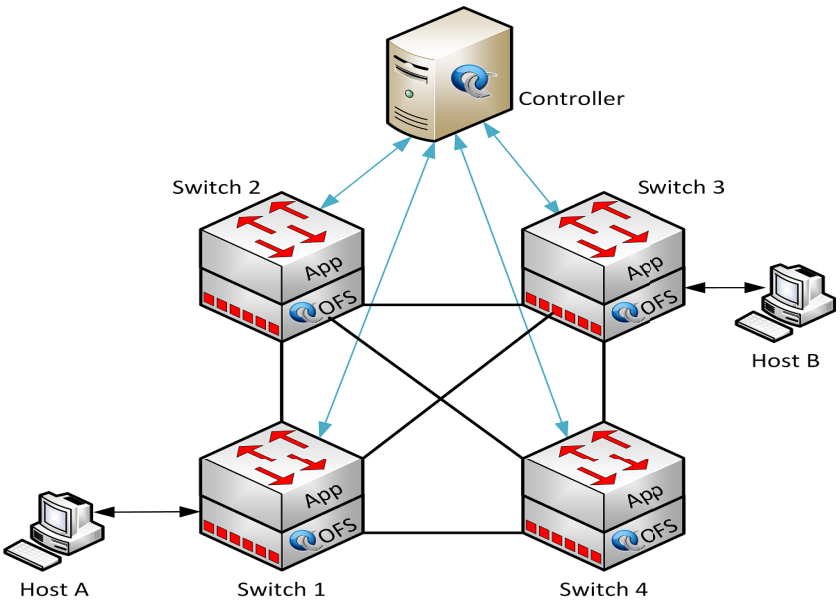


Figure 5.1.: Scenario Used to Evaluate Mean RTT and Maximum TCP Bandwidth

5.3.2. Scenarios

This section presents the experimental topology and scenarios used for the performance evaluation. Comparing the performance of DAIM with current OpenFlow controllers is considered as the primary goal. In terms of analysing the performance results correctly, the parameters and configurations have been defined with respect to links, bandwidth, delay and network size for the testing topologies.

5.3.2.1. Testing RTT and TCP Bandwidth

For both mean RTT and TCP bandwidth evaluation, a fully mesh network topology is used as shown in Figure 5.1. Both scenarios are created based on OpenVswitch in a virtualised network using Mininet. The configuration of the topology is built and used a fixed traffic profile with four switches, two hosts and one controller for simplicity, and to ensure that the controller calculates flow entries and installs them along the path from source to destination. This setup only applies to the evaluation of current centralised OpenFlow controllers. On the other hand, DAIM controllers are distributed to manage their own connected switches (refer to Figure 4.16 on page 122).

To analyse mean RTT values, *host A* performs a ping command to *host B* based on a fixed link bandwidth of 100 Mbps and 5 ms delay for each link. Each ping test is conducted with empty flow table to trigger the controller to insert new flow entries in each OpenFlow switch. The average RTT is measured in milliseconds and the

graph is plotted for different sizes of the ping packet. Similarly, Iperf is performed to test the TCP bandwidth between *host A* and *host B* with a fixed link delay of 5 ms for each link. The bandwidth available measurements are plotted against the actual link capacity of each test.

It is worth noting that flow table entries have an *idle_timeout* and a *hard_timeout* associated with them. For *idle_timeout* while matching packets, the flow entry will not be removed unless *hard_timeout* is reached, and then the flow entry will be forced to be removed. In Mininet, the *idle_timeout* are set to 5 seconds by default, whereas the *hard_timeout* is not set. To optimise the memory usage in the switches, an *idle_timeout* (10 seconds) is configured in the experiments for removing the flow entry in its flow table, forcing the path to rebuild when necessary. This strategy provides the most efficient use of existing flow table memory.

5.3.2.2. Testing Throughput and Latency

The scenario used to evaluate the flow setup throughput and latency is composed of an OpenFlow controller and merely one OpenFlow switch connected directly to the controller. The switch can vary the number of connected hosts up to a million and consequently these hosts are connected directly to the controller. The switch and hosts have been emulated with the Cbench tool in a virtual network environment (VirtualBox). The detail of the Cbench tool used to emulate the switch is depicted in Figure 5.2.

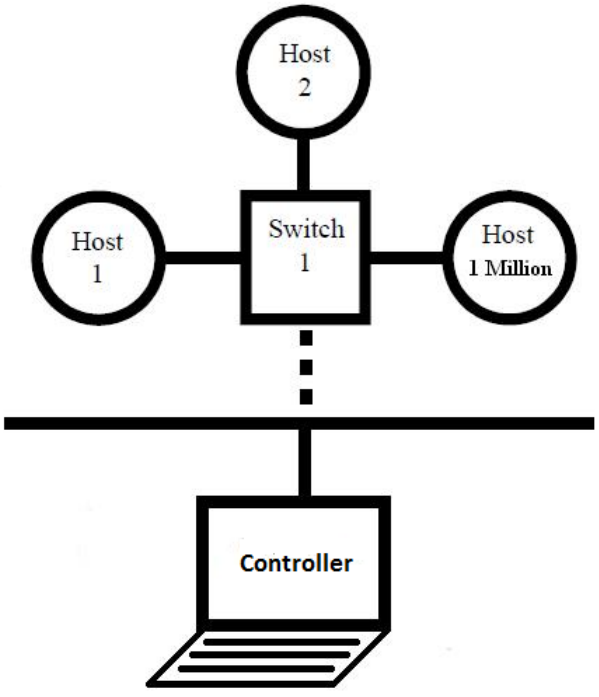


Figure 5.2.: Scenario Used to Evaluate the Flow Setup Throughput and Latency

The test is performed on the same device, the learning switch application and the Cbench evaluation tool to evaluate the throughput and latency. There have been several types of simulations performed. For example, increasing the number of MAC hosts available (”N” MACs ’1000 10000 100000 1000000 10000000’) and the number of test loops, hence increasing the number of hosts connected to the controller. Because the DAIM controller is responsible for only one dedicated switch, increasing the number of connected hosts is required in order to test the maximum number of Packet_In messages and evaluate the controller performance. Note that it is not feasible to vary the number of switches (“N” switches) because each of them has to be connected to at least one DAIM controller. As a result, Cbench tool can be used to test DAIM performance with only one connected switch. For a fair comparison, this test strategy is also used for the evaluation of NOX, POX and NOX-MT.

5.4. Results of Performance Evaluation

Mean RTT, throughput, latency and bandwidth are the main four parameters that are more important or determinants for the performance of any application. This section discusses the performance evaluation of DAIM communication channel (subsection 4.2.1 on page 97) as well as the L2 learning switch application. This application choice is based on the fact that it is relatively simple and basic switching has been implemented in all the controllers tested. Moreover, it uses all of the internal mechanisms and shows the effectiveness of the chosen programming language by developing single hash lookup. The learning switch application has been tested with the following controllers: NOX, POX and NOX-MT. The performance results of DAIM are compared to the existing OpenFlow controllers.

5.4.1. DAIM Communication Channel Results

Test 1: Determine the Mean RTT by Using the Ping Command

Description: This test shows a measurement of the length of time it takes for a signal to be sent plus the length of time it takes for an acknowledgement of that signal to be received. This time delay, therefore, consists of the propagation times between the two points of a signal.

Controllers: NOX, POX and DAIM channel

Application: Learning switch, Spanning Tree

Benchmarking parameters: Packet size from 56, 4000, 8000, 16000, 32000, and 65000 bytes

Command:

```
1 h1$ ping 10.0.0.2 -c 370 -s $packetsize
```

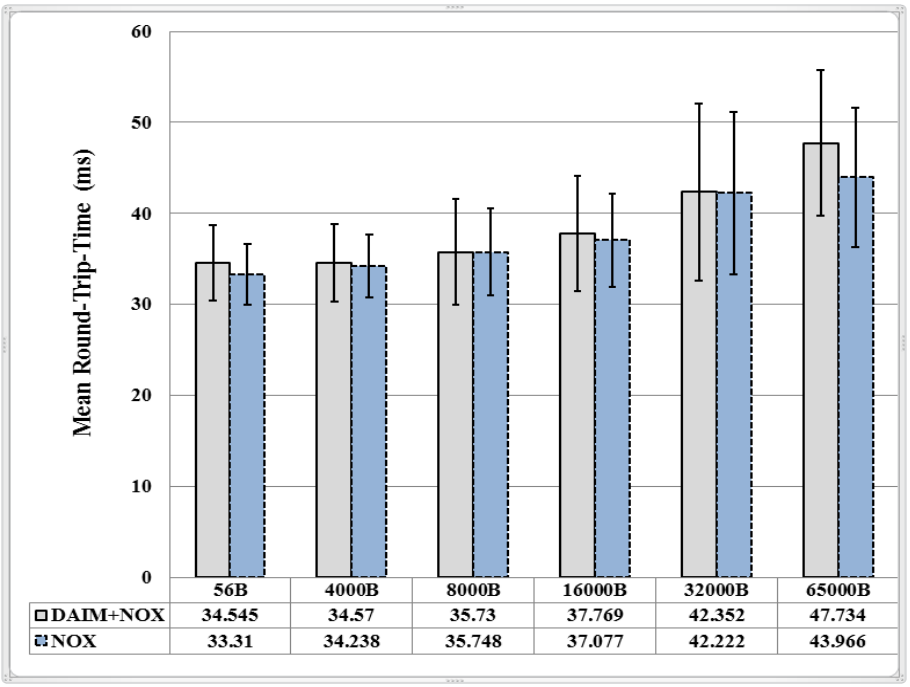


Figure 5.3.: Mean RTT DAIM Channel and NOX

Results (lower is better):

Figure 5.3 illustrates the mean RTT for the increasing packet sizes from default size of 56 up to 65,000 bytes. According to 1,850 measurements, the RTT values are converted into milliseconds in order to show how well the end hosts are connected. The table attached below shows the average RTT results in milliseconds. In the beginning, the performance of DAIM takes slightly longer time to process the ping than NOX. While the packet sizes are increasing, the RTT values with DAIM are performing very similar to NOX as shown in packet sizes 4,000 up to 32,000 bytes. However, it is obvious that DAIM produces longer RTT than NOX for 65,000 bytes with 47.73 and 43.96 ms respectively. Overall, the mean RTT of each packet size is nearly identical between both controllers with NOX performing slightly better.

Furthermore, the lines over the bar indicate the amount of variation from the mean standard deviation (σ). A low σ reflects on smaller packet sizes, whereas a high σ is shown as packet sizes increase due to RTT rate and are spread out over a large range of values. For example, the σ of DAIM is higher than NOX for most of the packet sizes. Overall, the σ for both DAIM and NOX are gradually increasing in accordance with the packet sizes.

Figure 5.4 shows a side by side comparison of the mean RTT performed by DAIM channel and the POX controller. From 1,850 measurements, the mean RTT for both DAIM and POX are escalating dramatically from default size of 56 up to 65,000 bytes. For each packet size, it is evident that the performance of DAIM takes significantly lower RTT to process the ping than POX. As packet sizes increases,

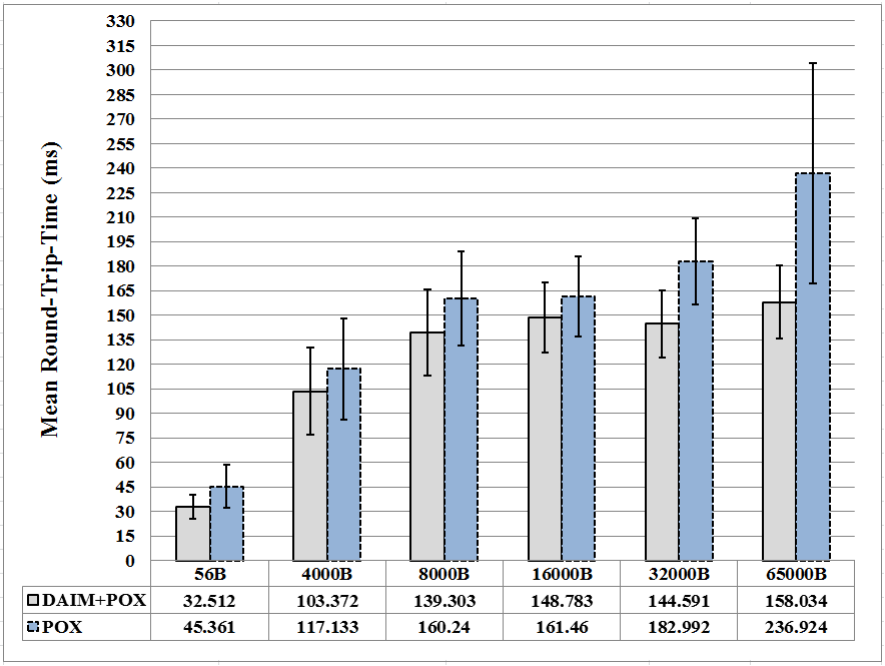


Figure 5.4.: Mean RTT DAIM Channel and POX

the mean RTT values with DAIM are performing quite steadily as shown in packet sizes from 8,000 up to 32,000 bytes with 139.30, 148.78 and 144.59 ms respectively. In contrast, the mean RTT values produced by POX are growing exponentially as the packet gets larger. Overall, the mean RTT performance comparison shows that the DAIM channel outperforms the original POX controller.

Test 2: Controller Throughput

Description: This test shows the measurements of the throughput of multiple controllers and compares them to DAIM. The default Cbench configurations of test loops and duration are used under this test. Also, the mappings of destination MAC addresses are learned before the test.

Controllers: NOX, POX and DAIM channel

Application: Learning switch

Command:

```
1 $ cbench -c localhost -p 6633 -s 1 -M 100000 -t
```

Results (higher is better):

Figure 5.5 shows the Cbench throughput results of the extended OpenFlow controllers with a single thread, in which an OpenFlow switch is contacting the controller in response to a new Packet_In (new packet arrival) event. The number of Flow_Mod responses per second for 16 tests are plotted in this figure. Each test is run with 1,000 ms duration and 100,000 unique source MAC addresses. DAIM

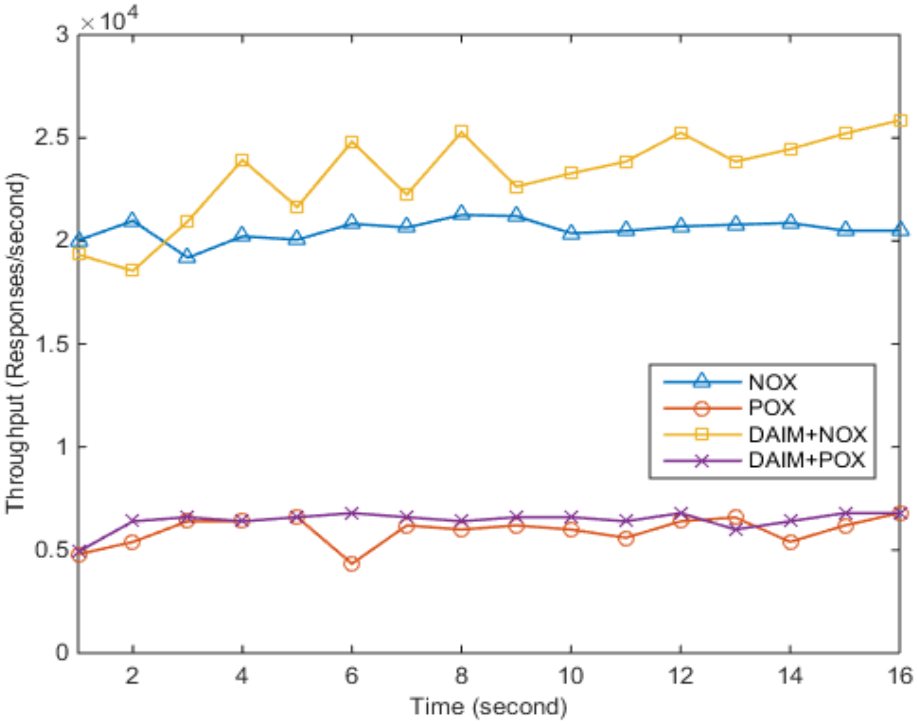


Figure 5.5.: Number of Flow Requests Handled per Second

channel running with NOX is able to handle the highest throughput on average 23,196 Flow_Mod responses per second, followed by NOX with average 20,540 responses per second. In comparison, the Python-based controller runs significantly slower. The throughput performance of DAIM channel running with POX can serve on average 6,438 responses per second. POX is the outlier in this test achieving only 5,952 responses per second on average. These results demonstrate that connecting the DAIM channel to any controller is more efficient and can perform higher throughput than the original controller.

Test 3: Controller Latency

Description: This test shows the measurements of average latency introduced by different controllers and compares them when running with the DAIM channel. These values show just part of the picture since there are differences in the implementation of the learning switch in different controllers.

Controllers: NOX, POX and DAIM channel

Application: Learning switch

Command:

```
1 $ cbench -c localhost -p 6633 -m 10000 -l 10 -s 1 -M 100000
```

Results (lower is better):

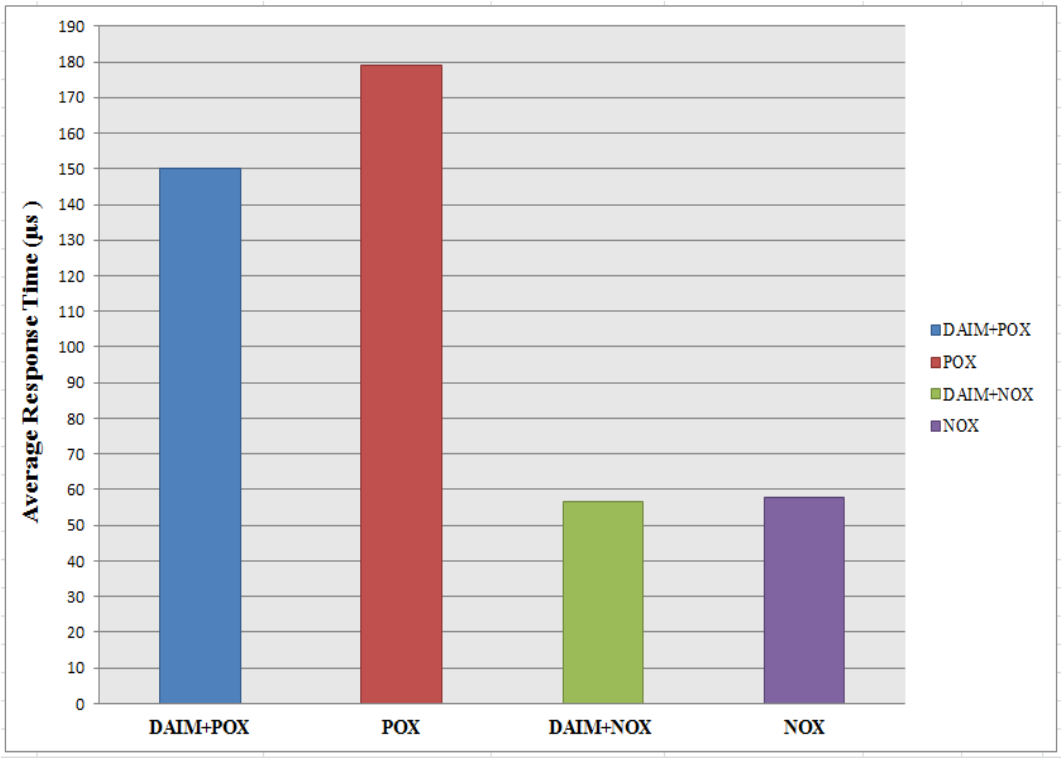
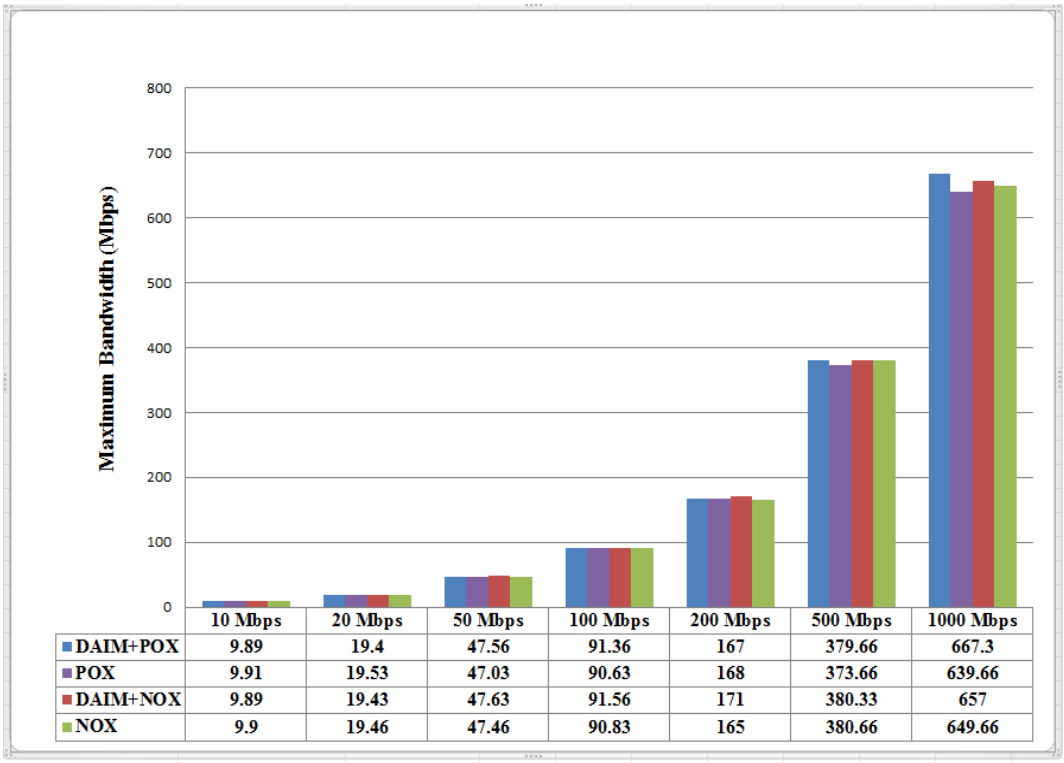


Figure 5.6.: Delay to Respond to Flow Requests

DAIM+POX: min/max/avg/stdev = 646.99/689.93/666.75/11.18 responses/s
POX: min/max/avg/stdev = 446.00/665.98/558.56/73.54 responses/s
DAIM+NOX: min/max/avg/stdev = 1512.88/1920.73/1760.78/105.16 responses/s
NOX: min/max/avg/stdev = 1528.00/1912.40/1731.09/109.24 responses/s

Controller	μs
DAIM+POX	149.981
POX	179.031
DAIM+NOX	56.793
NOX	57.767

The Cbench tool is used to perform latency test, in which an OpenFlow switch forwards a packet to the controller and waits for a reply, then repeats this process as quickly as possible. The total number of responses received can be used to compute the average time the controller took to process each event. The latency results are shown in Figure 5.6. DAIM channel running with NOX has the lowest average latency at 56.79 μs, followed by NOX with the average latency at 57.76 μs. Because Python-based controller runs much slower, this is clearly apparent in the latency difference between NOX and POX implementations. The measurement of average response time for DAIM channel running with POX is 149.98 μs. The POX controller produces the greatest latency in this test taking 179.03 μs on average.



	10 Mbps	20 Mbps	50 Mbps	100 Mbps	200 Mbps	500 Mbps	1000 Mbps
DAIM+POX	9.89	19.4	47.56	91.36	167	379.66	667.3
POX	9.91	19.53	47.03	90.63	168	373.66	639.66
DAIM+NOX	9.89	19.43	47.63	91.56	171	380.33	657
NOX	9.9	19.46	47.46	90.83	165	380.66	649.66

Figure 5.7.: TCP Bandwidth Utilisation Comparison

These results can confirm that the DAIM channel can provide the lowest average latency compared to the original NOX and POX controllers.

Test 4: Iperf Reported TCP Bandwidth

Description: This test shows the measurements of the maximum achievable TCP bandwidth on the network operated by different controllers and compares them to DAIM. The client and server functionality of Iperf are used to measure the throughput between the two ends.

Controllers: NOX, POX and DAIM channel

Application: Learning switch, Spanning Tree

Benchmarking parameters: Link capacity from 10, 20, 50, 100, 200, 500 and 1000 Mbps

Command:

```
h1$ iperf -s -p 5566 -i 1
h2$ iperf -c 10.0.0.1 -p 5566 -t 15
```

Results (higher is better):

The performance results with respect to TCP bandwidth utilisation are shown in Figure 5.7. Iperf is used to run a server on one host and a client on the second host

and parsed the bandwidth achieved. The maximum bandwidth utilisation is tested at different link variations from 10 Mbps to 1 Gbps. For 10 and 20 Mbps links, the Iperf-reported TCP bandwidth for NOX and POX controllers are slightly higher compared to that seen with the DAIM channel. However, as the link parameter increases from 50 Mbps up to 1 Gbps, the network bandwidth obtained with the DAIM channel is higher than its original controllers. For example, DAIM running with NOX can achieve a maximum bandwidth of 91.56 Mbps from 100 Mbps link capacity whereas NOX itself can only reach 90.83. For 500 Mbps link capacity, the reported bandwidth for DAIM running with POX can gain up to 379.66 Mbps compared to 373.66 by POX. Moreover, DAIM channel with POX shows surprisingly high bandwidth utilisation of 667.3 Mbps for 1 Gbps link, followed by DAIM running NOX with 657, NOX with 649.66 and POX with 639.66. These results show a clear indication of effective bandwidth utilisation provided by the DAIM channel.

It is also apparent that under large bandwidth more packets traverse the network. This can cause more overhead and computation time to process the flows. Thus, the maximum bandwidth utilisation is affected as the link parameter increases because increasing bandwidth does not influence computation speed.

5.4.2. Layer 2 Learning Switch Application Results

Test 5: Determine the Mean RTT by Using the Ping Command

Description: This test shows a measurement of the length of time it takes for a signal to be sent plus the length of time it takes for an acknowledgement of that signal to be received. Therefore, this time delay consists of the propagation times between the two points of a signal.

Controllers: NOX, POX, NOX-MT and DAIM

Application: Learning switch, Spanning Tree

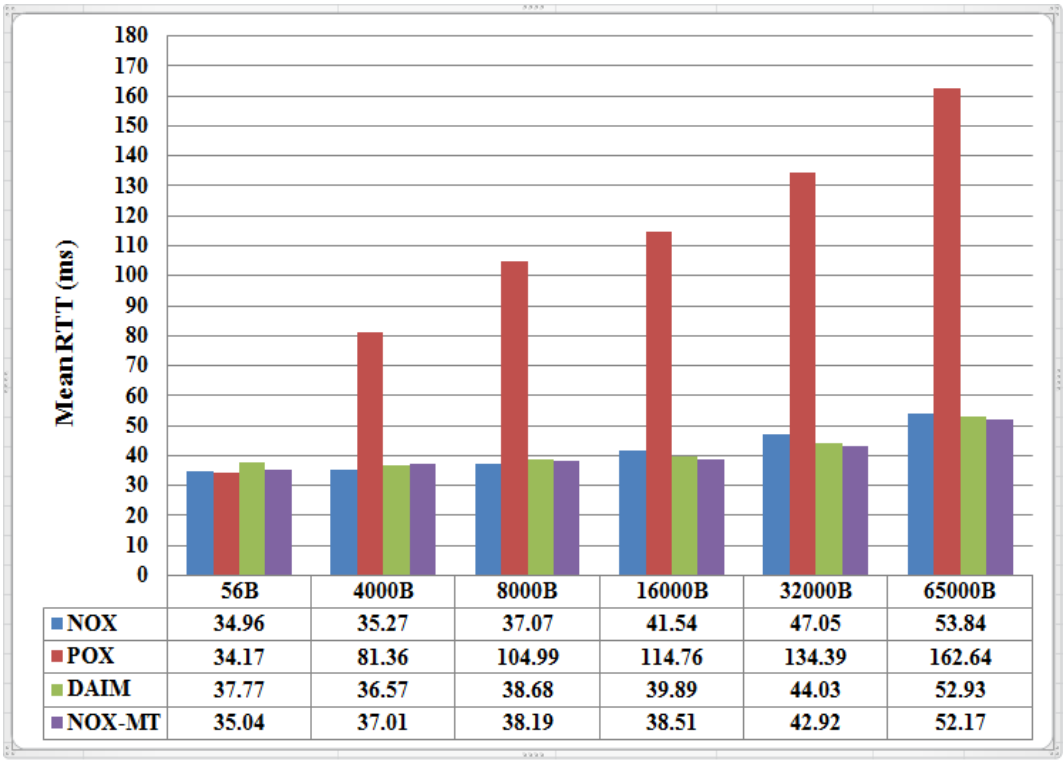
Benchmarking parameters: Packet size from 56, 4000, 8000, 16000, 32000, and 65000 bytes

Command:

```
1 h1$ ping 10.0.0.2 -c 370 -s $packetsize
```

Results (lower is better):

Each ping command generates 370 packets with the interval time of 1 second. The ping tests are performed with increasing size of the packet's payload 56B (8B header = 64B), 4000B, 8000B, 16000B, 32000B and 65000B. The measurements of the mean RTT values collected from each running controller are shown in Figure 5.8. Although POX produces the lowest mean RTT for 56B packet size with 34.17 ms, it takes significantly longer RTT to process a ping as the packet size increases. This can clearly be seen from packet sizes 4000 to 65000 bytes where the highest RTT from



	56B	4000B	8000B	16000B	32000B	65000B
NOX	34.96	35.27	37.07	41.54	47.05	53.84
POX	34.17	81.36	104.99	114.76	134.39	162.64
DAIM	37.77	36.57	38.68	39.89	44.03	52.93
NOX-MT	35.04	37.01	38.19	38.51	42.92	52.17

Figure 5.8.: Mean RTT Comparison

POX can reach 162.64 ms to complete a ping. On the other hand, NOX is able to perform the lowest RTT for 4000B and 8000B with 35.27 and 37.07 ms respectively, whereas the performance of DAIM and NOX-MT controllers take slightly longer RTT to execute the ping command than NOX ($<$ 1.75 ms). For packet sizes 16000B up to 65000B, NOX-MT shows the best mean RTT values at 38.51, 42.92 and 52.17 ms, followed by DAIM at 39.89, 44.03 and 52.93 ms respectively. Despite DAIM producing slightly longer RTT for smaller packets, it can outperform both NOX and POX controllers when pinging with larger packets. In comparison, the average RTT of each packet size between DAIM and NOX-MT is nearly identical with NOX-MT performing slightly better.

Test 6: Controller Throughput

Description: This test shows the measured average maximum throughput of multiple controllers and compares them to DAIM. For the switch control application, having a large number of unique source MAC addresses may result in a write-intensive workload.

Controllers: NOX, POX, NOX-MT and DAIM

Application: Learning switch

Benchmarking parameters: Number of unique MAC addresses (10^3, 10^4, 10^5, 10^6, 10^7)

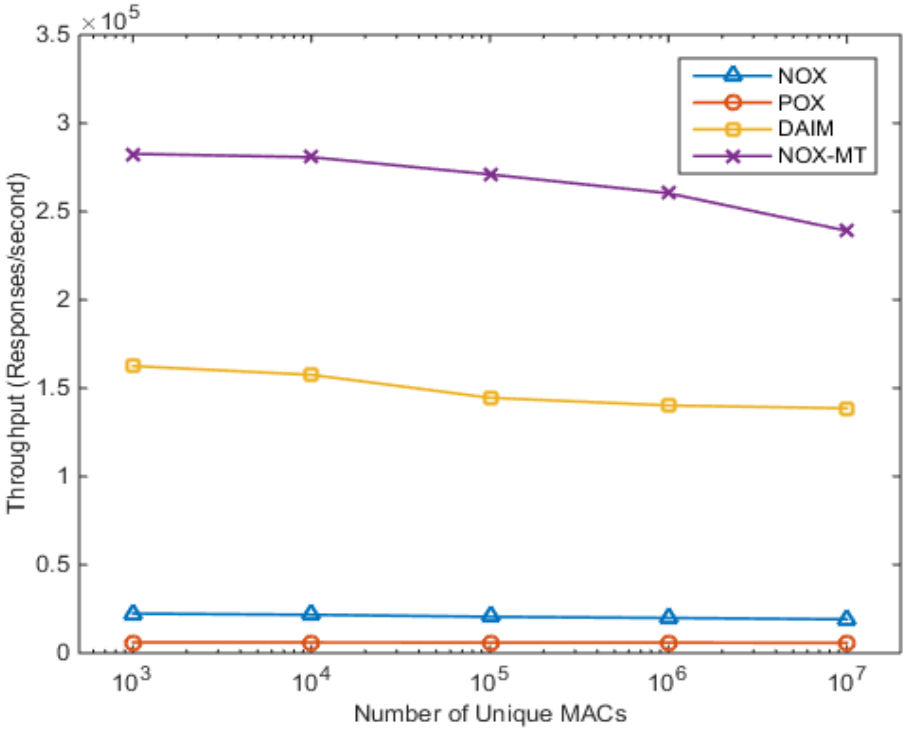


Figure 5.9.: Average Maximum Throughput Achieved with Different Number of MACs

Command:

```
$ cbench -c localhost -p 6633 -m 10000 -l 10 -s 1 -M $maddr -t
```

Results (higher is better):

Figure 5.9 demonstrates Cbench throughput mode results in correlation with the number of connected hosts. Under this test, DAIM, NOX and POX are used to process flows in a single threaded manner. In addition, these controllers and Cbench are each bound to a distinct physical core on the same processor. Because NOX-MT is a multi-threaded and highly optimised implementation of NOX, it shows the best throughput at maximum 292,612 responses per second. DAIM can produce the second highest throughput with 162,519 responses per second, followed by NOX with 22,357. The lowest throughput has been demonstrated by a typical Python-based POX controller, which runs considerably slower serving 6,096 responses per second. Although DAIM runs slower than NOX-MT, the performance of DAIM outperforms NOX by more than 7 times.

Moreover, the number of connected hosts in the network has a significantly impact on the performance of most controllers in this test. As shown in Figure 5.9, both NOX-MT and DAIM are clearly affected by this workload. NOX-MT's maximum throughput reduces from 292K to 239K responses per second with 10^7 hosts. In con-

Controller	Minimum response time (10^{-6} ms/flow)
NOX	57,531
POX	145,443
NOX-MT	11,392
DAIM	18,113

Table 5.1.: The Minimum Response Time

trast, the performance of DAIM decreases dramatically when more than 10^5 hosts are connected. The performance of NOX and POX slightly decreased to 19,200 and 5,800 flows per second respectively when the number of hosts increases to 10^7. All of these are caused by specific details when implementing the learning switch application, especially the implementation of its lookup table. In fact the performance of the controller is uniformly divided among the hosts.

Test 7: Controller Latency

Description: This test shows the average latency introduced by different controllers running a learning switch application and compares them to DAIM. These values show just part of the picture since there are differences in the implementation of the learning switch in different controllers.

Controllers: NOX, POX, NOX-MT and DAIM

Application: Learning switch

Command:

```
1 $ cbench -c localhost -p 6633 -m 10000 -l 10 -s 1 -M 100000
```

Results (lower is better):

The average response time of the controllers indicates insignificant correlation with the number of connected hosts. To evaluate latency of tested controllers, the delay of processing flow requests is calculated in regards to the average response time with one connected switch and 10^5 hosts. The latency tests are measured in milliseconds and the results are shown in Table 5.1 and Figure 5.10. The lowest average latency has been demonstrated by NOX-MT and DAIM controllers at 0.011 ms and 0.018 ms respectively. The average time it took the NOX controller to process each response is 0.057 ms, while the largest latency in this test is typical of Python-based POX controller taking 0.145 ms on average. If the network has a high latency, packets need to wait longer and hence increasing the possibility of packet loss and reducing the overall performance. As a result, POX is more suitable for fast prototyping than for enterprise deployment. According to *Test 6* and *Test 7*, the performance results show the DAIM controller can handle a high throughput while keeping the flow setup latency minimal.

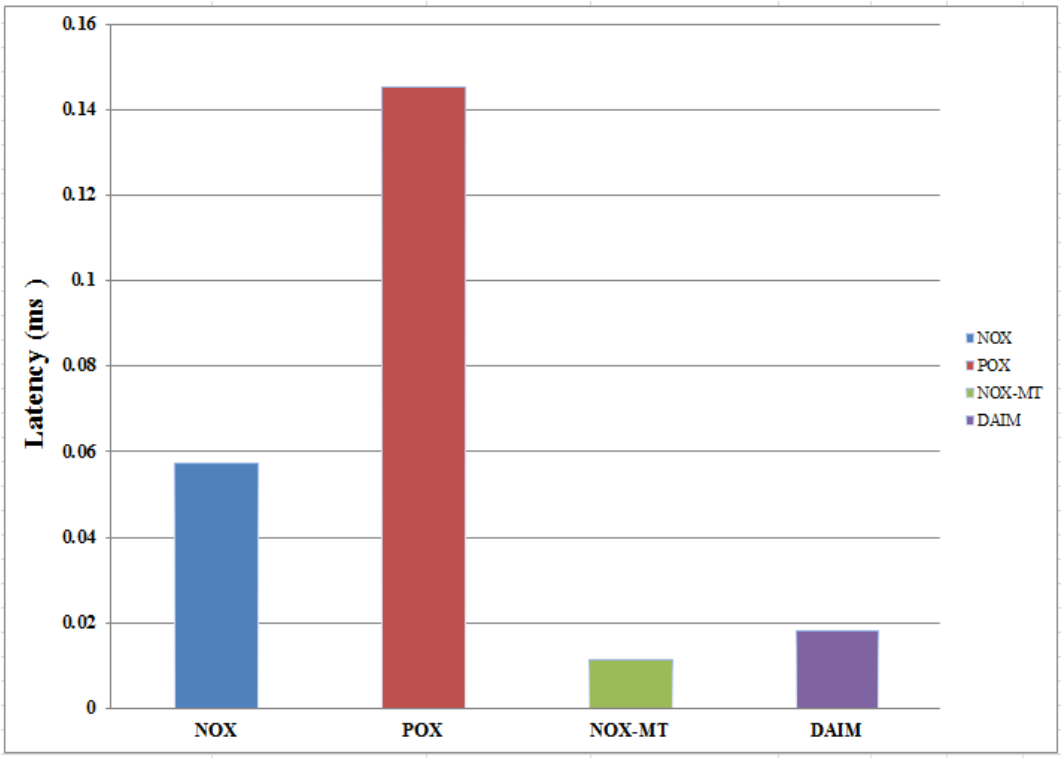


Figure 5.10.: Flow Setup Latency Comparison

Test 8: Iperf Reported TCP Bandwidth

Description: This test shows the measurements of the maximum achievable TCP bandwidth on the network operated by different controllers and compares them to DAIM. The client and server functionality of Iperf are used to measure the throughput between the two ends.

Controllers: NOX, POX, NOX-MT and DAIM

Application: Learning switch, Spanning Tree

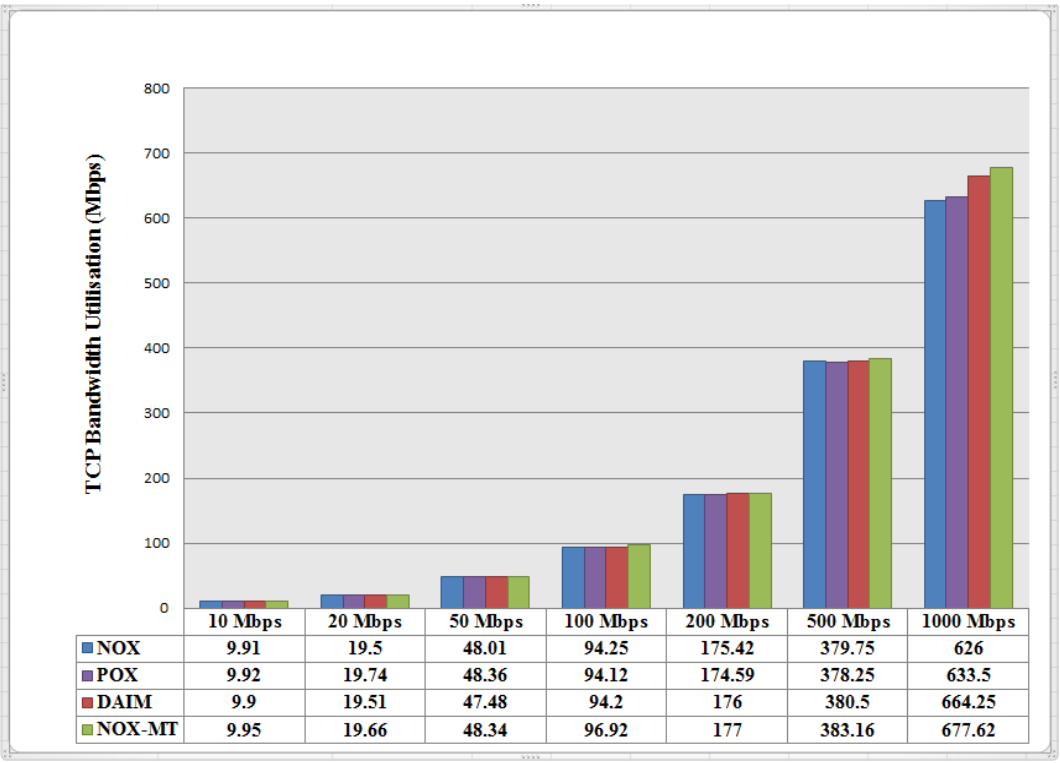
Benchmarking parameters: Link capacity from 10, 20, 50, 100, 200, 500 and 1000 Mbps

Command:

```
1 mininet> iperf
```

Results (higher is better):

In this test, Iperf is used to benchmark the controllers by determining the speed and the bandwidth available between two hosts. Once connected, these two hosts send packets to each other and report the results. The performance results in terms of TCP bandwidth utilisation are shown in Figure 5.11. The maximum bandwidth available is measured at different bandwidth parameters from 10 Mbps up to 1 Gbps.



	10 Mbps	20 Mbps	50 Mbps	100 Mbps	200 Mbps	500 Mbps	1000 Mbps
NOX	9.91	19.5	48.01	94.25	175.42	379.75	626
POX	9.92	19.74	48.36	94.12	174.59	378.25	633.5
DAIM	9.9	19.51	47.48	94.2	176	380.5	664.25
NOX-MT	9.95	19.66	48.34	96.92	177	383.16	677.62

Figure 5.11.: Network Bandwidth Comparison

For the links that have a bandwidth of 10, 20, 50 and 100 Mbps, the Iperf-reported TCP bandwidths from all the controllers are performing at a nearly identical rate. In particular, NOX-MT can achieve the highest bandwidth utilisation at 96.92 Mbps for the link that has a bandwidth of 100 Mbps. While the link bandwidth increases, NOX-MT also shows the best bandwidth utilisation in which it reaches a maximum 677.62 Mbps for the 1 Gbps link parameter. Even though DAIM produces the lowest TCP bandwidth at 47.48 Mbps from 50 Mbps link capacity, DAIM can achieve better performance from larger link bandwidth than NOX and POX controllers. For example, DAIM can perform a maximum bandwidth of 664.25 Mbps from 1 Gbps link, followed by POX at 633.5 and NOX at only 626 Mbps. For link bandwidth of 500 Mbps, the reported bandwidth available for DAIM can gain up to 380.5 Mbps compared to 379.75 by NOX and 378.25 by POX. All of these results can show evidence of an efficient bandwidth utilisation provided by the DAIM controller.

5.5. Build a Physical OpenFlow Test Lab Controlled by DAIM

This section shows how to create a simple SDN/OpenFlow lab to launch research tests with Raspberry Pi 2 (model B). The main goal is to demonstrate that DAIM

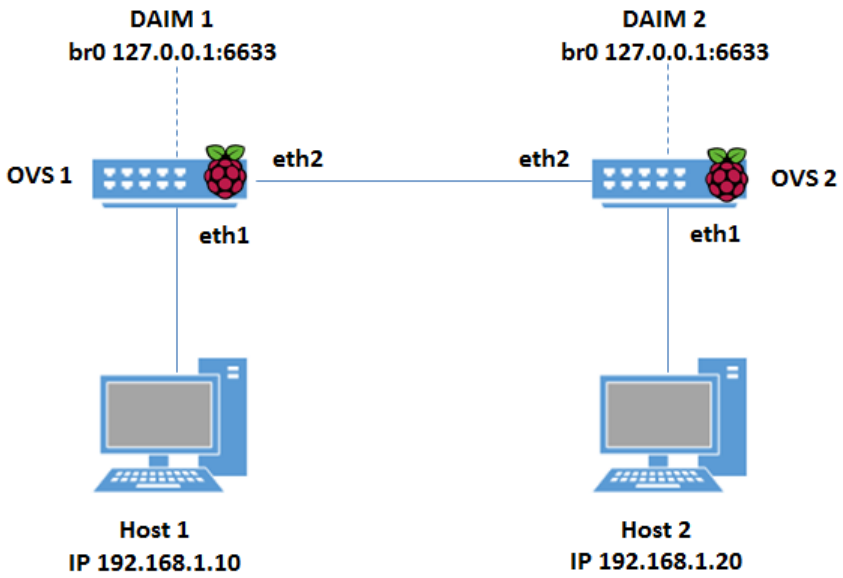


Figure 5.12.: Physical OpenFlow Test Lab Topology

can perform operational functions and can be deployed in real networks. A small-size of OpenFlow network has already been set up physically in the lab with two software switches controlled by DAIM and two hosts. An Ubuntu 14.10/Linaro 15.01 "developer" image is the operating system used for the Raspberry Pi's, which comes with the LXDE desktop pre-installed. In addition, each Raspberry Pi runs an OpenVswitch implementation and a DAIM controller on the same device, whereas the hosts are just normal PCs. The OpenFlow software reference system is also installed to add OpenFlow switching capability to the Raspberry Pi with multiple NICs. Figure 5.12 shows a basic linear topology for OpenFlow experiment. The setup installation and configuration as well as some useful tests are also demonstrated in this section.

5.5.1. Configuration Summary

Software OpenFlow switches:

Switch	IP address to talk to the controller	Controller Ethernet Port	OpenFlow Ethernet Ports
Switch 1 (Raspberry Pi)	loopback	br0 (virtual)	eth1 192.168.1.2 eth2 192.168.1.99
Switch 2 (Raspberry Pi)	loopback	br0 (virtual)	eth1 192.168.1.1 eth2 192.168.1.98

DAIM Controllers:

IP address	TCP port number
127.0.0.1	6633

5.5.2. Setup OpenFlow Switch and DAIM Controller on a Raspberry Pi

In this setup, the DAIM controller is used to manage the data plane and turn the Raspberry Pi into an intelligent learning switch.

Step1. Setup OpenVswitch

The simplest way to install OpenVswitch on Ubuntu is to run the following command as root user:

```
$ apt-get install openvswitch-common openvswitch-switch
```

For more details, refer to Appendix B on page 191.

Then, create an OpenVswitch bridge *br0* and add the physical interfaces to the virtual bridge:

```
$ ovs-vsctl add-br br0
$ ovs-vsctl add-port br0 eth1
$ ovs-vsctl add-port br0 eth2
```

Step2. Run the DAIM controller and attach OpenVswitch

After the data plane on each Raspberry Pi is installed, start the DAIM controller to effectively manage network flows:

```
$ ./aniv21 -p 6633
```

Then, connect the switch to the DAIM controller on a second terminal. Since OpenVswitch and the DAIM controller are on the same machine, the command will be:

```
$ sudo ovs-vsctl set-controller br0 tcp:127.0.0.1:6633
```

The DAIM controller uses OpenFlow default port 6633. The output in the DAIM console (controller terminal) will appear as follows:

```
Info: creating ANI socket...
Info: established ANI server!
Info: waiting for the switch connection...
Info: connected to the switch!
```

5.5.3. Basic Test

On a new terminal of each Raspberry Pi, first verify that the switches are connected to the DAIM controller by the following command:

```
$ sudo netstat -an | grep 6633
```

The output of 'netstat' should display the network connections for TCP port 6633 as shown below:

```
tcp    0    0 0.0.0.0:6633          0.0.0.0:*               LISTEN
tcp    0    0 127.0.0.1:6633        127.0.0.1:56533         ESTABLISHED
tcp    0    0 127.0.0.1:56533       127.0.0.1:6633          ESTABLISHED
```

Enter the command below to check the connection was made correctly:

```
$ sudo ovs-vsctl show
```

The output of 'ovs-vsctl show' which displays the bridge interfaces and the connection state should appear as follows:

```
31f45ffe-8526-48d1-8ad4-b61320838672
    Bridge "br0"
        Controller "tcp:127.0.0.1:6633"
            is_connected: true
        Port "eth1"
            Interface "eth1"
        Port "eth0"
            Interface "eth0"
        Port "br0"
            Interface "br0"
                type: internal
    ovs_version: "2.1.3"
```

After assigning the IP addresses to host 1 and host 2 (on the same subnet as shown in Figure 5.12), run ping to check they can communicate. On each switch, run the following command to see the flow entries are actually installed in the flow table:

```
$ sudo ovs-ofctl dump-flows br0
```

The output of 'dump-flows' after a ping command should look like the following:

```
NXST_FLOW reply (xid=0x4):
cookie=0x0,duration=2.604s,table=0,n_packets=1,n_bytes=98,
  idle_timeout=60,idle_age=2,priority=65535,icmp,in_port=2,vlan_tci=0
  x0000,dl_src=00:14:22:41:ce:b7,dl_dst=00:13:72:8b:cc:c1,nw_src
  =192.168.1.20,nw_dst=192.168.1.10,nw_tos=0,icmp_type=0,icmp_code=0
  actions=output:1
cookie=0x0,duration=2.581s,table=0,n_packets=1,n_bytes=98,
  idle_timeout=60,idle_age=2,priority=65535,icmp,in_port=1,vlan_tci=0
  x0000,dl_src=00:13:72:8b:cc:c1,dl_dst=00:14:22:41:ce:b7,nw_src
  =192.168.1.10,nw_dst=192.168.1.20,nw_tos=0,icmp_type=0,icmp_code=0
  actions=output:2
```

More sophisticated test procedures according to section 4.5 on page 123 have also been tested on hardware. As a result, this OpenFlow test lab is ready to host different novel research scenarios.

5.5.4. Preliminary Hardware Performance Results

This section briefly discusses the performance analysis of the DAIM controller running on a Raspberry Pi. The ping command is used to check network connectivity and measure the mean RTT between two IP hosts. The flow setup throughput and latency of the DAIM controller are measured by using Cbench. Iperf is the tool used to measure the available bandwidth of the network.

Test 9: Determine the Mean RTT by Using the Ping Command

The ping tests are performed with increasing size of the packet's payload from default size of 56 up to 65,000 bytes. Each ping command generates 100 packets and waits for one second between sending each packet. The mean RTT measurements collected from the ping tests are displayed in milliseconds below.

Packet size (bytes)	Mean RTT (ms)
56	10.55
4,000	30.41
8,000	46.59
16,000	80.94
32,000	148.06
65,000	289.76

The results show a clear indication that the performance with respect to mean RTT is affected by the size of the packet. In addition, the mean RTT values are growing significantly as the size of the packet increases. As a result, pinging or transferring a file with larger packet size may lead to packet loss and poor performance. However, the response time performance can improve in various ways. For example, using wide pipes and high-quality cables (fiber) will help ensure low RTT in the case of high traffic. Moreover, using other high-end networking devices can also reduce the response time as the Raspberry Pi does not have enough available resources to manage the packets quickly.

Test 10: DAIM Throughput

Cbench is used to emulate one OpenFlow switch connected to the DAIM controller and measure the number of flow setups per second that DAIM can handle. Each Cbench run consists of 16 test loops of default 1000 ms duration and a fixed 100000 unique source MAC addresses. The total responses received from each test are averaged to compute the responses per second result. The raw throughput results are shown below.

DAIM: min/max/avg/stdev = 24952.54/28823.65/27139.88/1070.14 responses/s

In this test, the DAIM controller running on a Raspberry Pi can only achieve an average throughput of 27,139 responses per second. There are a few factors that can affect this throughput performance. Firstly, insufficient CPU cycles and main memory consumptions of the Raspberry Pi could significantly reflect on the results of this experiment. Although Raspberry Pi provides a low-cost way to reboot computing in test labs, it imposes resource limits. Secondly, DAIM is designed and implemented as a single-threaded instance and hence not optimised for performance yet. Therefore, deploying more computing power and using optimisation techniques such as I/O batching and multi-threaded processing can effectively improve DAIM's throughput performance.

Test 11: DAIM Latency

The fundamental design requirements of any controller are to fairly support requests from different switches, achieve low flow setup latency and scale effectively on large networks. In this test, the latency is measured as the average response time with one switch, which sends requests to the DAIM controller, and each time waits for a reply before sending the next flow request. The raw results from Cbench latency mode can be seen below.

DAIM: min/max/avg/stdev = 346.97/391.33/362.17/15.47 responses/s

The average response time of DAIM with one simulated switch and 10^5 hosts is 0.276 ms. This delay to respond to flow requests is caused by the unavoidable overhead of the socket read/write system calls. Moreover, overhead from warming up the CPU caches and executing applications to process new flow requests can also cause a small additional setup time. Therefore, reducing such unavoidable overhead across a larger number of flow requests is required to improve the DAIM controller performance. This can be achieved by decreasing the number of system calls by reading/writing more bytes in every socket call, using a larger read buffer (batching) and reducing the number of application executions. As a result, DAIM can be optimised to restrain latency while keeping high throughput.

Test 12: Iperf Reported TCP Bandwidth

In this test, Host 2 (IP 192.168.1.20) is used as the Iperf server and Host 1 (IP 192.168.10) as the Iperf client. The available bandwidth is measured through TCP tests. By default, the server connects to the client on the TCP port 5001. However, using this port the kernel OpenVswitch daemon of the Raspberry Pi crashed and was unable to complete the Iperf experiments. Thus, a new server communication port is assigned to 6689 to facilitate the connection. The bandwidth displayed below by Iperf is the measured bandwidth from host 1 to host 2.

Client side:

```
1 ------------------------------------------------------------
2 Client connecting to 192.168.1.20, TCP port 6689
```

```
TCP window size: 16.0 KByte (default)
------------------------------------------------------------
[  3] local 192.168.1.10 port 44047 connected with 192.168.1.20 port
   6689
[ ID] Interval       Transfer     Bandwidth
[  3] 0.0-15.1 sec  9.25 MBytes  4.81 Mbits/sec
```

The test is performed with 15 seconds duration and 1 second interval between periodic bandwidth reports. As can be seen, the test was run from client to server displaying the transfer rate (MB) and bandwidth (Mb) performance taking the average number as the result. The test output shows the DAIM controller is able to transfer 9.25 MBytes of data at a rate of 4.81 Mbits/sec from host 1 to host 2 using the default window size. It is important to note that Iperf consumes all bandwidth available between client/server via TCP, regardless of LAN, WAN, or VPN connection. Moreover, increasing the TCP window size (*e.g.*, -w2000) will give better performance. For a small-size OpenFlow network running on limited resources, the DAIM controller can provide fair performance in terms of bandwidth utilisation.

5.6. Other Parameters That Can Affect the Performance Evaluation

Apart from the four tested parameters, the causes or reasons why performance can be limited or affected are analysed in this section. There are various aspects that may influence the performance of any application. These aspects can be listed as follows:

The code: According to the controllers tested, the code that has the greatest extension is the NOX controller, which is based on Python/C++ programming language. Table 5.2 shows the code extension for the learning switch application of different OpenFlow controllers. The number of lines of code to implement the same application for NOX, POX and NOX-MT is 222 lines, 186 lines and 204 lines respectively. In comparison, the DAIM code is more efficient and compact because it takes only 109 lines to implement the same learning switch application.

Tested controllers	Programming language	Extension, number of lines
NOX	Python/C++	222
POX	Python	186
NOX-MT	C++	204
DAIM	C++	109

Table 5.2.: SDN/OpenFlow Controllers: Code Extension

The programming language: A main distinction between the tested controllers is the language they are written in. Comparing the implementation language amongst them can also be enlightening. The programming language used to develop the controller is a major aspect that can affect the performance of any application. For example, the key limitation of Python is the speed, which can influence the performance evaluation of NOX and POX controllers. Despite Python's ability to produce shorter code for the same application, in many cases it is generally slower than equivalent code in C++. However, Python applications can offer faster development time and the affordability of memory. In contrast, NOX-MT and DAIM are developed using a natively compiled C++ language and hence resulting in better overall performance in terms of throughput.

Threads: The thread of execution can be a significant factor that affects the performance evaluation. Between the tested controllers, NOX, POX and DAIM are not optimised for performance and designed to process events in a single-threaded instance. They also do not support multi-threading and therefore show no scalability across CPU cores. In contrast, NOX-MT is a multi-threaded successor of NOX that uses well-known optimisation techniques such as I/O batching and a fast multiprocessor-aware malloc implementation to improve the baseline performance. As a result, NOX-MT can establish a new lower bound on the maximum throughput.

The operating system on which various controllers are tested: More accurate data and results can be obtained depending on the device where the performance tests are performed. For example, making use of the taskset tool provided by Ubuntu may produce more accurate results. If an OpenFlow controller is evaluated on a device that has a processor with 2 cores and each core has multiple threads (*e.g.*, between 2 and 8), the results of this evaluation may be more accurate and reliable than a controller evaluated on a device with only a single core and a single thread.

Part III.

Drawing Conclusions

6. Conclusion and Future Work

This chapter is a summary of the interpreted research outcomes that have been achieved by this thesis. The conclusions are validated and stated with respect to the propositions initially presented in the introduction chapter. Furthermore, some of the limitations and weaknesses of this research as well as directions for the future study are discussed.

The major goal of the research work elucidated in this thesis is to design, model and develop a distributed active information model (DAIM) as a new approach to the architecture and the control platform of a SDN network. Due to the fact that large multi-service networks are becoming more complex and difficult to manage, the conventional management paradigm is struggling to cope with network bottlenecks of switch and routing based network deployments. SDN is an emerging management framework that promises to overcome the bottlenecks of traditional data networks by separating the network's control logic from the underlying switches and routers, promoting a logically centralised network control and introducing the ability of programmable networks. However, an OpenFlow-based SDN network, managed from a central point, naturally leads to a single point of failure and lack of scalability. Many research proposals have been made to avoid having this failure for the entire network and allowing a scalable architecture, which are mostly based on a physically distributed but logically centralised control plane. Although they offer a simplified central view of the network and reduce the look-up overhead by allowing communication with local controllers, these solutions are not adapted to large-scale networks with several autonomous systems and require extensive traffic between controllers to maintain a global network view. Hence, a well-designed system with distribution strategies will make the OpenFlow network more efficient, robust and reliable.

In particularly, a *candidate* DAIM system has been proposed to allow programmability of network elements and local decision-making processes that will essentially contribute to complex distributed networks. The notion of programmable networks has currently gained considerable attention due to the rise of the SDN paradigm, which promises to greatly simplify network management through network programmability. The managed network elements are enabled to cope with the ever-changing environments and are able to satisfy the requirements for service providers, vendors and end-users. The robustness of the distributed network architecture as well as the distributed management control has been presented. The design and the implementation of the overall system model validate the research propositions in Chapter 1, and contributions are further validated by the analysis of network performance

metrics from experiments in this thesis.

This dissertation demonstrates that the DAIM model can be applied to the OpenFlow-based SDN architecture and also perform operational functions. In addition, the DAIM model is seamlessly integrated into the SDN structure at the level of switches to process the flows from local information distributed across network elements. DAIM is developed to avoid the problem of a single point of failure and address the scalability issue of OpenFlow. The proposal moves the computational load to the switches, and effectively the DAIM model creates a logically distributed control plane. This distributed control plane paradigm offered by DAIM allows the switches to manage flows locally instead of by a centralised OpenFlow controller, and enables switches to adapt dynamically to the ever-changing circumstances. The performance of the DAIM model has been shown to possess efficiency, robustness, flexibility and can improve the flow setup throughput and latency of an OpenFlow controller. Most importantly, building a distributed computing environment offered by DAIM can create the appropriate "hooks" that will enable the development of autonomic management strategies.

6.1. Research Propositions Validation

The primary objectives of the thesis to develop a new distributed active information model theory (Chapter 3), implement and validate a candidate DAIM system that could support distribution services in an OpenFlow network (Chapter 4) have been met. The performance of the DAIM model with respect to mean RTT, throughput, latency, and bandwidth utilisation have been thoroughly evaluated and assessed through a number of experiments and measurements (Chapter 5). The results obtained from simulation tests and hardware implementation indicate that the DAIM model with distributed computing capabilities is feasible for use in a suitable OpenFlow environment.

The research propositions presented in the introduction chapter are re-examined and verified here in regards to the research in Chapters 3, 4 and 5. These research propositions are validated and summarised in the same order as they are initially outlined and their contributions towards SDN-based applications and distributed communication systems are stated.

Proposition 1: *The DAIM model will be able to integrate seamlessly into the OpenFlow architecture at the level of the switches to provide a logically distributed control plane.*

Validation Highlights: An efficient DAIM model based on L2 forwarding is constructed and applied to the SDN architecture at the level of OpenFlow switches to create a logically distributed control plane. The DAIM system is composed of three core modules: a communication module, a local storage module and a controller module. By applying these modules, managing an OpenFlow network with physically distributed controllers can become adaptable, scalable and robust. The DAIM model can enhance the OpenFlow switches to make their own local decisions through its active performance, and thus significantly reduce the workload of a centralised SDN/OpenFlow controller.

The first deployments of OpenFlow typically use a single centralised controller. This may cause the problems of having a single point of failure and performance issue of the ability to support a large number of flow requests as the network size grows. These limitations are overcome by use of the DAIM model, which is physically distributed, easy-to-maintain locally and dynamic. In order to successfully build the DAIM model, a well-designed architecture is needed to organise massive information of network elements under large-scale and unpredictable distributed network environments. This is described in Chapter 3, whereby the theoretical background based on the O:MIB structure and O:XML technique are firstly presented to enhance the information collection process. The fundamental principles of the DAIM model are described and its practical feasibility to the SDN paradigm is also discussed in terms of design, architecture and implementation. Furthermore, the reason why the DAIM model is important and required is further explained, and hence the contribution of DAIM is justified.

The contribution of the implemented DAIM model is discussed in the first part of Chapter 4. The main contribution to knowledge of this chapter is developing the DAIM model within OpenFlow switches as a logically distributed control plane paradigm. The DAIM model is invented with the goal to address the limitations of current OpenFlow approaches and future distributed network systems aiming at an autonomic management strategy. DAIM model implementation has been described in three phases including a basic carrier, semi-distributed and fully distributed functionality. The details of the DAIM system are further discussed from a software specification perspective. More specifically, three core modules that construct the DAIM ecosystem are presented with some sample code reviews and flowchart diagrams of the implemented algorithms. These implemented modules are comprised of the communication module, the local storage module and the controller module. The most significant OpenFlow messages required for the communication between the switch and the DAIM controller are also explained.

We have successfully applied an effective DAIM model into the SDN structure to provide a logically distributed control platform for OpenFlow networks. As a consequence, the Proposition 1 is validated and this objective has been achieved. The DAIM structure is further used for the experimental validation and performance analysis in this dissertation.

Proposition 2: *This logically distributed control plane will allow the switches to manage flows locally by having DAIM operate on packets instead of a centralised controller in an SDN environment.*

Validation Highlights: Using Mininet, the deployment of the DAIM model in OpenFlow networks can perform operational functions locally without support and vetting from a centralised controller. The system validation results in satisfactory performance in terms of efficiency and robustness. A small-size OpenFlow lab based on Raspberry Pi's has been set up physically to demonstrate that DAIM can be deployed in real world networks.

The feasibility validation is mainly from a set of tasks performed to check the compliance of the DAIM system with its purpose and functions. Examples of the tasks are using the ping command between hosts, creating Linux command line chat server, streaming video via VLC media player and running a simple web server-client to verify network connectivity. These tasks and scenarios are demonstrated in the second part of Chapter 4. The scenarios used to verify the functionalities of the DAIM model are created in a virtualised network using Mininet emulator. Using this tool to test the system's functionality is considered as the primary goal. Different types of network topologies are also used in such scenarios, which can be categorised into the following basic types: linear, ring, tree, and fully mesh topology. Moreover, a flow table buildup from ping traffic is explained to show that DAIM operates exactly as it is designed to do in a consistent and functional manner, and to demonstrate the underlying protocol in action through various network services and applications. The objective of DAIM validation is to produce documented evidence, which provides a high degree of assurance that all components of the system will consistently work correctly when brought into use.

The feasibility of applying DAIM to real networks has been further illustrated in the physical OpenFlow test lab. The hardware implementation of DAIM based on a set of Raspberry Pi's is described in Chapter 5. The main goal is to demonstrate that DAIM can perform operational functions and can be deployed in real networks. A small-size OpenFlow network has already been set up physically in the lab with two software switches controlled by DAIM and two hosts. An Ubuntu 14.10 image is the operating system used for the Raspberry Pi's, which comes with the lightweight LXDE desktop pre-installed. Each Raspberry Pi runs an OpenVswitch implementation and a DAIM controller on the same device, whereas the hosts are normal PCs. The OpenFlow software reference system is also installed to add OpenFlow switching capability to the Raspberry Pi with multiple NICs. The setup installation and configuration are documented and some useful tests are also conducted to verify the functionality of the system.

Taking account of the outcomes of Mininet and Raspberry Pi, we can conclude that by applying DAIM, an OpenFlow network can perform operational functions in terms of forwarding packets and installing appropriate flows without the vetting from a centralised controller. Therefore, after revisiting Proposition 2, the author is

of the opinion that the DAIM model can also be applied to a more sophisticated architecture or other distributed computing environments such as WSNs, peer-to-peer applications, and cellular networks.

Proposition 3: *The performance of the DAIM model can be proved to possess efficiency, robustness/resilience, flexibility, and will improve the flow setup throughput and latency of an OpenFlow controller.*

Validation Highlights: The results from experimental tests show that the DAIM controller has fair performance results in regard to mean RTT, flow setup throughput, latency and TCP bandwidth utilisation. The performance comparisons between DAIM and other controllers used primarily for research purposes show DAIM is a promising and innovative distributed control plane in managing flows with a distributed nature. The experiments further prove the DAIM controller outperforms both NOX and POX controllers.

In this thesis, the performance of an OpenFlow controller is defined by four characteristics: mean RTT, throughput, latency and bandwidth. The aim is to obtain the lowest mean RTT, maximum throughput (number of outstanding packets, flows per second), minimum latency (response time, ms), and maximum TCP bandwidth (the rate of data transfer) for each controller. To quantify the controller performance, the ping command with different packet sizes is executed to measure the mean RTT values. The Cbench tool is used to measure the number of flow setups per second that the controller can handle in terms of throughput and latency. Whereas, Iperf is served as a benchmarking tool for testing the network's maximum TCP bandwidth utilisation. The performance results of DAIM are compared with other OpenFlow controllers including NOX, POX and NOX-MT. The outcomes of experimental evaluations are discussed in Chapter 5. For all experiments, each controller implements a normal L2 learning switch application provided by the controller. NOX, POX and DAIM are designed as a single-threaded instance, whereas NOX-MT is a multi-threaded and highly optimised implementation of NOX.

Regarding mean RTT, the DAIM controller produced slightly longer RTT for smaller packets (< 1.75 ms difference) but can outperform both NOX and POX controllers when pinging with larger packets (*e.g.*, 16000, 32000 and 65000 bytes). In comparison, the average RTT of each packet size between DAIM and NOX-MT is nearly identical with NOX-MT performing slightly better. Moreover, the experimental results from Cbench indicate that the throughput performance of DAIM outperforms POX and NOX by a factor of 26 and 7 respectively on a server with two duo-core 3GHz processors. NOX-MT uses optimisation techniques such as I/O batching and multi-threaded processing and thus shows the best average throughput. The performance of the DAIM controller is further evaluated in terms of flow setup latency. The comparison results show that DAIM has a better latency performance than NOX and POX in the scenario with one connected switch and 10^5 hosts. Furthermore, Iperf is used to run a server on one host and a client on the second host, and check the speed between them. The Iperf results show a clear indication of effective

bandwidth utilisation provided by the DAIM controller.

The DAIM controller demonstrates a similar RTT and bandwidth performance with NOX-MT, and DAIM has the better performance than NOX and POX in all network metrics. The test results show the DAIM controller can handle a high throughput while keeping the flow setup latency minimal. Thus, DAIM is proved to possess efficiency, resilience and improves the throughput and latency of an SDN controller.

Proposition 4: *Most importantly, in building the distributed computing environment offered by DAIM, this can create the appropriate "hooks" that will allow the development of autonomic management strategies.*

Validation Highlights: After successfully implemented a distributed computing environment offered by DAIM, this research outcome could be the first attempt in enabling the autonomic service management for SDN. Additionally, the invented DAIM model can bring forward a new approach to autonomic communication networks.

It is reasonable to deploy the decentralised management paradigm instead of centralised management paradigm in order to meet the requirements of future complex distributed networks. Therefore, an innovative DAIM architecture is proposed to cope with the scalability and central point of failure issues of SDN. This architecture guides the system implementations, feasibility validations as well as experiments through the thesis. The performance of the DAIM framework in terms of efficiency and robustness is therefore tested and evaluated. The outcome of the implemented DAIM model has a strong impact on future distributed systems and applications in OpenFlow networks. DAIM is successfully developed to create a logically distributed control plane and hence gives effect to autonomic behaviours.

Because decentralisation is a sine qua non, an essential condition, of autonomic systems, building a distributed computing environment by DAIM can consequently enable the development of autonomic management strategies. The experiment results demonstrate the DAIM structure can be one of the architectural approaches to creating the autonomic service management for SDN. Moreover, DAIM can possibly be implemented as a generic model with a multi-agent system to become a truly autonomic communications network. The DAIM structure can be utilised to investigate the functionalities required by the autonomic networking within the ACNs community. The DAIM model can be further applied to enable adaptability and autonomy to other distributed networks such as WSNs, P2P networks and Ad-Hoc sensor networks.

6.2. Research Contributions and Findings

The model, distributed architecture and system implementation discussed in this thesis are considered as contributions and findings to current OpenFlow networks.

In essence, an alternative solution to distribute control functions across OpenFlow switches is proposed, implemented, validated and documented in the published papers. Apart from the original primary objectives of the thesis being satisfied, a number of additional contributions and findings to the original objectives were achieved. The innovation and novelty of these contributions include the following:

1. Review of literature on network management for distributed systems in NGN networks. This contribution provides the needs of distributed systems in NGN networks. A Springer book chapter has been published as a contribution to the knowledge [84]. (subsection 2.2.3 on page 29)

2. Review of literature on Software-Defined Networking (SDN) paradigm. This contribution presents the state of the art in programmable networks based on the standard OpenFlow protocol version 1.0. It also provides an in-depth analysis of the ongoing research efforts and challenges of SDN. (section 2.4 on page 37)

3. A comparative review of accurate OpenFlow simulation and emulation tools is conducted. Mininet is used to implement the DAIM model due to its open access for researchers which provides a realistic setting for SDN-based environment. A journal paper has been published as a contribution to the knowledge [81]. (subsection 2.4.6 on page 56)

4. A new DAIM model is proposed as an alternative to the previous efforts on object-oriented management information base (O:MIB) and *hybrid O:XML* semantics [31]. It has been seamlessly integrated into the SDN architecture at the level of the switches to provide a logically distributed control plane. A journal paper and two conference papers based on this model have been published as contributions to the knowledge [9], [83], [79]. (chapter 3 on page 73)

5. This DAIM model is implemented from scratch based on its theory and designed architecture. A new message channel is created to facilitate the communications between the controller and OpenFlow switches, which is implemented using UNIX BSD socket programming API to process OpenFlow messages without any modification. (subsection 4.2.1 on page 97)

6. An intelligent L2 switch application is developed in DAIM to control packet flows in an OpenFlow switch. (subsection 4.2.2 on page 98)

7. Introduction to the cross-controller communication protocol based on the LLDP mechanism. (subsection 4.2.3 on page 99)

8. Three core modules that construct the DAIM ecosystem are further implemented. Inside the *communication* module, there are various routines for creating different types of OpenFlow messages. The *storage* module is built to store information of network elements. The *controller* module is responsible for all control functions. (section 4.3 on page 101)

9. The feasibility of using DAIM in different OpenFlow scenarios has been tested, discovered and documented. Additionally, experiments are carried out to verify the functionality of DAIM and check the compliance of the overall system with its purpose and functions. (section 4.5 on page 123)
10. Different performance metrics including mean RTT, throughput, latency and bandwidth have been tested and discovered by the used of available and appropriate benchmarking tools (*Cbench* and *Iperf*). (subsection 5.3.1 on page 137)
11. The experimental validation of DAIM communication channel is conducted and shows a good performance in terms of flow setup speed. (subsection 5.4.1 on page 143)
12. The performance evaluation of the DAIM controller is carried out and shows a better performance with respect to throughput and latency. The performance of DAIM is compared with NOX, POX, and NOX-MT. (subsection 5.4.2 on page 149)
13. A small-size OpenFlow network has already been set up physically in the lab to illustrate that DAIM can perform operational functions and can be deployed in real networks. Performance of DAIM on hardware has been tested, analysed and documented. (section 5.5 on page 154)
14. There are different factors such as programming language and device resources influencing the performance of an OpenFlow controller. The important parameters that can affect the performance evaluation have been analysed. (section 5.6 on page 160)
15. A reference implementation of the DAIM model is made available to help further research, trials and validation of the performance of scalability and robustness for other researcher's use. It is accessible on the public domain (GitHub), and is considered as a contribution to the SDN and the networking research community. (Appendix A on page 185)

6.3. Research Limitations

Although the DAIM model is successfully implemented and the outcomes of the research in this dissertation are properly justified, there are some limitations as follows:

- The OpenFlow test lab is constructed on a limited network scale because of the limited research resources, time and efforts. In the future with available resources, a more sophisticated network architecture can be re-constructed in the test lab to host novel research scenarios. The performance analysis of the DAIM controller based on hardware (Raspberry Pi's) is briefly discussed in subsection 5.5.4 on page 158. It has been observed that deploying more

computing power and using other high-end networking devices can significantly improve DAIM's performance. This is because the Raspberry Pi does not have enough available resources to manage the flow entries quickly. For example, the throughput of DAIM running on a server with two quad-core 3GHz processors can achieve a maximum of 162,519 responses per second compared to 27,139 with DAIM running on a Raspberry Pi. Furthermore, more experiments from real large-scale networks are required in order to obtain more practical results.

- In DAIM's current implementation, there is no immediate link failover recovery. If network environment changes such as a switch port is down, all active flows associated with that port are not re-routed. These flows are not removed from the flow table because as packets keep arriving the flow *idle-timeout* gets reset. If the packet stream stops, then the flow times out. Another path to the destination is reinstalled when the packet stream starts again. All of this can possibly be resolved by using OpenFlow protocol version 1.1. However, failover scenarios are not in the scope of this research and OpenFlow version 1.1 has not been developed for the test hardware.
- Because each DAIM controller is designed to manage only one dedicated switch, it is not applicable to analyse the correlation between the DAIM's throughput and the number of switches ("N" switches) connected to the controller. As a result, Cbench evaluation tool can be used to test the performance of DAIM with only one connected switch. Increasing the number of connected hosts ("N" MACs) is required in order to test the maximum number of Packet_In messages that the controller can handle. To provide more fine-tuning of test parameters, the traditional Cbench code needs to be modified as our own solution and benchmark for OpenFlow controllers testing.

6.4. On-going Work and Future Directions

This research work on a new information model that allows logical distribution of the control plane for SDN opens several new avenues for research. Although the DAIM model has been seamlessly integrated into the SDN architecture, DAIM is far from perfect and can be optimised in various ways. Further investigative research is suggested to aim at improvement in the following areas:

- The performance of the DAIM model can be optimised not only to maximise flow setup throughput but also to maintain low request-handling latency. This can be achieved by using well-known optimisation techniques. Examples of these are using a fast multiprocessor-aware malloc implementation that will scale effectively in a multi-core machine, using a larger read buffer (batching) to minimise the overhead of I/O, decreasing the number of system calls by reading/writing more bytes in every socket call and reducing the number of application executions. DAIM can also be optimised to read and process

OpenFlow messages as a fully multi-threaded event loop in order to achieve high performance.

- A multi-agent based system essentially has the properties of *intelligence* and *autonomics*. JADE and its extension JadeX are Java agent development platforms that aim to standardise the multi-agent technology for various users. Both of them are also FIPA compliance agent systems. Developing the intelligent DAIM agents based on these platforms will be carried out in the next stage work. The main goal is to create the DAIM agents with more advanced and autonomous functionalities such as collecting local link information to construct the entire network graph and computing the best path to the designated destination. Furthermore, it is very important for each DAIM controller to communicate with one another. Therefore, developing an efficient controller-to-controller communication and protocol will be highly beneficial for large-scale networks where the controller instances share massively dispersed information to ensure fine-grained network wide consistency.
- Regarding the functionalities of DAIM that have been developed so far, attention must be paid to the fact that more test scenarios such as failover and recovery can be conducted to increase fault tolerance. Currently, DAIM is not developed to support immediate link failover recovery. One promising solution to solve this issue in the current version is to maintain a local cache of all installed flow entries. When the controller receives information on a broken link, all affected flow entries are removed from those active switches. After the removal, no flow is available for subsequent flows and hence a re-calculation is triggered to produce a new active path. OpenFlow specification version 1.1 has introduced the concepts of failover groups in which the switch takes a failover decision to another interface without having to wait for the controller. Unfortunately, the author has discovered this at the end of the research. Thus, adopting later versions of the OpenFlow protocol will be absolutely useful and highly valuable for DAIM.
- Finally, outcomes of the DAIM model that have been designed and implemented through this dissertation can also be used by other related research, such as the current distributed systems research. The reference design of DAIM implementation including a number of open source components can be utilised in further experiments relating to distributed networking as well as cloud-based networking. This will facilitate other researchers in conducting further research on the model, network architecture, protocols and theory described in this work.

Bibliography

[1] The IETF policy framework working group: Charter. *available at http://www.ietf.org/html.charters/policy-charter.html technical report.*

[2] International organization for standardization. *http://www.iso.org. Technical Report.*

[3] CIM concepts whitepaper, CIM version 2.4+. *available at http://www.dmtf.org/standards/cim. technical report, DMTF*, 2003.

[4] R.; Al-Obasiat Y.; Braun. A Multi-Agent Flexible Architecture for Autonomic Services and Network Management; Computer Systems and Applications. *Computer Systems and Applications, 2007. AICCSA '07. IEEE/ACS International Conference on;*, pages 124–131, 2007.

[5] Abdallah Al Sabbagh, Pakawat Pupatwibul, Ameen Banjar, and Robin Braun. Optimization of the openflow controller in wireless environments for enhancing mobility. In *Local Computer Networks Workshops (LCN Workshops), 2013 IEEE 38th Conference on*, pages 930–935. IEEE, 2013.

[6] Siamak Azodolmolky. *Software Defined Networking with OpenFlow.* Packt Publishing Ltd, 2013.

[7] Abhishek Bagewadi and KN Rama Mohan Babu. Towards an ethernet learning switch and bandwidth optimization using pox controller.

[8] Ameen Banjar, Pakawat Pupatwibul, Abdallah AL Sabbagh, and Robin Braun. Using an icn approach to support multiple controllers in openflow. *International Journal of Electrical & Computer Sciences*, 14(2), 2014.

[9] Ameen Banjar, Pakawat Pupatwibul, and Robin Braun. DAIM: a mechanism to distribute control functions within openflow switches. *Journal of Networks*, 9(01):1–9, 2014.

[10] Ameen Banjar, Pakawat Pupatwibul, and Robin Braun. Comparison of TCP/IP routing versus openflow table and implementation of intelligent computational model to provide autonomous behavior. In *Computational Intelligence and Efficiency in Engineering Systems*, pages 121–142. Springer, 2015.

[11] Ameen Banjar, Pakawat Pupatwibul, Robin Braun, and Bruce Moulton. Analysing the performance of the openflow standard for software-defined networking using the omnet++ network simulator. In *Computer Aided System Engineering (APCASE), 2014 Asia-Pacific Conference on*, pages 31–37. IEEE, 2014.

[12] Paul Baran. On distributed communications networks. *Communications Systems, IEEE Transactions on*, 12(1):1–9, 1964.

[13] Jaxon:java based openflow controller. http://jaxon.onuos.org/.

[14] Leonardo Richter Bays and Daniel Stefani Marcon. Flow based load balancing: Optimizing web servers resource utilization. *Journal of Applied Computing Research*, 1(2):76–83, 2011.

[15] Beacon. https://openflow.stanford.edu/display/beacon/home.

[16] Fouad Benamrane, Redouane Benaini, et al. Performances of openflow-based software-defined networks: An overview. *Journal of Networks*, 10(6):329–337, 2015.

[17] A. Bianco, R. Birke, L. Giraudo, and M. Palacin. Openflow switching: Data plane performance. pages 1–5. IEEE.

[18] Kenneth P Birman. *Reliable distributed systems: technologies, web services, and applications*. Springer Science & Business Media, 2005.

[19] Raouf Boutaba and Andreas Polyrakis. Projecting fcaps to active networks. In *Enterprise Networking, Applications and Services Conference Proceedings*, pages 97–104. IEEE, 2001.

[20] R. Braun and F. Chiang. A distributed active information model enabling distributed autonomics in complex electronic enviornments. In *Broadband Communications, Information Technology & Biomedical Applications, 2008 Third International Conference on*, pages 473–479.

[21] Helios by NEC. http://www.nec.com/.

[22] Z. Cai, AL Cox, and TSE Ng. Maestro: A system for scalable openflow control. Report, Technical Report TR10-08, Rice University, December 2010.

[23] Marco Canini, Daniele Venzano, and D Kostic. A Nice Way to Test OpenFlow Applications. *NSDI, Apr*, 2012.

[24] Jeffrey D Case, Mark Fedor, Martin L Schoffstall, and James Davin. Simple network management protocol (SNMP). Technical report, 1990.

[25] Yanan Chang, Debao Xiao, Hui Xu, and Limiao Chen. Design and implementation of netconf-based network management system. In *Future Generation Communication and Networking, 2008. FGCN'08. Second International Conference on*, volume 1, pages 256–259. IEEE, 2008.

[26] F. Chiang and R. Braun. Self-adaptability and vulnerability assessment of secure autonomic communication networks. *Managing Next Generation Networks and Services*, pages 112–122, 2007.

[27] F. Chiang and R. Braun. Towards a management paradigm with a constrained benchmark for autonomic communications. *Computational Intelligence and Security*, pages 250–258, 2007.

[28] F. Chiang, R. Braun, and J. Hughes. A biologically-inspired multi-agent framework for autonomic service management. *International Journal of Pervasive Computing and Communications*, 2(3):261–276, 2007.

[29] F. Chiang, H. Fernandez, R. Braun, and J. Agbinya. Integrating object-oriented O:XML semantics into autonomic decentralised functionalities. In *7th International Symposium on Communications and Information Technologies*, pages 768–773. IEEE.

[30] F. Chiang and V. Mahadevan. Towards the distributed autonomy in complex environments. In *Information and Multimedia Technology, 2009. ICIMT'09. International Conference on*, pages 169–172. IEEE.

[31] Frank Chiang. Self-adaptability, resilience and vulnerability on autonomic communications with biology-inspired strategies. *PhD thesis University of Technology Sydney, Australia*, 2008.

[32] Frank Chiang, Johnson Agbinya, and Robin Braun. Risk and vulnerability assessment of secure autonomic communication networks. In *Wireless Broadband and Ultra Wideband Communications, 2007. AusWireless 2007. The 2nd International Conference on*, pages 40–40. IEEE, 2007.

[33] OpenFlow Switch Consortium. Openflow switch specification version 1.1. 0. Report, Tech. Rep., February, 2011.

[34] A.R. Curtis, J.C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. Devoflow: Scaling flow management for high-performance networks. *SIGCOMM-Computer Communication Review*, 41(4):254, 2011.

[35] S. Dobson, S. Denazis, A. Fernandez, D. Gaiti, E. Gelenbe, F. Massacci, P. Nixon, F. Saffre, N. Schmidt, and F. Zambonelli. A survey of autonomic communications. *Acm Transactions on Autonomous and Adaptive Systems*, 1(2):223–259, 2006. V03zn Times Cited:112 Cited References Count:165.

[36] David Erickson. The beacon openflow controller. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 13–18. ACM, 2013.

[37] ZhengCai AlanL Cox TS EugeneNg. Maestro: Balancing fairness, latency and throughput in the openflow control plane.

[38] T. Feng, J. Bi, H. Hu, and H. Cao. Networking as a service: a cloud-based network architecture. *Journal of Networks*, 6(7):1084–1090, 2011.

[39] Marcial Fernandez. Evaluating openflow controller paradigms. In *ICN*, pages 151–15, 2013.

[40] Rodriguez H. Fernandez. Active MIB, an object oriented solution for network management. *Master thesis, Chalmers University of Technology*, 2007.

[41] Floodlight. an open SDN controller. http://floodlight.openflowhub.org/.

[42] Pantou: Openflow 1.0 for openwrt. http://www.openflow.org/wk/index.php/open-flow 1.0 for openwrt.

[43] N. Foster, M.J. Freedman, R. Harrison, J. Rexford, M.L. Meola, and D. Walker. Frenetic: a high-level language for openflow networks. In *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow*, page 6. ACM.

[44] Open Networking Foundation. Software-defined networking: The new norm for networks. *ONF WHITE PAPER*, April 2012.

[45] Chen Gang, Yang Baofei, Zhou Wenan, and Song Junde. Research on service management data modeling based on sid. In *Vehicular Technology Conference, 2007. VTC-2007 Fall. 2007 IEEE 66th*, pages 1995–1999.

[46] Paul Goransson and Chuck Black. *Software Defined Networks: A Comprehensive Approach*. Elsevier, 2014.

[47] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. Nox: towards an operating system for networks. *ACM SIGCOMM Computer Communication Review*, 38(3):105–110, 2008.

[48] Soheil Hassas Yeganeh and Yashar Ganjali. Kandoo: a framework for efficient and scalable offloading of control applications. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 19–24. ACM, 2012.

[49] IBM. *TXSeries for Multiplatforms: Concepts and Planning, 5th edn.* International Business Machines Corporation, first edition edition, November 2005.

[50] Iperf:. http://iwl.com/white-papers/iperf.

[51] M. Jarschel, S. Oechsner, D. Schlosser, R. Pries, S. Goll, and P. Tran-Gia. Modeling and performance evaluation of an openflow architecture. In *Proceedings of the 23rd International Teletraffic Congress*, pages 1–7. ITCP.

[52] Michael Jarschel, Frank Lehrieder, Zsolt Magyari, and Rastin Pries. A flexible openflow-controller benchmark. In *Software Defined Networking (EWSDN), 2012 European Workshop on*, pages 48–53. IEEE, 2012.

[53] J.O. Kephart and D.M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[54] Dominik Klein and Michael Jarschel. An openflow extension for the omnet++ inet framework. 2013.

[55] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, and T. Hama. Onix: A distributed control platform for large-scale production networks. *OSDI, Oct*, 2010.

[56] Diego Kreutz, Fernando MV Ramos, PE Verissimo, C Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-defined networking: A comprehensive survey. *proceedings of the IEEE*, 103(1):14–76, 2015.

[57] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, page 19. ACM.

[58] Dan Levin, Andreas Wundsam, Brandon Heller, Nikhil Handigol, and Anja Feldmann. Logically centralized?: state distribution trade-offs in software defined networks. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 1–6. ACM, 2012.

[59] KA Manjula and P Karthikeyan. Distributed computing approaches for scalability and high performance. *International Journal of Engineering Science and Technology*, 2(6):2328–2336, 2010.

[60] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[61] Jan Medved, Robert Varga, Anton Tkacik, and Ken Gray. Opendaylight: Towards a model-driven SDN controller architecture. In *2014 IEEE 15th International Symposium on*, pages 1–6. IEEE, 2014.

[62] Kraig Meyer, Mike Erlinger, Joe Betser, Carl Sunshine, Germán Goldszmidt, and Yechiam Yemini. Decentralizing control and intelligence in network management. In *Integrated Network Management IV*, pages 4–16. Springer, 1995.

[63] Jeffrey C Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, Andrew R Curtis, and Sujata Banerjee. Devoflow: Cost-effective flow management for high performance enterprise networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, page 1. ACM.

[64] Tatsuya Mori, Masato Uchida, Ryoichi Kawahara, Jianping Pan, and Shigeki Goto. Identifying elephant flows through periodically sampled packets. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, pages 115–120. ACM, 2004.

[65] Tore Mørkved. Peer-to-peer programming with wireless devices. 2005.

[66] M. Muck, S. Gault, V. Merat, E. Patouni, Z. Boufidis, A. Lilis, N. Alonistioti, B. Steinke, R. Feuillette, P. Martigne, E. Alexandri, O. Holland, G. Bartolomeo, P. Demestichas, G. Dimitrakopoulos, K. Tsagkaris, V. Stavroulaki, F. Zhiyong, and H. Sharma. Autonomic communications for heterogeneous wireless communication systems technical report. *https://ict-e3.eu/project/white_papers/e2r/6.E2RII_Autonomics_White_Paper.pdf*.

[67] Mul. http://sourceforge.net/p/mul/wiki/home/.

[68] Thomas D Nadeau and Ken Gray. *SDN: Software Defined Networks.* " O'Reilly Media, Inc.", 2013.

[69] J. Naous, D. Erickson, G.A. Covington, G. Appenzeller, and N. McKeown. Implementing an openflow switch on the netfpga platform. pages 1–9. ACM.

[70] Marcelo R Nascimento, Christian E Rothenberg, Marcos R Salvador, Carlos NA Corrêa, Sidney C de Lucena, and Maurício F Magalhães. Virtual routers as a service: the routeflow approach leveraging software-defined networks. In *Proceedings of the 6th International Conference on Future Internet Technologies*, pages 34–37. ACM, 2011.

[71] P. NathRK. Internet technology with client server architecture (2010). *http://www.data-e-education.com/E079_Centralized_Network_Architecture.html/*, 2010. [Online; accessed April 17, 2012].

[72] The nodeflow openflow controller. http://garyberger.net/?p=537.

[73] Node.js. http://nodejs.org/.

[74] B Nunes, Marc Mendonca, X Nguyen, Katia Obraczka, and Thierry Turletti. A survey of software-defined networking: Past, present, and future of programmable networks. 2014.

[75] ofsoftswitch13 cpqd. https://github.com/cpqd/ofsoftswitch13.

[76] Trema openflow controller framework. https://github.com/trema/trema.

[77] B. Otero, S. Sahuquillo, P. Barlet-Ros, S. Spadaro, and J. Solé-Pareta. Self-* algorithms and autonomic communication systems. *Internal publication of Technical university of Catalonia.*

[78] POX. http://www.noxrepo.org/pox/about-pox/.

[79] P. Pupatwibul, B. Jozi, and R. Braun. Investigating O:MIB-based distributed active information model (DAIM) for autonomics. In *International Conference on Information and Communication Technologies and Applications*, pages 7–12. IIIS.

[80] Pakawat Pupatwibul, Ameen Banjar, Abdallah Al Sabbagh, and Robin Braun. Developing an application based on openflow to enhance mobile IP networks. In *Local Computer Networks Workshops (LCN Workshops), 2013 IEEE 38th Conference on*, pages 936–940. IEEE.

[81] Pakawat Pupatwibul, Ameen Banjar, Abdallah AL Sabbagh, and Robin Braun. A comparative review: Accurate openflow simulation tools for prototyping. *Journal of Networks*, 10(5):322–327, 2015.

[82] Pakawat Pupatwibul, Ameen Banjar, and Robin Braun. Performance evaluation of TCP/IP vs. openflow in inet framework using omnet++, and implementation of intelligent computational model to provide autonomous behaviour. In *The Asian Conference on Technology, Information & Society 2014*, pages 43–56. The International Academic Forum (IAFOR).

[83] Pakawat Pupatwibul, Ameen Banjar, and Robin Braun. Using DAIM as a reactive interpreter for openflow networks to enable autonomic functionality. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, pages 523–524. ACM.

[84] Pakawat Pupatwibul, Ameen Banjar, Abdallah AL Sabbagh, and Robin Braun. *An Intelligent Model for Distributed Systems in Next Generation Networks*, pages 315–334. Springer, 2014.

[85] Pakawat Pupatwibul, Abdallah AL Sabbagh, Ameen Banjar, and Robin Braun. Distributed systems in next generation networks. In *1st Australian Conference on the Applications of Systems Engineering ACASE'12*, page 32, 2012.

[86] Umesh Hodeghatta Rao and Sanjay Mohapatra. Deploying network management solutions in enterprises. In *Networked Computing (INC), 2010 6th International Conference on*, pages 1–6. IEEE, 2010.

[87] Jordi Perez Romero, Oriol Sallent, Ramon Agusti, and Miguel Angel Diaz-Guerra. *Radio resource management strategies in UMTS*. John Wiley & Sons, 2005.

[88] Charalampos Rotsos, Nadi Sarrar, Steve Uhlig, Rob Sherwood, and Andrew W Moore. OFLOPS: An open framework for openflow switch evaluation. In *Passive and Active Measurement*, pages 85–95. Springer, 2012.

[89] Ryu. http://osrg.github.com/ryu/.

[90] AA Sabbagh, Robin Braun, and Mehran Abolhasan. A comprehensive survey on rat selection algorithms for heterogeneous networks. 2011.

[91] Robert R Schaller. Moore's law: past, present and future. *Spectrum, IEEE*, 34(6):52–59, 1997.

[92] Alexander Shalimov, Dmitry Zuikov, Daria Zimarina, Vasily Pashkov, and Ruslan Smeliansky. Advanced study of sdn/openflow controllers. In *Proceedings of the 9th Central & Eastern European Software Engineering Conference in Russia*, page 1. ACM, 2013.

[93] R. Sherwood, G. Gibb, K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar. Flowvisor: A network virtualization layer. *OpenFlow Switch Consortium, Tech. Rep*, 2009.

[94] Rob Sherwood, Michael Chan, Adam Covington, Glen Gibb, Mario Flajslik, Nikhil Handigol, Te-Yuan Huang, Peyman Kazemian, Masayoshi Kobayashi, Jad Naous, et al. Carving research slices out of your production networks with openflow. *ACM SIGCOMM Computer Communication Review*, 40(1):129–130, 2010.

[95] Rob Sherwood, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martin Casado, Nick McKeown, and Guru M Parulkar. Can the production network be the testbed? In *OSDI*, volume 10, pages 1–6, 2010.

[96] Rob Sherwood and YAP KOK-KIONG. Cbench: an open-flow controller benchmarker. *2013-05-13]. http://www, openflow, org/wk/index, php/Oflops*, 2010.

[97] Vishal Shukla. *Introduction to Software Defined Networking - Openflow & Vxlan (Paperback)*. CreateSpace Independent Publishing Platform, North Charleston, SC, 2013.

[98] Simple Network Access Control (SNAC). http://www.openflow.org/wp/snac/.

[99] Joris Soeurt and Iwan Hoogendoorn. Shortest path forwarding using openflow. *University of Amsterdam, February*, 13:2011–2012, 2012.

[100] Indigo: Open source openflow switches. http://www.openflowhub.org/display/indigo/.

[101] R. Sterritt and D.F. Bantz. Personal autonomic computing reflex reactions and self-healing. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 36(3):304–314, 2006.

[102] R. Sterritt and D. Bustard. Towards an autonomic computing environment. In *14th International Workshop on Database and Expert Systems Applications*, pages 694–698. IEEE.

[103] John Strassner, Sven van der Meer, and James Won-Ki Hong. The applicability of self-awareness for network management operations. In *Modelling Autonomic Communications Environments*, pages 15–28. Springer, 2009.

[104] John C Strassner and Yacine M Ghamri-Doudane. *Modelling Autonomic Communications Environments*. Springer, 2009.

[105] Lalith Suresh, Julius Schulz-Zander, Ruben Merz, Anja Feldmann, and Teresa Vazao. Towards programmable enterprise wlans with odin. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 115–120. ACM, 2012.

[106] Amin Tootoonchian and Yashar Ganjali. Hyperflow: A distributed control plane for openflow. In *Proceedings of the 2010 internet network management conference on Research on enterprise networking*, pages 3–3. USENIX Association.

[107] Amin Tootoonchian, Sergey Gorbunov, Yashar Ganjali, Martin Casado, and Rob Sherwood. On controller performance in software-defined networks. In *USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE)*, volume 54, 2012.

[108] Google Code University. Introduction to distributed system design. *http://-code.google.com/edu/parallel/dsd-tutorial.html*. [Online; accessed July 22, 2011].

[109] A. Varga. Inet framework for the omnet++ discrete event simulator. *http://-github. com/inetframework/inet*, 2012.

[110] András Varga. The omnet++ discrete event simulation system. In *Proceedings of the European Simulation Multiconference (ESM'2001)*, volume 9, page 185. sn.

[111] András Varga and Rudolf Hornig. An overview of the omnet++ simulation environment. In *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, page 60. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[112] Dinesh C Verma. Simplifying network administration using policy-based management. *Network, IEEE*, 16(2):20–26, 2002.

[113] NS-3 version 3.16. OpenFlow switch support. http://www.nsnam.org/docs/release/3.16/models/html/openflow-switch.html. December 2012.

[114] Open vswitch and ovs controller. http://openvswitch.org/.

[115] Shie-Yuan Wang, Chih-Liang Chou, and Chun-Ming Yang. Estinet openflow network simulator and emulator. *Communications Magazine, IEEE*, 51(9):110–117, 2013.

[116] Julius Werner. *Description of Network Research Enablers on the Example of OpenFlow New Network Architectures*, volume 297 of *Studies in Computational Intelligence*, pages 167–177. Springer Berlin / Heidelberg, 2010.

[117] Baozhen Wu and Yanan Chang. Integrating SNMP agents and CLI with NETCONF-based network management systems. In *Computer Science and Information Technology (ICCSIT), 2010 3rd IEEE International Conference on*, volume 1, pages 81–84. IEEE, 2010.

[118] K. K. Yap. Openflowvms simulating openflow networks. *http://www. openflow. org/wk/index.php/OpenFlowVMS.*, March 2011.

[119] James Yu and Imad Al Ajarmeh. An empirical study of the netconf protocol. In *Networking and Services (ICNS), 2010 Sixth International Conference on*, pages 253–258. IEEE, 2010.

[120] Minlan Yu, Jennifer Rexford, Michael J Freedman, and Jia Wang. Scalable flow-based networking with difane. *ACM SIGCOMM Computer Communication Review*, 41(4):351–362, 2011.

A. DAIM Source Code for Data Analysis

The reference implementation of the DAIM controller can be downloaded from GitHub link below:

https://github.com/ndomon/DAIM.git

To start the DAIM controller running with Mininet follow these steps:

Step 1: download the DAIM controller distribution and compile with the following commands:

```
~$ gcc -c ani.cpp com.cpp con.cpp linked.cpp
```

```
~$ gcc ani.o com.o con.o linked.o -o aniv21 -lstdc++ -lpthread
```

Step 2: change directory to the executable file and enter *./aniv21 -h* to see the documented arguments.

```
~$ cd ANIv21/dist/Debug/GNU-Linux-x86/
```

```
~/ANIv21/dist/Debug/GNU-Linux-x86$ ./aniv21 -help
Usage:
    -p, --port controller port e.g. -p 2000
    -q, --quite turn off verbose mode
    -c, --cbench Cbench benchmarking mode
    -v, --version show ANI version
    -h, --help show this help
```

Step 3: start a default Mininet topology with one switch and two hosts to talk to the DAIM controller:

```
~$ sudo mn --controller remote
```

This command will create a network with a connection to the remote controller at 127.0.0.1:6633

Step 4: run a simple learning switch using a DAIM controller framework:

```
~/ANIv21/dist/Debug/GNU-Linux-x86$ ./aniv21 -p 6633
```

Note that every OpenFlow switch in the network must be connected and managed by at least one DAIM controller.

After a while, the OpenVswitch in Mininet will get connected to the DAIM controller. The output from this controller after a *pingall* command should look like the following:

```
Info: creating ANI socket...
Info: established ANI server!
Info: waiting for the switch connection...
Info: connected to the switch!
Total host entries freed: -1
Total port entries freed: -1
Info: sent hello to the switch
Info: features request sent to the switch
Info: received packet from the switch
Info: OpenFlow packet detected
Info: echo reply sent to the switch
Info: received packet from the switch
Info: OpenFlow packet detected
Host MAC: FA:3A:FD:D8:1C:16 Ingress Port: 1 Datapath: 0
Info: ARP broadcast
Info: received packet from the switch
Info: OpenFlow packet detected
Host MAC: FA:3A:FD:D8:1C:16 Ingress Port: 1 Datapath: 0
Host MAC: 66:2C:4D:DF:97:89 Ingress Port: 2 Datapath: 0
Info: host found in the table
Info: ARP flow mod
Info: received packet from the switch
Info: OpenFlow packet detected
Info: host found in the table
Info: ICMP flow mod
Info: received packet from the switch
Info: OpenFlow packet detected
Info: echo reply sent to the switch
Info: host already in the table
Info: host found in the table
Info: ARP flow mod
Info: received packet from the switch
Info: OpenFlow packet detected
Info: echo reply sent to the switch
^C
Info: caught signal, Ctrl + C: 2
Total host entries freed: 2
Total port entries freed: -1
```

A.1. Cross-controller Communications

The following C++ code is emphasised to show how the DAIM controllers communicate and share their information. Inside the controller, this functionality is implemented using a separate thread to send messages independently. The following code fraction shows the body of the separate thread function:

```
1  void *node_share_func (void *message)
2  {
3      bool threadl = false;
4      int ports_count = -1;
5      uint64_t datap;
6      ports_count = ((struct switch_info *)message)->no_of_ports;
7      datap = ((struct switch_info *)message)->datapath_id;
8      send_anim:
9      sleep (10);
10     for (int portno = 1; portno < (ports_count + 1); portno++)
11 {
12     struct ofp_packet_out *n_packet_out = NULL;
13     struct ofp_action_output *n_action_output = NULL;
14     struct of_ethernet *n_packet_out_eth = NULL;
15     char eth_buffer[1024];
16     char n_buffer[10240];
17     char message[1024];
18     long sw_write_len = -1;
19     memset(n_buffer, '\0', sizeof (n_buffer));
20     memset(eth_buffer, '\0', sizeof (eth_buffer));
21     n_packet_out_eth = (struct of_ethernet *) eth_buffer;
22     n_packet_out_eth->src[0] = 0;
23     n_packet_out_eth->src[1] = 0;
24     n_packet_out_eth->src[2] = 0;
25     n_packet_out_eth->src[3] = 0;
26     n_packet_out_eth->src[4] = 0;
27     n_packet_out_eth->src[5] = 1;
28     n_packet_out_eth->dst[0] = 1;
29     n_packet_out_eth->dst[1] = 1 << 5 | 1 << 1 | 1;
30     n_packet_out_eth->dst[2] = 1 << 5;
31     n_packet_out_eth->dst[3] = 0;
32     n_packet_out_eth->dst[4] = 0;
33     n_packet_out_eth->dst[5] = 1;
34     n_packet_out_eth->type = htons(LLDP_DATA);
35     n_packet_out = (struct ofp_packet_out *) n_buffer;
36     n_packet_out->header.version = OFP_VERSION;
37     n_packet_out->header.type = OFPT_PACKET_OUT;
38     n_packet_out->header.xid = htonl (0);
39     n_packet_out->buffer_id = htonl (0xffffffff);
40     n_packet_out->in_port = htons (OFPP_CONTROLLER);
41     n_packet_out->actions_len = htons (sizeof (struct
         ofp_action_output));
42     n_action_output = (struct ofp_action_output *) n_packet_out->
         actions;
43     n_action_output->type = htons (OFPAT_OUTPUT);
```

```
44      n_action_output->max_len = htons (0);
45      n_action_output->len = htons (8);
46      n_action_output->port = htons (portno);
47      memcpy (n_buffer + sizeof (struct ofp_packet_out) + sizeof (
          struct ofp_action_output), eth_buffer, sizeof (struct
          of_ethernet));
48      memset (message, '\0', sizeof(message));
49      sprintf(message, "%s %u", "Hello, I am controller from datapath
          ", datap);
50      memcpy (n_buffer + sizeof (struct ofp_packet_out) + sizeof (
          struct ofp_action_output) + sizeof (struct of_ethernet),
          message, sizeof(message));
51      n_packet_out->header.length = htons (sizeof (struct
          ofp_packet_out) + sizeof (struct ofp_action_output) + sizeof (
          struct of_ethernet) + sizeof(message));
52      sw_write_len = send(sw_sockfd, n_buffer, ntohs (n_packet_out->
          header.length), 0);
53      if (sw_write_len < 1) cerr << "Error: can not send controller
          message to port " << portno << ", " << strerror(errno) << endl;
54      else cout << "Info: controller message send to port " << portno
          << endl;
55      sleep (1);
56      }
57      goto send_anim;
58  }
```

Therefore, each controller sends a message containing its datapath ID to every available port on the switch. The controllers that are connected to these ports will receive these messages. To facilitate how OpenFlow packets flow through the network and better explanation, some Wireshark captures are taken and labeled with related information. For example, in a simple linear topology with two switches and two hosts, Figure A.1 shows how messages from *DAIM 1* sent to *DAIM 2* will be seen on the wire.

Similarly from the other direction, messages from *DAIM 2* sent to *DAIM 1* can be seen on the wire in Figure A.2.

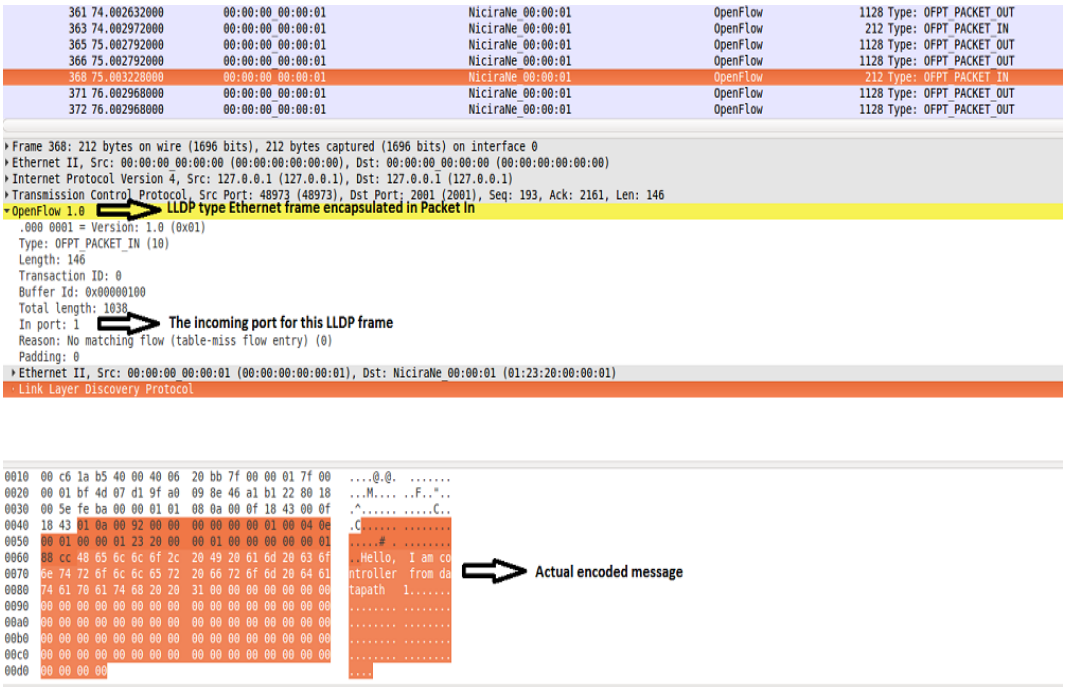


Figure A.1.: Controller Message from DAIM 1 to DAIM 2

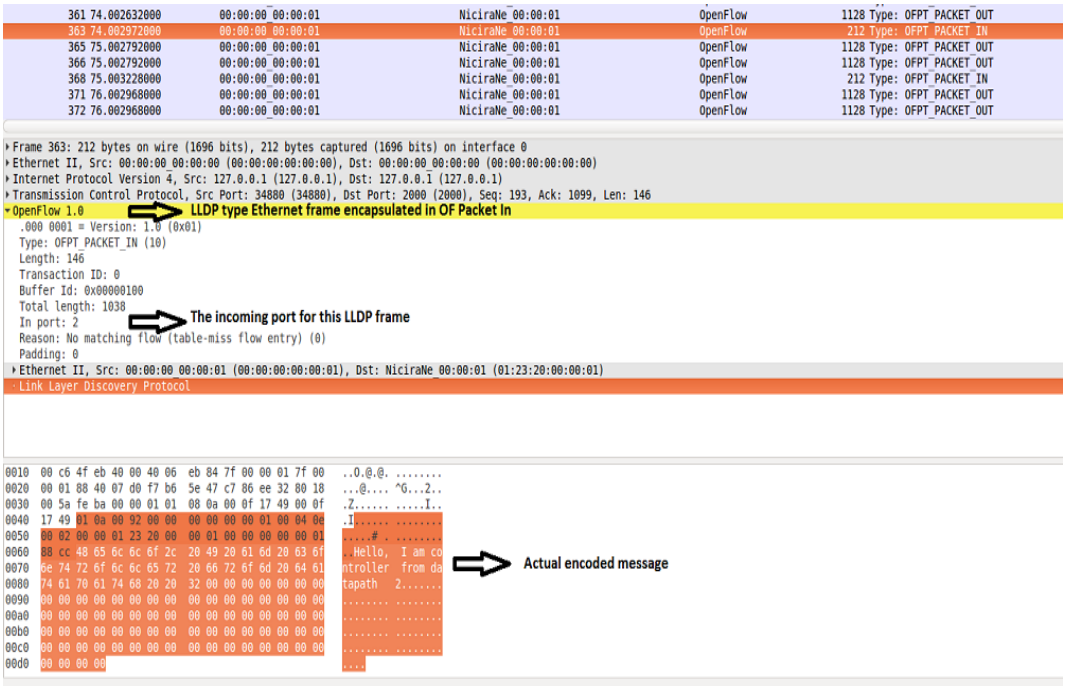


Figure A.2.: Controller Message from DAIM 2 to DAIM 1

B. Create OpenFlow Network with Multiple PCs

This section describes how to set up a small Linux-based OpenFlow network controlled by NOX with a set of PCs in half day. We created a minimal OpenFlow network setup of five PCs with two OpenFlow switches controlled by an NOX 0.9.0 controller, and two hosts (see Figure B.1). These two OpenFlow switches are PC based software OpenFlow switches (OpenVswitch implementation). We run OpenFlow reference design on those PCs. Any PC running Linux 2.6 kernel would work, but there are some dependencies on the Linux distribution.

B.1. Configuration Summary

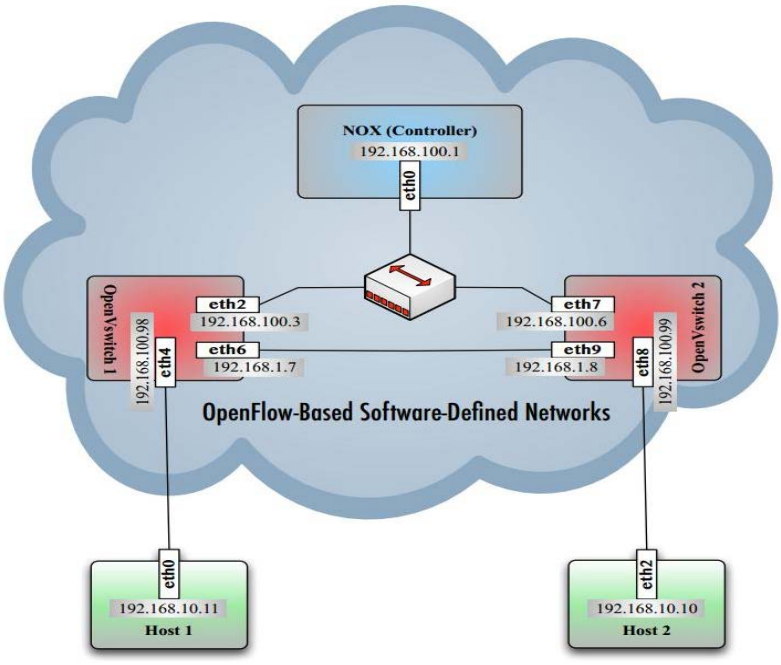


Figure B.1.: OpenFlow-Based SDN Lab Using OpenVswitch and Controlled by NOX via OpenFlow Protocol

OpenFlow Switches

OpenVswitch	IP address to talk to the controller	Controller Ethernet Port	OpenFlow Ethernet Ports	Host Ethernet Port
OpenVswitch1	192.168.100.3 /255.255.255.0	eth 2	eth 6	eth 4
OpenVswitch2	192.168.100.6 /255.255.255.0	eth 7	eth 9	eth 8

OpenFlow Controller (NOX 0.9.0)

Bridge IP Address	TCP Port Number
192.168.100.1/255.255.255.0	6633

B.2. Assigning Static IP Address for Network Interfaces

Edit the network interface file by entering the following command into the terminal:

```
$sudo gedit /etc/network/interfaces
```

When the text editor opens, enter the address of each interface according to the topology as follows:

```
auto lo ethX
iface lo inet loopback
iface ethX inet static
address xxx.xxx.xxx.xxx(enter IP)
netmask xxx.xxx.xxx.xxx
gateway xxx.xxx.xxx.xxx(enter gateway IP here)
```

Ensure to save any changes in the text editor.

Restart the network interface by the new settings:

```
$sudo /etc/init.d/networking restart
```

B.3. Set Bridge IP Address for NOX Controller

Install the bridge utilities by the following command in terminal:

```
$sudo apt-get install bridge-utils
```

Edit the network interface file by entering the following command into the terminal:

```
$sudo gedit /etc/network/interfaces
```

When the text editor opens, enter the bridge IP address according to the topology as follows:

```
auto lo
iface lo inet loopback
# Bridge between eth0 and eth1
iface br0 inet static
address 192.168.100.1
netmask xxx.xxx.xxx.xxx
gateway xxx.xxx.xxx.xxx(enter gateway IP here)
pre-up ip link set eth0 down
pre-up ip link set eth1 down
pre-up brctl add-br br0
pre-up brctl add-if br0 eth0 eth1
pre-up ip addr flush dev eth0
pre-up ip addr flush dev eth1
```

Ensure to save any changes in the text editor.

Restart the network interface by the new settings:

```
$sudo /etc/init.d/networking restart
```

B.4. NOX Controller Setup

Prerequisite packages depend on the Linux distribution on the controller PC. In the case of new instructions by Kostas Choumas, you have to install the following required dependencies:

```
cd /etc/apt/sources.list.d/
sudo wget http://openflowswitch.org/downloads/debian/nox.list
sudo apt-get update
sudo apt-get install nox-dependencies
sudo apt-get install libtbb-dev
sudo apt-get install libboost-serialization-dev libboost-all-dev
```

Then, change to the directory that the NOX's source code should be placed, and download the latest version from the git repository and switch to the downloaded folder:

```
git clone git://github.com/noxrepo/nox
cd nox
```

To compile and install NOX:

```
./boot.sh
mkdir build/
cd build/
../configure
make -j 5
make install
```

The ../configure command checks for the required dependencies, and 'make -j 5' should work on a system with 1 GB of RAM; 'make -j' may work on systems with more RAM.

Start NOX with the command:

```
cd src
./nox_core -v -i ptcp:6633 switch
```

The command will start the controller and establish a connection with the OpenFlow switch. The -v is for verbose. -h for both will give you the help files for the controllers. The ptcp:6633 is a passive connection to the default OpenFlow 6633 port (TCP), whereas *'switch'* is a core application of the NOX performing L2_forwarding functions. An active connection method using standard TCP exists but this does not work on this setup. In the OpenFlow reference system, a log of the verbose output can be saved by appending –log-file [filename].

B.5. Installing OpenVswitch on a Node

Install packages for OpenVswitch:

```
$sudo apt-get install pkg-config autoconf automake linux-libc-dev
```

Download and untar OpenVswitch (do not do anything yet):

```
cd ~/
wget http://openvswitch.org/releases/openvswitch-1.1.1.tar.gz
tar zxvf openvswitch-1.1.1.tar.gz
```

Installation:

If using Debian and its variants, OpenVswitch must be built as a kernel module. If everything is sound, installing OpenVswitch should be little more than following the steps in INSTALL.Linux found under the untarred OpenVswitch root directory.

```
cd ~/openvswitch-1.1.1
./boot.sh
./configure --with-l26=/lib/modules/`uname -r`/build
make
make install
```

Then, instantiate the kernel module:

```
$ su -
# /sbin/insmod ./datapath/linux-2.6/openvswitch_mod.ko
# /sbin/insmod ./datapath/linux-2.6/brcompat_mod.ko
```

Create the database for each OpenVswitch:

```
# mkdir -p /usr/local/etc/openvswitch
# ovsdb-tool create /usr/local/etc/openvswitch/conf.db vswitchd/
  vswitch.ovsschema
```

Start the OVS database server:

```
# ovsdb-server /usr/local/etc/openvswitch/conf.db\
--remote=punix:/usr/local/var/run/openvswitch/db.sock\
--remote=db:Open_vSwitch,manager_options\
--private-key=db:SSL,private_key\
--certificate=db:SSL,certificate\
--bootstrap-ca-cert=db:SSL,ca_cert --pidfile --detach
```

Database sanity check:

```
# ps -ef | grep ovsdb-server
```

Initialise database for the first time:

```
# ovs-vsctl --no-wait init
```

Start database:

```
# ovs-vswitchd unix:/usr/local/var/run/openvswitch/db.sock --pidfile
   --detach
```

Create OpenFlow switch data path:

```
 # ./utilities/ovs-dpctl add dp dp0
```

Then, add the interfaces to the data path:

Interfaces names for OpenVswitch 1 (eth4, eth6)

```
# ./utilities/ovs-dpctl add-if dp0 eth4
# ./utilities/ovs-dpctl add-if dp0 eth6
```

Interfaces names for OpenVswitch 2 (eth8, eth9)

```
# ./utilities/ovs-dpctl add-if dp0 eth8
# ./utilities/ovs-dpctl add-if dp0 eth9
```

Configure the OpenFlow Switch to communicate with the controller.

Then let OpenFlow protocol module talk to the controller. Let's assume we'll use datapath-id 0x004E46324304 for this OpenFlow switch and the controller is running on 192.168.0.100 port 6633.

```
# ./utilities/ovs-openflowd dp0 --datapath-id=0000004E46324304 tcp
  :192.168.0.100
  port 6633 --out-of-band
```

B.6. Installing OpenFlow Switching Reference System

Download software and required packages.

If using git:

```
sudo apt-get install git-core automake m4 pkg-config libtool
git clone http://openflowswitch.org/openflow
cd openflow
./boot.sh
```

If installing from tarball as root user:

```
mkdir openflow
cd openflow
wget http://openflowswitch.org/downloads/openflow-1.0.0.tar.gz
tar xzf openflow-1.0.0.tar.gz
cd openflow-1.0.0
sudo apt-get install gcc linux-headers-`uname -r`
```

Make and install:

```
./configure --with-l26=/lib/modules/`uname -r`/build
make
make install
```

B.7. NOX Controller Graphical User Interface (GUI)

NOX can provide a front-end GUI for supporting network visualization and monitoring. It also serves as an interface for the user and developer to communicate with NOX in run time. NOX GUI can be extended with custom views order to visualize custom characteristics for research or demo purposes, or with custom input in order to trigger NOX component functions.

This GUI is developed in Python and uses the QT library. It communicates with NOX by way of JSON messages exchanged with the messenger component (simple json should be already installed if you are running NOX).

Install required dependencies for Ubuntu:

```
$ sudo apt-get install pyqt4-dev-tools python-qt4 python-simplejson
  python-qt4-sq
```

Start the NOX controller and follow by minimum 'monitoring' and 'trackhost_pktin' argument.

```
~/nox/build/src# sudo ./nox_core -v -v -i ptcp:6633 monitoring
  trackhost_pktin
```

Running the GUI:

```
/nox/src# ./nox-gui.py
```

B.8. Installing OpenFlow Wireshark Dissector

The OpenFlow reference system comes with an OpenFlow Wireshark dissector. Users must be root in order to use Wireshark, and thus must be installed on a node.

To Install Wireshark and prerequisites, glib is required for the plug-in to work:

```
$sudo apt-get install wireshark libgtk2.0-dev
```

Then locate the patch under the utilities directory:

```
cd utilities/wireshark_dissectors/openflow
make
sudo make install
```

Usage:

In order use Wireshark, X11 tunnelling needs to be enabled. To do this, add the *-X* option when using SSH to access the console and the nodes.

Also, tcpdump pcap files can be opened using Wireshark. Once the plug-in has been installed, OpenFlow packets can be interpreted from these files as well (see Figure B.2).

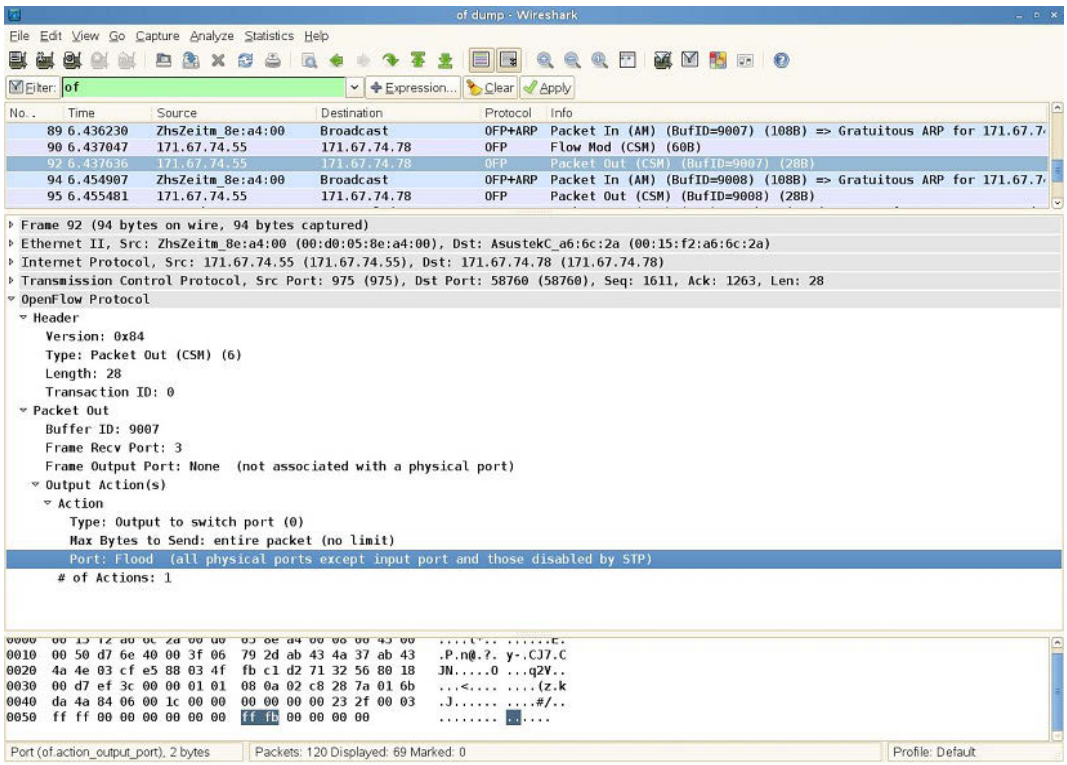


Figure B.2.: OpenFlow Dissector in Wireshark

C. OpenFlow Laboratory with Mininet

C.1. Setting up Mininet Environment

This OpenFlow laboratory consists of four key components: (1) a virtualisation software using VirtualBox; (2) a terminal program supporting SSH (PuTTY); (3) an X server for X11 forwarding (e.g. Xming); (4) the Mininet VM image. The setup in this chapter is based on Mininet version 2.0 using a pre-packaged VM image that runs over Ubuntu. The latest version can be downloaded at **www.mininet.org/-download**. This VM image comes with all OpenFlow binaries, pre-installed tools to support larger emulated networks, and Mininet itself. VirtualBox (free, GPL) is the virtualisation program used to import and run the Mininet VM image (800MB - 1GB compressed). The Mininet image file can be imported to VirtualBox by double-clicking on the VM image (.ovf) or go to **File** and select **Import Appliance**, then go to **Settings** and add an additional *host-only network adapter* to log in to the VM image. The VM should be configured with two interfaces. For example, the NAT interface to access the Internet having a 10.x IP address, and host-only interface to enable the communication with the host machine could be eth1 having 192.168.x IP address.

Follow these steps for running Mininet from VM:

- Download the Mininet VM image.
- Download and install a virtualisation system. It is recommended to use VirtualBox (free, GPL) because it is free and works on OS X, Windows, and Linux.
- Import and start the VM in VirtualBox.
- Log in to Mininet VM, using the following default name and password:

```
1 mininet-vm login: mininet
2 Password: mininet
```

The root account is not enabled for login and hence must use sudo to run a command with superuser privileges.

- Establish an SSH connection into VM after having set up a host-only network on eth1:

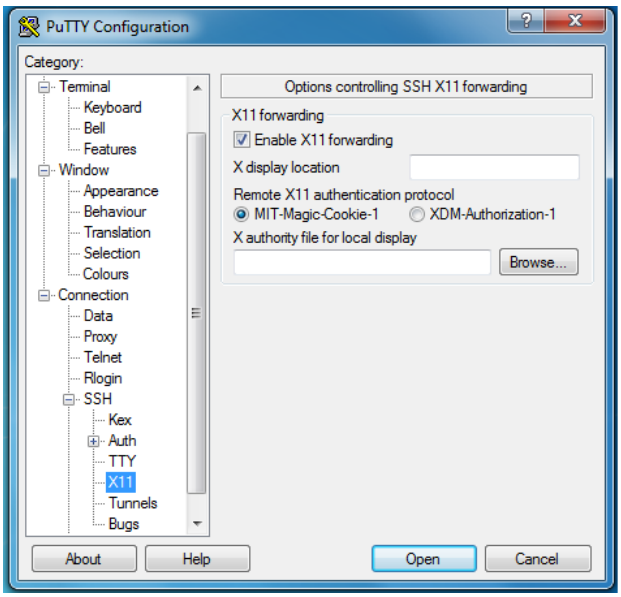


Figure C.1.: Enabling X11 Forwarding in PuTTY

```
sudo dhclient eth1    # make sure that eth1 has an IP address
ifconfig eth1         # note the IP address of eth1 probably 192.168.
  x.x
```

- Next, use the PuTTY SSH client to log into the Mininet VM:

```
$ ssh -X mininet@[eth1's IP address]
```

Under Windows, the X-server (Xming server) must be running in order to use the X11 applications such as xterm and Wireshark. Start the Xming and make an SSH connection with X11 forwarding option enabled. If using PuTTY (SSH terminal), connect to Mininet by entering the VM's IP address (eth1) and enabling X11 forwarding. Use the PuTTY's GUI to enable X11 forwarding by selecting **PuTTYConnection | SSH | X11**, then click on **Enable X11 forwarding** (see Figure C.1).

- Figure C.2 shows a screen shot of the experimental environment based on VirtualBox, Mininet, Xming (X-server), and PuTTY (SSH client). Mininet VM has been logged into by using PuTTY SSH terminal. Since the X11 forwarding is also enabled, the Wireshark GUI will appear as a separate window and running in a background process. Start the Wireshark to capture OpenFlow packets by entering the following command in PuTTY terminal:

```
mininet@mininet-vm:~$ sudo wireshark &
```

- Before running Mininet, it is important to select the **Capture** device in Wireshark as 'lo' or loopback interface and start capturing the packets. Enter

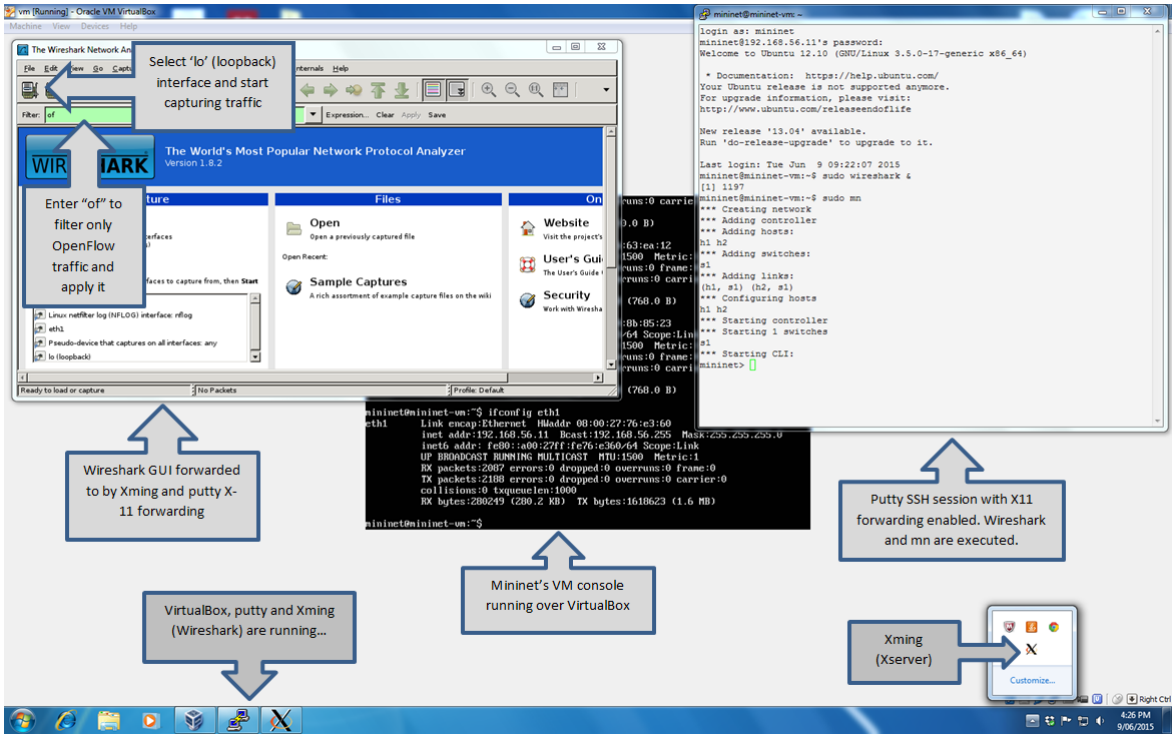


Figure C.2.: OpenFlow Laboratory Using Mininet

"of" (OpenFlow) in the filtering box of Wireshark in order to display only the OpenFlow related packets, and apply it to the capturing traffic. Because the Mininet emulator has not started yet, no OpenFlow packets should be shown in the main window of Wireshark. Next subsection will demonstrate a sample experiment using Mininet.

C.2. Experimenting with Mininet

This sample experiment uses the default topology of Mininet (by running $sudo mn). This default topology consists of two hosts connected to one OpenFlow switch, and plus the reference OpenFlow controller. In order to display the links, nodes, and dump information regarding all nodes in the setup, run the following commands respectively:

```
1 mininet> net
2 mininet> nodes
3 mininet> dump
```

If the first string entered into the Mininet CLI (mininet>) is a host, switch or controller name, the command is executed on that node. For example, run a command on a host process to see the Ethernet and loopback interface of Host 1 (h1):

```
mininet> h1 ifconfig -a
```

To test the connectivity of each host and verify ping from h1 to h2, run a simple ping command:

```
mininet> h1 ping -c 1 h2
```

Use the Wireshark capturing window to see OpenFlow control traffic. Host 1 (h1) sends ARPs for the MAC address of Host 2 (h2), which causes a packet_in message to the OpenFlow controller. Then, the controller sends a packet_out message to broadcast the packet and flood it to other ports on the switch, for this example the only other data port. Host 2 sees the ARP request and sends a broadcast reply. This reply goes to the controller, which sends it to Host 1 and also installs a flow entry in s1 (OpenFlow switch) flow table.

Host 1 can now send its ping through an ICMP Echo Request because it already knows the MAC address of Host 2. This request and its corresponding reply from Host 2 both go to the controller with an action to push down a flow entry. The actual packets are getting sent out a specific port too. If the same ping is repeated, the ping time from the second ping command should be a much lower ping time (<100 μs). This is because a flow entry containing ICMP ping traffic was installed previously in the switch. Hence, no control traffic was generated, and the packets immediately pass through the switch. Using the Mininet CLI built-in *pingall* command is the easiest way to run this test, which does an all-pairs ping.

Another useful test of Mininet is a self-contained regression test, which do not need to drop into the CLI.

```
$ sudo mn --test pingpair
```

This command creates a minimal Mininet topology, starting up the OpenFlow reference controller, run an all-pairs ping test, and tore down both the topology and the controller.

iperf is another useful test for performance evaluation.

```
$ sudo mn --test iperf
```

This command may take a few seconds to complete. It creates the same Mininet topology that runs an Iperf server on one host, an Iperf client on the second host, and reports the bandwidth achieved between these two hosts.

Mininet also permits to set link parameters, which can be set automatically from the following command line:

```
$ sudo mn --link tc,bw=10,delay=10ms
mininet> iperf
...
mininet> h1 ping -c10 h2
```

The above sample command sets the delay for each link to 10 ms. The round trip time (RTT) should be around 40 ms because the ICMP request traverses two links (one to the destination, one to the switch), and the ICMP reply traverses two links coming back.

Mininet uses a simple Python API to define custom topologies for experiments, and an example is provided in ~mininet/custom/topo-2sw-2host.py. This example connects two switches directly with one host connected to each switch:

```
"""Custom topology example
Two directly connected switches plus a host for each switch:
    host --- switch --- switch --- host
   h1 <-> s3 <-> s4 <-> h2
Adding the 'topos' dict with a key/value pair to generate our newly
  defined
topology enables one to pass in '--topo=mytopo' from the command
  line.
"""
from mininet.topo import Topo
class MyTopo( Topo ):
    "Simple topology example."
    def __init__( self ):
        "Create custom topo."
        # Initialize topology
         Topo.__init__( self )
        # Add hosts and switches
        leftHost = self.addHost( 'h1' )
        rightHost = self.addHost( 'h2' )
        leftSwitch = self.addSwitch( 's3' )
        rightSwitch = self.addSwitch( 's4' )
        # Add links
        self.addLink( leftHost, leftSwitch )
        self.addLink( leftSwitch, rightSwitch )
        self.addLink( rightSwitch, rightHost )
topos = { 'mytopo': ( lambda: MyTopo() ) }
```

This custom Python script can be passed as a command line parameter to Mininet. When a custom Mininet file is provided, it can add new topologies, switch types, and tests to the command-line. For example:

```
$ sudo mn --custom ~/mininet/custom/topo-2sw-2host.py --topo mytopo
  --test pingall
```

Moreover, there is a complex way of debugging and accessing to the console of hosts, switches, and controllers by starting Mininet with *-x* in the command line parameter (sudo mn -x). The command will allow xterm to pop up for running useful interactive commands. For example, in the xterm of switch: s1 (root) can run the following command to dump the flow table of s1:

```
# dpctl dump-flows tcp:127.0.0.1:6634
```

Because the flow table of the switch s1 is empty, nothing will print out. However, after running a normal ping command (#ping 10.0.0.2) from the host (h1) and return back to xterm of switch s1 and dump the flow table, there should be multiple flow entries installed in the table. For example:

```
root@mininet-vm:~# dpctl dump-flows tcp:127.0.0.1:6634
stats_reply (xid=0x525af393): flags=none type=1(flow) cookie=0,
  duration_sec=5s,duration_nsec=19000000s, table_id=0,priority=65535,
  n_packets=1,n_bytes=98,idle_timeout=60,hard_timeout=0,icmp,in_port
  =2,dl_vlan=0xffff,dl_vlan_pcp=0x00,dl_src=ce:a9:b7:c9:54:d5,dl_dst
  =12:b7:d3:00:f8:c9,nw_src=10.0.0.2,nw_dst=10.0.0.1,nw_tos=0x00,
  icmp_type=0,icmp_code=0,actions=output:1
cookie=0,duration_sec=5s,duration_nsec=20000000s,table_id=0,priority
  =65535,n_packets=1,n_bytes=98,idle_timeout=60,hard_timeout=0,icmp,
  in_port=1,dl_vlan=0xffff,dl_vlan_pcp=0x00,dl_src=12:b7:d3:00:f8:c9,
  dl_dst=ce:a9:b7:c9:54:d5,nw_src=10.0.0.1,nw_dst=10.0.0.2,nw_tos=0
  x00,icmp_type=8,icmp_code=0,actions=output:2
```

It is also useful to use the dpctl built-in command in Mininet to manually install the necessary flows. For example in your SSH terminal:

```
$ dpctl add-flow tcp:127.0.0.1:6634 in_port=1,actions=output:2
$ dpctl add-flow tcp:127.0.0.1:6634 in_port=2,actions=output:1
```

It will direct packets coming at port 1 to port 2 and the other way around. This can be verified by dumping the flow table.

Mininet runs OpenVswitch in OpenFlow mode and requires an OpenFlow controller by default. Mininet has several built-in controller classes including the OpenFlow reference controller, OpenVswitch's ovs-controller, and the NOX-Classic controller (deprecated). Users can simply choose which controller they want from the following command:

```
$ sudo mn --controller ref
$ sudo mn --controller ovsc
$ sudo mn --controller NOX,pyswitch
```

All of these examples uses a controller to turn the OpenVswitch into an Ethernet learning switch.

It can be helpful to bring links up and down for fault tolerance testing. To disable both halves of a virtual Ethernet pair:

```
mininet> link s1 h1 down
```

There should be an OpenFlow Port Status change notification gets generated. To bring the link back up:

```
mininet> link s1 h1 up
```

To display an xterm for h1 and h2:

```
mininet> xterm h1 h2
```

C.3. Running External Controllers

When an OpenFlow environment is started in Mininet, each switch can be connected to a remote controller. The remote controller could be in the Mininet VM, on the local machine, or running outside of the VM such as on the VM host or a different physical PC. This setup may be more convenient if a custom version of a controller framework and development tools like Eclipse are already installed on the local machine, or user needs to test a controller running on another physical machine. In order to run this, ensure that the controller is reachable from Mininet VM and enter in the host IP and the listening port (optionally):

```
$ sudo mn --controller=remote,ip=[controller IP],port=[controller
  listening port]
```

For example, the following command starts up a simple learning switch using a POX controller framework:

```
$ cd ~/pox
$ ./pox.py forwarding.l2_learning
```

In one terminal, and in another terminal, start up Mininet to connect to the remote controller, which is actually running locally but outside of Mininet's control:

```
$ sudo mn --controller=remote,ip=127.0.0.1,port=6633
```

Note that these are actually the default IP address and port values. If there are some traffic generated (mininet> h1 ping h2), there should be some output in the POX window showing that the switch has connected and that some flow entries have been installed in the flow table.

After a while, the OpenVswitch in Mininet will get connected to the POX controller. The output from this controller should look like the following:

```
POX 0.0.0 / Copyright 2011 James McCauley
DEBUG:core:POX 0.0.0 going up...
DEBUG:core:Running on CPython (2.7.3/Sep 26 2012 21:51:14)
INFO:core:POX 0.0.0 is up.
This program comes with ABSOLUTELY NO WARRANTY. This program is free
   software, and you are welcome to redistribute it under certain
  conditions.
Type 'help(pox.license)' for details.
DEBUG:openflow.of_01:Listening for connections on 0.0.0.0:6633
INFO:openflow.of_01:[Con 1/1] Connected to 00-00-00-00-00-01
DEBUG:forwarding.l2_learning:Connection [Con 1/1]
```

```
10 Ready.
11 POX>
```

The debug messages in the POX terminal show that the OpenVswitch is connected to POX controller and behaves as an L2 learning switch.

There are several OpenFlow controller frameworks readily available and should work readily with Mininet as long as they are started up and specified in the remote controller option with the correct IP address of the machine, where the controller is running and the correct port that it is listening on.

D. OpenFlow Setup in OMNeT++ INET Framework

The OMNeT++ INET Framework 2.0 network simulator is a C++ discrete event simulator. The main advantage of this simulator is that it simplifies the integration of new modules, and allows existing modules to be customised. The INET Framework is a network simulation package that contains models for wired and wireless networking protocols including UDP, TCP, SCTP, IP, IPv6, and Ethernet. The INET Framework has recently implemented an extension to enable OpenFlow to be modelled. The OpenFlow extension is still in early development and is currently based on OpenFlow switch specification version 1.2.

D.1. Installing OMNeT++ 4.2

Based on Windows versions, the supported Windows versions are the Intel 32-bit versions of Windows XP and later versions such as Windows 7 and 8.

Note that 64-bit Windows versions are also supported, but be aware that binaries bundled with OMNeT++ are 32-bit ones, and simulations will also be compiled in 32-bit mode.

Download the OMNeT++ source code from **http://omnetpp.org.** Make sure you select the Windows-specific archive, named omnetpp-4.2-src-windows.zip.

This package is mostly self-contained: in addition to OMNeT++ files it includes a C++ compiler, a command-line build environment, and all libraries and programs required by OMNeT++.

Copy the OMNeT++ archive to the directory where you want to install it. Choose a directory whose full path **does not contain any space**; for example, do not put OMNeT++ under *Program Files*.

Extract the zip file. To do so, right-click the zip file in Windows Explorer, and select Extract All from the menu. You can also use external programs like Winzip or 7zip. Rename the resulting directory to omnetpp-4.2.

When you look into the new omnetpp-4.2 directory, should see directories named doc, images, include, tools, etc., and files named mingwenv.cmd, configure, Makefile, and others.

D.2. Configuring and Building OMNeT++

Start mingwenv.cmd in the omnetpp-4.2 directory by double-clicking it in Windows Explorer. It will bring up a console with the MSYS bash shell, where the path is already set to include the omnetpp-4.2/bin directory.

If you want to start simulations from outside the shell as well (for example from Explorer), you need to add OMNeT++'s bin directory to the path; instructions are provided later.

First, check the contents of the configure.user file to make sure it contains the settings you need. In most cases, you do not need to change anything.

notepad configure.user

Then enter the following commands:

```
$ ./configure
$ make
```

The build process will create both debug and release binaries.

D.3. Verifying the Installation

You should now test all samples and check they run correctly. As an example, the dyna example is started by entering the following commands:

```
$ cd samples/dyna
$ ./dyna
```

By default, the samples will run using the graphical Tkenv environment. You should see GUI windows and dialogs.

D.4. Starting the IDE

OMNeT++ comes with an Eclipse-based Simulation IDE. You should be able to start the IDE by typing:

```
$ omnetpp
```

We recommend that you create a shortcut for starting the IDE. To do so, locate the omnetpp.exe program in the omnetpp-4.2/ide directory in Windows Explorer, right-click it and choose *Send To > Desktop (create shortcut)* from the menu. On Windows 7, you can right-click the taskbar icon while the IDE is running, and select *Pin this program to taskbar* from the context menu.

When you try to build a project in the IDE, you may get the following warning message:

> Toolchain "..." is not supported on this platform or installation. Please go to the Project menu, and activate a different build configuration. (You may need to switch to the C/C++ perspective first so that the required menu items appear in the Project menu.)

If you encounter this message, choose *Project > Properties > C/C++ Build > Tool Chain Editor > Current toolchain > GCC for OMNeT++.*

D.5. Installing INET Framework 2.0

This is a manual installation of the downloaded INET version 2.0 or assume that the INET git repository is cloned to the local file system. The only tricky part is how to import the project into the workspace.

1. Download the INET version 2.0 sources

2. Unzip it into the directory of your preference, for example, tar xvfz inet-2.0.tgz

3. Run the OMNeT++ IDE, and import the project through *File -> Import -> Existing Projects* to the Workspace. A project named "INET" should appear.

4. Build with *Project -> Build*, or hit *Ctrl+B*

5. Now you should be able to launch example simulations.

Please also read the INSTALLATION file in the INET sources, as it may contain additional information.

> INET 2.0 can be compiled with the latest OMNeT++ version; you just need to replace getFieldArraySize -> getArraySize in two files where the compiler complains. If you are not using MANET routing protocols it does not matter. In fact, if you turn off the MANET routing in the Project Features dialog, you even do not have to modify the code in those two files.

D.6. Installing OpenFlow Extension for the OMNeT++

This is the implementation of the OpenFlow system for INET-2.0 and OMNeT++ 4.2 based on the OpenFlow switch specification 1.0.

1. Download the OMNeT++ source code from the URL below or clone the ofomnet git repository to the local file system.

https://github.com/lsinfo3/ofomnet/archive/master.zip

2. Unzip it into the directory of your choice; tar xvfz ofomnet-master.tgz

3. Run the OMNeT++ IDE, and import the project via *File -> Import -> Existing Projects* to the Workspace. A project named "openflow" should appear.

4. Make INET project as a reference project in the workspace by right click at openflow -> *Properties -> Project References* and select INET project.

5. Build with *Project -> Build*, or hit *Ctrl+B*

6. Now you should be able to launch OpenFlow example simulations.

Required changes to the INET Framework 2.0

The following changes are required for FlatNetworkConfigurator to ignore the data plane interfaces for OpenFlow switches.

The EtherMACBase::registerInterface method in src/linklayer/ethernet/EtherMACBase is changed so that the data plane interfaces of OpenFlow switches are not added to the InterfaceTable module, and thus are ignored by the FlatNetworkConfigurator.

Modify file src/linklayer/ethernet/EtherMACBase and Change method EtherMACBase::registerInterface

FROM

```
if (ift)
    ift->addInterface(interfaceEntry);
```

TO

```
if (ift && par("doRegisterAtIft").boolValue())
    ift->addInterface(interfaceEntry);
```

ADD new parameter

```
bool doRegisterAtIft = default(true);
```

In the following modules:

- EtherMAC.ned
- EtherMACFullDuplex.ned

D.7. Example of Simple OpenFlow testing in OMNeT++

Figure D.1 demonstrates a simple OpenFlow network topology in OMNeT++. The OMNeT++ tool can be used to create different network scenarios, topologies, behaviours, and environments or other application fields for different purposes. Using

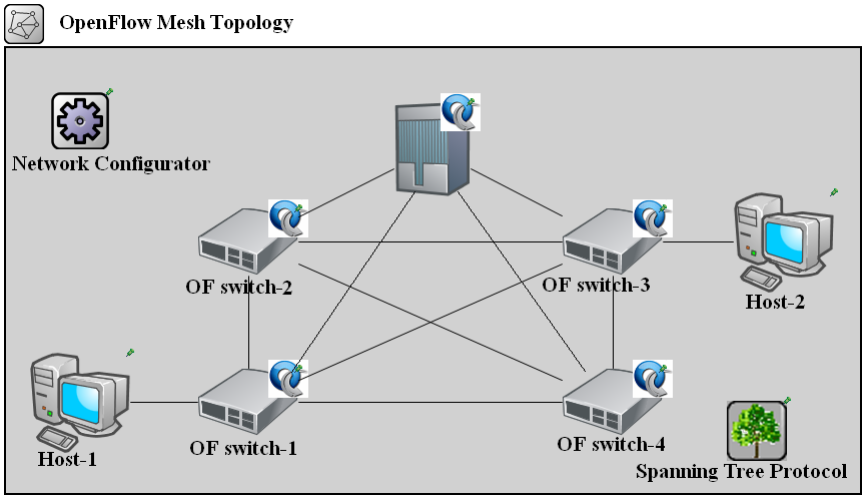


Figure D.1.: OpenFlow Mesh Topology with Spanning Tree Protocol

OMNeT++, it is easy to simulate geographic distance and help predict how that would affect the behaviours of this new infrastructure when considering different technologies or products running on different software.

The corresponding Network Description (NED) source file would look like this:

```
package openflow.scenarios;
import inet.nodes.ethernet.EtherLink;
import inet.nodes.ethernet.EtherSwitch;
import inet.nodes.inet.Router;
import inet.nodes.inet.StandardHost;
import inet.networklayer.autorouting.ipv4.FlatNetworkConfigurator;
import inet.util.ThruputMeter;
import inet.util.ThruputMeteringChannel;
import openflow.nodes.*;
import openflow.utility.SpanningTree;
network OpenFlow Mesh Topology
{
    types:
        channel ethernetline extends ThruputMeteringChannel
        {
            delay = 5ms;
            datarate = 100Mbps;
            thruputDisplayFormat = "ms";
        }
    submodules:
        controller: Open_Flow_Controller {}
        open_Flow_Switch1: Open_Flow_Switch {}
        open_Flow_Switch2: Open_Flow_Switch {}
        open_Flow_Switch3: Open_Flow_Switch {}
        open_Flow_Switch4: Open_Flow_Switch {}
        Host1: StandardHost {}
        Host2: StandardHost {}
        flatNetworkConfigurator: FlatNetworkConfigurator {}
```

```
        spanningTree: SpanningTree {}
    connections allowunconnected:
        Host1.ethg++ <--> ethernetline <--> open_Flow_Switch1.ethg++;
        open_Flow_Switch1.ethg++ <--> ethernetline <--> open_Flow_Switch2.ethg++;
        open_Flow_Switch2.ethg++ <--> ethernetline <--> open_Flow_Switch3.ethg++;
        open_Flow_Switch3.ethg++ <--> ethernetline <--> open_Flow_Switch4.ethg++;
        open_Flow_Switch4.ethg++ <--> ethernetline <--> open_Flow_Switch1.ethg++;
        open_Flow_Switch4.ethg++ <--> ethernetline <--> open_Flow_Switch2.ethg++;
        open_Flow_Switch2.gate_controller++ <--> ethernetline <--> controller.ethg++;
        open_Flow_Switch3.gate_controller++ <--> ethernetline <--> controller.ethg++;
        open_Flow_Switch1.gate_controller++ <--> ethernetline <--> controller.ethg++;
        open_Flow_Switch4.gate_controller++ <--> ethernetline <--> controller.ethg++;
        open_Flow_Switch1.ethg++ <--> ethernetline <--> open_Flow_Switch3.ethg++;
        open_Flow_Switch3.ethg++ <--> ethernetline <--> Host2.ethg++;
}
```

When a program is started, it first reads all NED files containing the model topology, and then it reads a configuration file (usually called omnetpp.ini). The example of a Ping application initialisation file (ini) can be seen below:

```
[General]
network = OpenFlow Mesh Topology
output-vector-file = ${resultdir}/${inifile}/${inifile}-${runnumber}.vec
output-scalar-file = ${resultdir}/${inifile}/${inifile}-${runnumber}.sca
sim-time-limit = 300s
debug-on-errors = false

#openflow
**.controller.ofa_controller.port = 6633
**.open_Flow_Switch*.sendCompletePacket = false
**.controller.behavior = "Forwarding"
**.ofa_switch.connectPort = 6633
**.ofa_switch.connectAddress = "controller"
**.buffer.capacity = 10
**.ofa_switch.flow_timeout = 5s
**.open_Flow_Switch*.etherMAC[*].promiscuous = true
**.open_Flow_Switch*.serviceTime = 0.0000098s
**.controller.serviceTime = 0.00024s
```

```
19
20 # NIC configuration
21 **.ppp[*].queueType = "DropTailQueue" # in routers
22 **.ppp[*].queue.frameCapacity = 10  # in routers
23 **.configurator.networkAddress = "192.168.1.0"
24 **.open_Flow_Switch*.sendCompletePacket = false
25 **.open_Flow_Switch*.tcp.mss = 800
26 **.controller.tcp.mss = 800
27
28 [Config one]
29 description = "RTT Ping"
30 #ping App
31 **.Host1.numPingApps = 1
32 **.Host1.pingApp[*].typename = "PingApp"
33 **.Host1.pingApp[*].count = 370
34 **.Host1.pingApp[*].destAddr = "Host2"
35 **.Host1.pingApp[*].packetSize = 56B
36 **.Host1.pingApp[*].sendInterval = 1s
37 **.Host1.pingApp[*].startTime = default
38 **.Host1.pcapRecorder[*].pcapFile = "Host1Log.pcap"
39 **.Host2.pcapRecorder[*].pcapFile = "Host1Log.pcap"
40 **.rtt.result-recording-modes = "stats,vector"
41 **.pingApp[*].rtt*.vector-recording = true
42 **.vector-recording = false
43 **.pingApp[*].rtt*.scalar-recording = true
44 **.scalar-recording = false
```

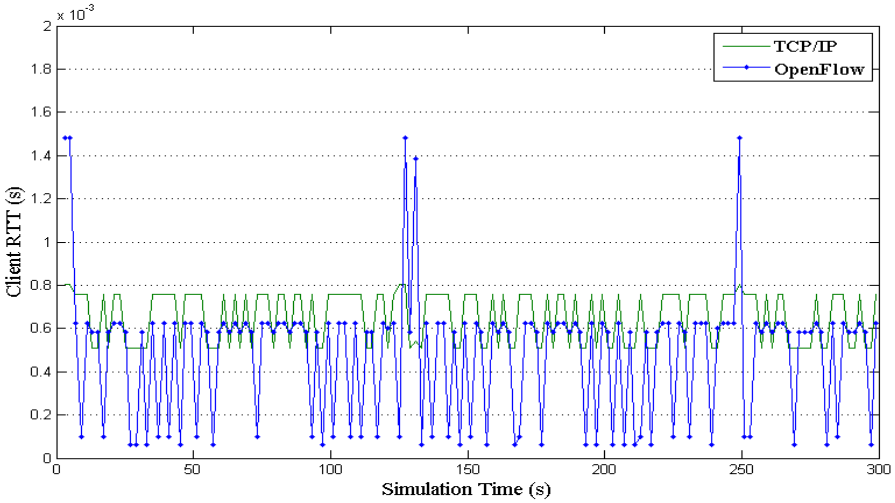


Figure D.2.: Measured RTT of TCP/IP vs. OpenFlow

This study compares the performance of TCP/IP with the newly emerging OpenFlow standard for SDN. The performance comparison simulates OpenFlow and TCP/IP networks using the OMNeT++ INET Framework discrete events network

simulator. By analysing the key network metric of round-trip-time (RTT), the results indicate that OpenFlow performed slightly better than TCP/IP in this analysis.

Figure D.2 shows RTT for OpenFlow and TCP/IP, where each client has generated the pings and logged the results. It can be clearly seen that TCP/IP has longer round trip times than OpenFlow. The results indicate that OpenFlow outperforms TCP/IP and suggests that OpenFlow's performance benefits from the packet transmission methods operating at Layer 2, 3, and 4. It is also evident that OpenFlow produces several sudden spikes that are caused by the expired Idle-timeout (120 seconds). Hence, the packet will be forwarded to the controller for computing a path again.

E. Hardware for OpenFlow Test Lab

A small Linux based OpenFlow network has already been set up physically in the lab with the following hardware by the author:

- Raspberry Pi 2 Model B

  CPU: 900 MHz quad-core ARM Cortex A7 (ARMv7 instruction set)

  SoC: Broadcom® BCM2836 (CPU, GPU, DSP, SDRAM, and single USB port)

  GPU: Broadcom VideoCore IV @ 250 MHz, OpenGL ES 2.0 (24 GFLOPS), 1080p30 MPEG-2 and VC-1 decoder (with license), 1080p30 h.264/MPEG-4 AVC high-profile decoder and encoder

  Memory: 1 GB (shared with GPU)

- Dell OptiPlex GX620

  CPU Intel Pentium 4 531 / 3 GHz

  Memory: 4 GB

- Ethernet cable Category 5e
- USB 2.0 to Fast Ethernet Adapter Model No:KY-RD9700
- SanDisk - SDSDQ-008G-A11M - 8GB Memory card Type: microSDHC

Figure E.1 shows a logical setup of the OpenFlow test lab consisting of two software switches controlled by DAIM and two hosts. Ubuntu 14.10 image is used as the operating system for the Raspberry Pi's, which comes with a lightweight LXDE desktop pre-installed. Each Raspberry Pi runs an OpenVswitch implementation and the DAIM controller on the same device, whereas the hosts are normal PCs. The OpenFlow software reference system is also installed to add OpenFlow switching capability to the Raspberry Pi with multiple NICs. This physical network is already running as a test-bed for the future trend work in this dissertation.

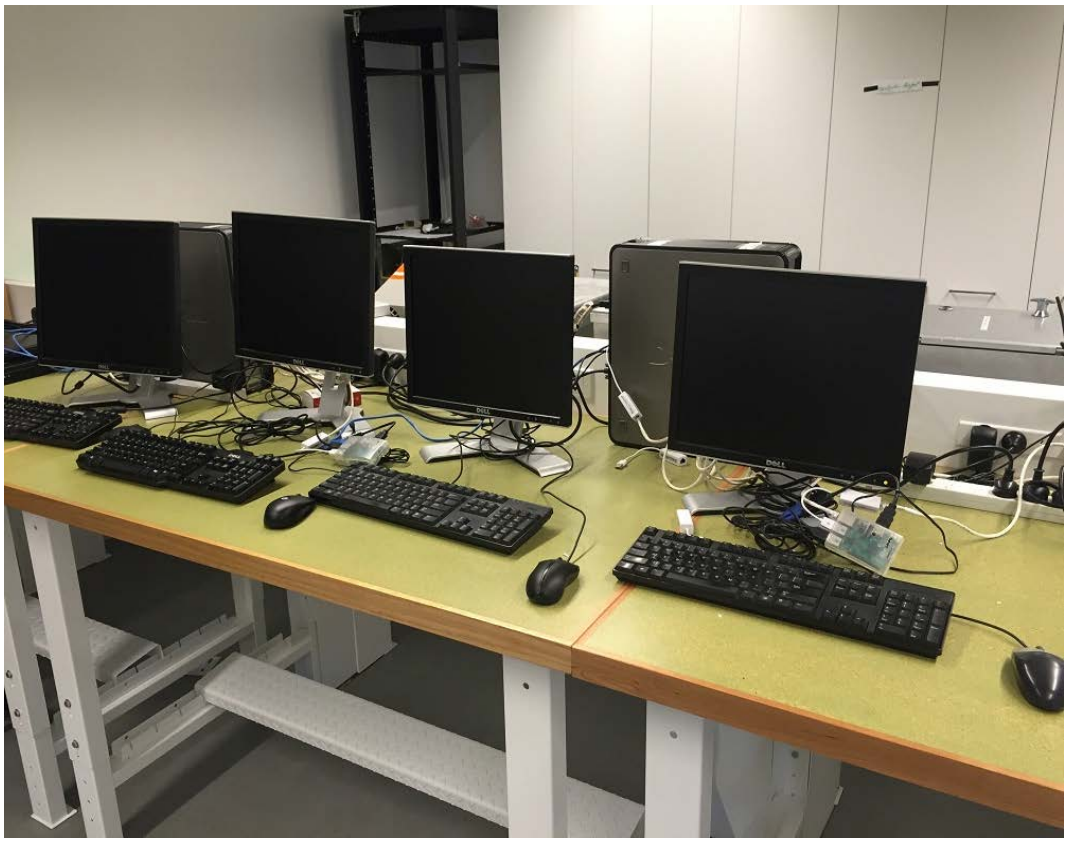


Figure E.1.: A Small-Size OpenFlow Network

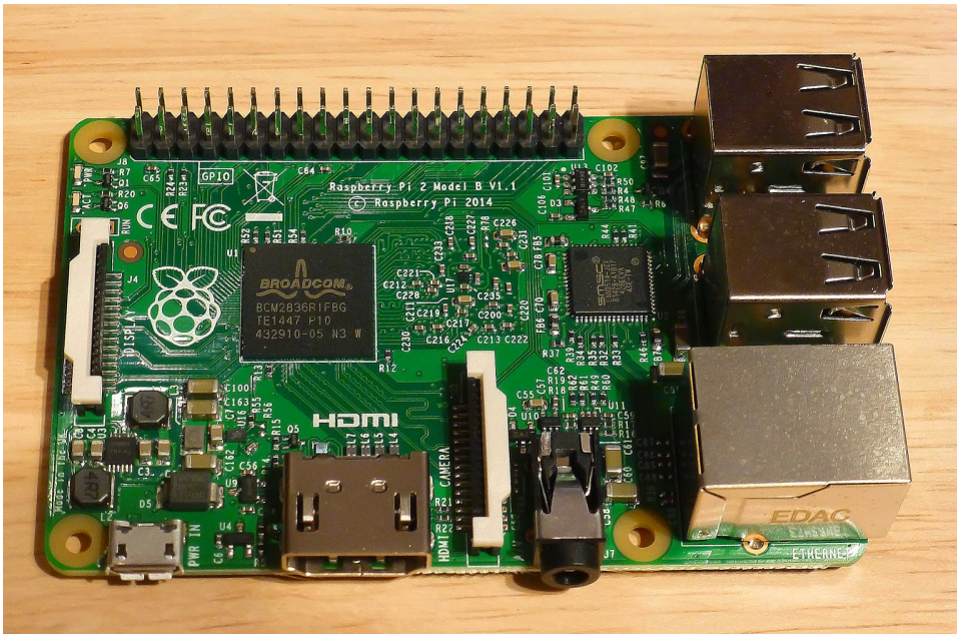


Figure E.2.: Raspberry Pi 2 Model B 1GB