

VICTORIA UNIVERSITY OF WELLINGTON

*Te Whare Wananga o te Upoko o te Ika a Maui*



School of Engineering and Computer Science

*Te Kura Mātai Pūkaha, Pūrorohiko*

PO Box 600  
Wellington  
New Zealand

Tel: +64 4 463 5341  
Fax: +64 4 463 5045  
Internet: [office@ecs.vuw.ac.nz](mailto:office@ecs.vuw.ac.nz)

## **Traffic Classification in Enterprise Networks with the Era of IoT**

### **Final Report**

Matt Hayes

Supervisor(s): Prof Winston Seah and Dr Bryan Ng

Submitted in partial fulfilment of the requirements for COMP489

#### **Abstract**

The emergence of the Internet of Things (IoT) combined with disparate device count and link capacity variation across the enterprise is driving the need for improved traffic classification to address enterprise networking issues. Today, a detailed understanding of network traffic is required to configure traffic classification for uses such as Quality of Service (QoS) and security; however this becomes impractical as IoT vastly increases the number of different device types and flows on the network. A solution is required that allows simple definition of classification policies with automated configuration deployment, so that organisations can apply and update traffic classification efficiently and effectively. This report analyses traffic classification options for enterprise networks in the era of IoT, proposes an architecture that leverages the capabilities of Software-Defined Networking (SDN) and presents results from a prototype implementation.



# Acknowledgements

Thank you to Professor Seah and Dr Ng for their patient and insightful assistance with this project. Any mistakes or oversights are mine, not theirs.

This project is dedicated to the memory of my father, Gilbert Hayes (1946-2014). He inspired me through my childhood with practical applications for mathematics and science, and taught me to question all assumptions in life. He also bought me a personal computer<sup>1</sup> in an age when they were neither cheap nor common, fostering skills that have held me in good stead throughout my life.

---

<sup>1</sup> A Spectravideo SV-318, see: <http://en.wikipedia.org/wiki/SV-318>



# Contents

Introduction .....	1
1.1. Context.....	1
1.2. Software-Defined Networking .....	3
1.3. Report Structure.....	4
2. Problem Description and Analysis .....	5
2.1. Problem Statement.....	5
2.2. Requirements for Traffic Classification in the Enterprise.....	5
2.3. Categorising Traffic Classification Methods.....	6
2.4. Analysis .....	6
2.5. Possible Solutions.....	7
Payload Inspection .....	7
Statistical Classification .....	7
Multiclassifier.....	8
Role of SDN in Solutions.....	8
2.6. Hypothesis .....	8
2.7. Chapter Summary .....	9
3. Design.....	11
3.1. Architecture .....	11
OpenFlow .....	11
3.2. Introduction to the Prototype System .....	12
3.3. Design Principles .....	13
3.4. Traffic Classification Modules .....	14
Traffic Classification Policy Module .....	14
Static Classification Module .....	14
Identity Classification Module .....	15
Payload Classification Module.....	15
Statistical Classification Module.....	15
3.5. Nmeta Supplementary Features .....	18
Configuration .....	18
Data Management .....	18
REST API.....	19
3.6. Non-Functional Considerations .....	19

Performance Considerations .....	19
Security Considerations.....	19
Scalability Considerations.....	20
3.7. Chapter Summary .....	20
4. Evaluation.....	21
4.1. Evaluation Methods .....	21
4.2. Lab Environments.....	21
Virtual Lab Environment .....	21
Physical Lab Environment .....	23
Bandwidth Congestion .....	24
HTTP Response Time Measurements.....	24
4.3. Test Use Cases.....	24
4.4. Test Use Cases Static-1 and Static-2 .....	26
Goal .....	26
Method .....	26
Desired Outcome(s).....	27
Configuration .....	27
Results .....	28
Repeatability Test and Results .....	30
Analysis .....	31
Summary of Static Traffic Classification Findings.....	32
4.5. Test Use Cases Identity-1 and Identity-2 .....	32
Goal .....	32
Method .....	33
Desired Outcome(s).....	33
Configuration .....	33
Results .....	36
Analysis .....	37
Summary of Identity Traffic Classification Findings .....	37
4.6. Test Use Cases Payload-1 and Payload-2.....	37
Goal .....	37
Method .....	37
Desired Outcome(s).....	38
Configuration .....	38
Results .....	39
Analysis .....	40

Summary of Payload Traffic Classification Findings .....	41
4.7. Test Use Cases Statistical-1 and Statistical-2.....	41
Goal .....	41
Method .....	41
Desired Outcome(s).....	41
Configuration .....	41
Results .....	42
Analysis .....	43
Summary of Statistical Traffic Classification Findings .....	44
4.8. Evaluation of Hypothesis.....	44
4.9. Chapter Summary .....	44
5. Conclusion.....	46
5.1. Other Observations .....	46
Reticulation of the OpenFlow Protocol.....	46
Denial of Service Vulnerabilities .....	46
Non-Functional Requirements .....	46
Appendix A - Test Details.....	51
Installation .....	51
Server (PC1) Configuration .....	51
Client (PC3) Installation and Configuration .....	51
Set up QoS Queues on Switches .....	51
5.2. Running Tests .....	54
Test Use Case Static-1 and Static-2 .....	54
Test Use Cases Identity-1 and Identity-2 .....	54
Test Use Cases Payload-1 and Payload-2 .....	55
Test Use Case Statistical-1 .....	56
Appendix B - WAN3 Build Instructions.....	57
Pre-Requisites.....	57
VM1 – Server / Controller .....	57
Install VirtualBox Additions .....	59
Configure Networking.....	59
Install Ryu .....	59
VM2 – Central Open vSwitch.....	60
Install Open vSwitch: .....	61
Configure Networking.....	61
Configure Open vSwitch.....	62

VM3 - WAN Simulation .....	62
Build Guest.....	62
Configure Networking.....	64
VM4 – Remote Open vSwitch .....	65
VM5 - Client 1 .....	65
VM6 - Client 2 .....	65
Appendix C - Troubleshooting.....	66
Open vSwitch Troubleshooting.....	66
General Switch Commands.....	66
OpenFlow Commands.....	66
Change OpenFlow Version .....	66
Check Queueing .....	67
Pica8 Troubleshooting.....	67
Logs .....	67
Dummynet Troubleshooting .....	67
Appendix D - nmeta Caveats .....	67
Caveats .....	67
Future Enhancements .....	67



# Figures

Figure 1 - Network Usage Metadata .....	1
Figure 2 - Monolithic vs Software-Defined Network Paradigms .....	3
Figure 3 - Representative Comparison of Classifier Efficiency and Capability .....	12
Figure 4 - nmeta logical architecture.....	14
Figure 5 - Observed Packets vs Max Packet Size .....	17
Figure 6 - Last Directional Interpacket Arrival Delta .....	17
Figure 7 - Example Statistical Flow Entry Dictionary (code v6.2).....	18
Figure 8 - WAN3 Test Environment.....	22
Figure 9 - Physical Lab Environment .....	23
Figure 10 - Queueing Configuration .....	26
Figure 11 - Test Static-1 in Virtual Lab on code rev 5.6 .....	28
Figure 12 - Test Static-1 in Physical Lab on code rev 5.6 .....	29
Figure 13 - Test Static-2 in Virtual Lab on code rev 5.6 .....	29
Figure 14 - Test Static-2 in Physical Lab on code rev 5.6 .....	30
Figure 15 - Test Identity-1 in Virtual Lab on code rev 5.6 .....	36
Figure 16 - Test Identity-2 in Virtual Lab on code rev 5.6 .....	37
Figure 17 - Test Payload-1 in Virtual Lab on code rev 6.5.....	39
Figure 18 - Test Payload-2 in Virtual Lab on code rev 6.5.....	40
Figure 19 - Test Statistical-1 in Virtual Lab on code rev 6.2.....	42
Figure 20 - Test Statistical-2 (control) in Virtual Lab on code rev 6.2.....	43



# Chapter 1

## Introduction

The emergence of the Internet of Things (IoT) combined with a requirement for Quality of Service (QoS) in enterprise networks is driving a need for improved traffic classification techniques [1] [2]. This project investigates potential solutions to the real world problem of accurate and efficient traffic classification in enterprise networks. It proposes a solution to the problem and evaluates the functional performance of a prototype system.

### 1.1. Context

At its most fundamental, the usage of a data network can be described by two interlinked classes; firstly what connects to the network, and secondly how these participants communicate over the network.

In the analogy of a legacy circuit-switched Public Switched Telephone Network (PSTN), the first class describes the phone lines that connect to the network (i.e. phone number, physical address, billing name) and the second class the calls made over the network (i.e. calling-party, called-party, start time, duration).

In a packet-switched data network, this information about data (metadata) is very important for a number of use cases, including prioritisation and security. Conversation metadata (information about what communications occur) often has a gap between what the network knows (i.e. communications between pairs of network addresses) and what type of conversation actually occurs. The field of *traffic classification* attempts to fill this gap by identifying packets (or flows of packets) into types, so that they may be better understood, as per "Flow Enrichment" in Figure 1:

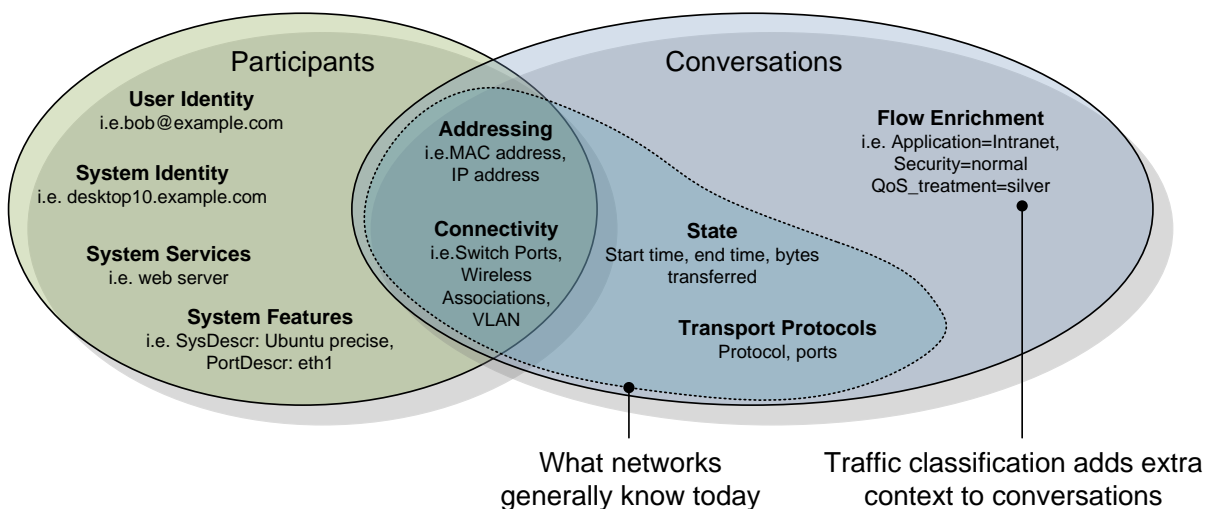


Figure 1 - Network Usage Metadata

Traffic classification is an easily over-looked problem. There is a temptation to move directly to questions of how to treat the traffic (i.e. what level of service should that type of flow receive?) without consideration first for how to identify the traffic. An *enterprise network* (a private network dedicated to carrying data communications for a single organisation) may have thousands, if not millions of packets in transit at any given moment, so it is a non-trivial exercise to identify the type of individual packets.

Traffic classification metadata is an important input into decisions regarding traffic treatment (i.e. decisions about priority and routing) as well as for other use cases such as security [3], billing [4] [5] and troubleshooting. A common consumer of traffic classification metadata is Quality of Service (QoS).

Using the previous PSTN analogy, QoS is required to ensure that calls to emergency services can proceed even when the network is overloaded with calls (a regulatory requirement). This is a simple classification task, as it is only necessary to check if the destination number is a member of the set of emergency services numbers to know if the call is an emergency services call. Traffic classification in a packet-switched data network is considerably more complicated, as identifying the type of conversation often requires analysis of more parameters than just the destination address. For example IP addresses may be assigned dynamically so are not necessarily a reliable indicator of identity, and popular IANA assigned TCP port numbers such as 80 and 443 are now used to carry a wide variety of traffic types.

Traffic classification and QoS are often used in enterprise networks to ensure acceptable user experience of applications, especially time sensitive ones such as voice and video, and critical line-of-business applications. Failure to implement QoS to protect these applications can result in poor user experience (examples: unintelligible audio, slow application response etc.) when the network is congested.

QoS is prevalent in enterprise networks is due to three conditions that are likely to exist:

1. Enterprise networks often have Service Level Agreements (SLA) between the operator of the network and business unit(s) [6]. This drives the adoption of network QoS as a means to ensure that Key Performance Indicators (KPI) can be met or exceeded; and
2. Enterprise networks may have a Wide Area Network (WAN) component that has lower bandwidth and higher latency than Local Area Networks (LAN), giving rise to the potential for congestion (contention for the use of limited bandwidth) and other impediments such as transmission and propagation delay. These conditions require effective QoS to protect service levels for important traffic flows.
3. Enterprise networks are generally under the control of a single entity, simplifying the deployment and maintenance of QoS, as it is not necessary to obtain the cooperation of multiple parties as is the case with public networks.

The advent of the *Internet of Things* (IoT) poses a growing challenge to effective traffic classification in enterprise networks. IoT is a fundamental change whereby a massive and diverse range of objects are becoming network addressable. This may be the networking of

previously unconnected electronic devices (i.e. security cameras, building management systems), but is also the embedding of networked computers into previously non-electronic items (i.e. signage, building structures, clothing). It is estimated that the number of Internet-connected devices will grow from approximately 2.5 billion in 2010 to between 50 and 100 billion by 2020 [7]. In enterprise networks this manifests itself as:

- a. An increase in the total number of IP-connected devices
- b. An increase in the number of distinct device types (increasing device heterogeneity)
- c. An increase in the volume of concurrent flows on the network

Scalable and accurate traffic classification is already a difficult problem. Roughan et al. say traffic classification "...is a challenging task, because many enterprise network operators who are interested in QoS do not know all the applications running on their network..." [8]. With the advent of IoT, the number of networked applications in an enterprise will likely grow significantly as new uses are found for the services that they provide.

A solution is required that at least partially automates traffic classification configuration so that organisations can efficiently and quickly apply and monitor traffic classification at a policy level, without having to make configurations on a per-flow, per-device or per-port basis.

## 1.2. Software-Defined Networking

Software-Defined Networking (SDN) is a promising new framework for traffic classification. It separates the forwarding and control functions of networking devices, making it possible to logically centralise control and apply a programmatic approach to the operation of a network, as per Figure 2

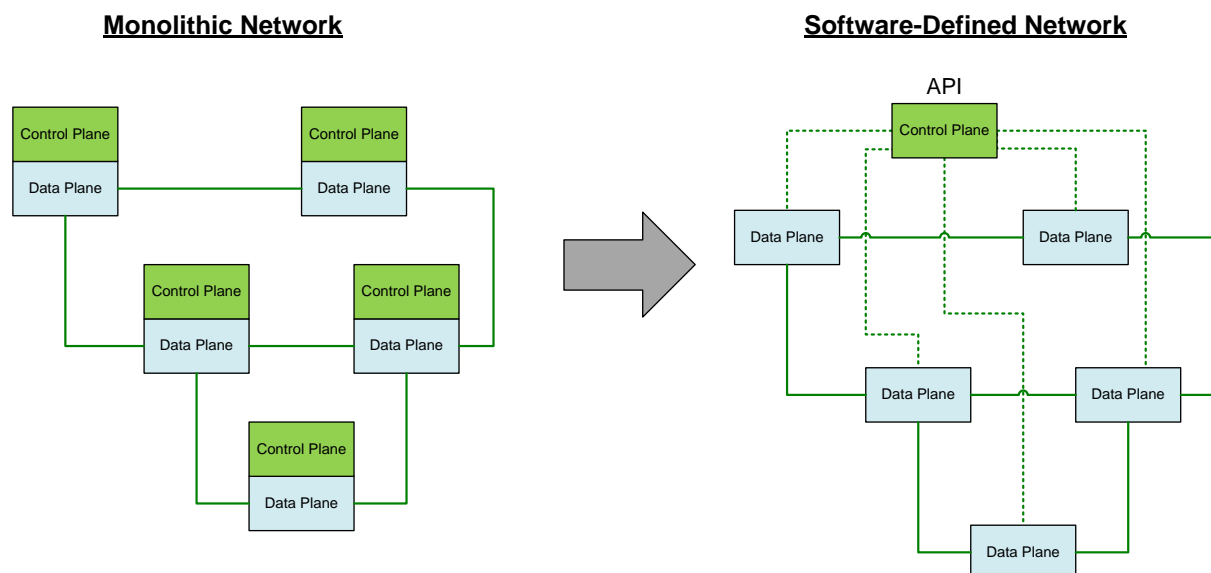


Figure 2 - Monolithic vs Software-Defined Network Paradigms

SDN is heralded as bringing innovation to the field of networking which has become subject to "ossification" [9] due to predominance of vertically integrated monolithic networking equipment. Monolithic networking cannot on its own deliver a system-wide view of flows, whereas this is inherent to the SDN architecture. Traffic classification in monolithic architecture must be run in separate 'islands' without a view of the complete system. This is both limiting and inefficient. The system-wide visibility of flows and potential for rapid innovation make SDN an appealing choice as the platform on which to develop improved traffic classification. For these reasons, this project proposes a solution that leverages the capabilities of SDN to deliver effective traffic classification for enterprise networks within the era of IoT.

### **1.3. Report Structure**

The remainder the report is structured as follows. Chapter 2 describes the problem in more detail, including analysis of related research. Chapter 3 outlines the design of an SDN-based traffic classification solution along with a description of the prototype system built for this project. It concludes by posing a hypothesis to be tested. Chapter 4 outlines the methodology and results from the evaluation of the prototype system against the hypothesis and chapter 5 presents conclusions. The appendices contain supplemental information.

# Chapter 2

## 2. Problem Description and Analysis

### 2.1. Problem Statement

Today, a detailed understanding of enterprise network traffic is required to design and configure traffic classification; however this becomes impractical as the number of different device types and hence flows on the network increases

### 2.2. Requirements for Traffic Classification in the Enterprise

Enterprise networks are heterogeneous; it is not possible to specify a standard example. Requirements are thus surmised from common conditions that may exist. Based on the experiences of the author, having worked in enterprise network design roles for more than 15 years, operators of enterprise networks are likely to have functional traffic classification requirements as per Table 1:

Requirement	Description	Rationale
<b>Selective Determinism</b>	Ability to set deterministic classifiers	Operators require consistent traffic classification behaviour for specified traffic types, so that actions (i.e. QoS treatment) can be performed on matching flows with a high degree of predictability.
<b>Agility</b>	Ability to classify unexpected traffic flows	A key tenet of the problem statement is that there are now too many flow types on the network for the operator to specify them all. Traffic classification must be able to intelligently classify unexpected flows.
<b>Application Awareness</b>	Can classify dynamic flows based on knowledge of application behaviour	Can appropriately classify related flows started from an initial known protocol. Some applications start extra dynamic connections (i.e. NFS, SIP starts RTP, etc.).
<b>Identity Awareness</b>	Support for classification based on endpoint identity	When devices were static it was relatively simple to write classification rules based on IP subnet/supernet as a surrogate for identity. With the proliferation of portable devices and wireless connectivity, IP addresses or subnets are no longer tied to a particular device and thus are not a good indicator of identity. For these reasons, operators desire a method to include other elements of identity in traffic classification rules.
<b>Timeliness</b>	Classifications are made within a short period of time, ideally before a large flow has had time to ramp up.	Timely classification is required for online consumers of traffic classification data, such as QoS and traffic engineering. There is no point applying QoS treatment to a flow if the classification data is not available until after the flow has finished. Timely classification is known as <i>online classification</i> [3].

*Table 1 - Enterprise Traffic Classification Requirements*

A number of non-functional requirements also exist. The system must be efficient so that it does not place undue load on network equipment or links, and does not materially degrade the

performance of the network. The system should make traffic classification results visible so that operators can check the validity of the results, report on them and use them for diagnostics. The system should be simple to operate. It should also be secure, scalable and highly available.

A common challenge identified in traffic classification literature dealing with Internet traces is the difficulty in establishing a *ground truth* from which the accuracy of traffic classification can be assessed during development and testing [8]. This is less of a problem in an enterprise network as operators are likely to have specific knowledge of their main applications and their features.

### 2.3. Categorising Traffic Classification Methods

Methods for classifying traffic can be broadly distributed into one or more of the following categories, as listed in Table 2:

Method	Description
<b>Static Classification</b>	Match any combination of parameters that appear in packet headers (i.e. link, network or transport layer features). Also referred to as a <i>port-based approach</i> [3]
<b>Trust</b>	Trust the end device to signal the classification of its traffic flows via some means to the network (i.e. setting DSCP field in IP packets). This is effectively a special case of <i>static classification</i> .
<b>Identity</b>	Identity-Based Classification combines a method of checking device identity with a traffic classification policy tailored for this identity [7].
<b>Payload Inspection</b>	Payload inspection, also referred to as Deep Packet Inspection (DPI) [3], involves pattern matching against packet payload (i.e. application-specific data encapsulated by the transport layer).
<b>Statistical Classification</b>	A statistical approach that builds classification signatures based on observed behaviour of traffic flows. Traffic metrics such as packet size and session duration are used in combination with statistical techniques (may include machine learning) to classify flows.

*Table 2 - Traffic Classification Methods*

Each method has strengths and weaknesses, as discussed in [2].

### 2.4. Analysis

Traffic classification is a classical QoS problem. While a broad range of academic papers cover traffic classification and QoS, there is a paucity of papers addressing the unique challenges of traffic classification in an enterprise environment, with the notable exception of [10]. The lack of focus on enterprise networks from the academic community may be due to the closed nature of networking systems used by many enterprises, and commercial privacy concerns that prevent researchers from being able to get sharable traces [3]. Additionally, the mechanics of enterprise networks, where many parallel paths can exist, present challenges to obtaining representative packet captures, as noted in [11].



Enterprise networks are a focus for this project as they a) have a requirement for improved traffic classification, b) are ripe for disruption due to their closed nature of their existing networking systems, c) are underrepresented in academic literature and d) are an area that receives substantial investment, with 7.5 billion US dollars spent worldwide on routers, switches and wireless LAN infrastructure in Q2 2014 [12].

## 2.5. Possible Solutions

Possible solutions proposed for traffic classification in enterprise networks in the era of IoT include:

### Payload Inspection

Various papers propose traffic classification schemes that use payload inspection methods, including LASER [13] and PortLoad [14].

*Payload inspection* (also known as Deep Packet Inspection (DPI)) involves pattern matching against packet payload data. Kim *et al.* [4] note that there are substantial drawbacks as payload inspection is "...resource-intensive, expensive, scales poorly to high bandwidths, does not work on encrypted traffic, and causes tremendous privacy and legal concerns..." [4]. The last point is generally moot in an enterprise network as employees usually waive their privacy rights through acceptance of the terms of their employment contract, but the point about encryption is valid as more applications move to secure connectivity and thus payload that cannot be inspected [3] [15].

Payload inspection relies on signatures and thus has the standard strengths and weaknesses associated with this type of approach - a low rate of false positives but a higher rate of false negatives [13]. With the heterogeneity of IoT devices and their flow types (who would write signatures?), it is unlikely that payload inspection on its own will provide much benefit for traffic classification.

### Statistical Classification

Statistical classification schemes are popular in academic papers. Examples include [15], [16], [17] and [18]. *Statistical classification* [8] builds classification signatures based on observed behaviour of traffic flows. Traffic metrics (referred to as *features*) such as packet size and session duration are used in combination with statistical techniques (including machine learning) to classify flows. This approach has the advantage that it simplifies configuration for traffic classification. It may however have limited adaptability to new flow types that it was not trained for, and removes deterministic control favoured by operators (i.e. fails Selective Determinism requirement). If fire alarm packets fail to get to their destination, it might be difficult to argue that the statistical algorithm used was the appropriate method of traffic classification (or indeed what classification it applied). Meeting the requirement for timeliness may be a challenge. Some papers attempt to address this through specific hardware, software and/or protocol updates. Interestingly Sanping Li *et al.* describe a clever system for processing statistical classification data at a high rate of throughput, but then specify that

"After a flow has timed out (packets matching that flow have stopped arriving at the switch), those flow features will be encapsulated and sent to the controller" [18]. This does not meet the requirement for timeliness as the traffic classification determination is not made until after the flow has completed.

## **Multiclassifier**

Roughan *et al.* [8] suggest that statistical classification could be augmented by combining with static classification to address determinism and improve accuracy. This view is echoed more recently by Dainotti *et al.* [3], and by Khalife *et al.* [19] in their 2014 paper on traffic classification taxonomy.

## **Role of SDN in Solutions**

SDN decouples the control plane and data planes, allowing the control plane to be logical centralised. The logical centralisation of network control gives rise to innovation opportunities not afforded by discrete monolithic network architecture. For instance, it becomes possible to have a system-wide view of the network flow state.

System-wide awareness of flows in monolithic networks requires bespoke solutions and/or use of identifiers carried within or around packets. Examples of the latter include use of the differentiated services field in IP packet headers [20]. In SDN architecture, the controller has a logically centralised view of the flow, removing the requirement to carry such administrative information in packets. This gives rise to innovation through the ability to write applications that leverage system-wide flow information.

SDN may also be able to assist with a common traffic classification research problem where privacy considerations prevent real network traffic from being studied. A possible solution is to supply the analysis system to organisations (i.e. enterprises) to run themselves, with only the resulting analytical data shipped back to the researchers [3]. This ensures that the research workers have no direct access to potentially private network data. SDN, when deployed on production networks, could allow researchers to construct systems that carry out traffic analysis without any requirement to install physical hardware. The system can be implemented as additional software on the SDN controller layer.

## **2.6. Hypothesis**

This project proposes the following hypothesis:

*SDN architecture is a suitable foundation for development of systems that can meet the functional traffic classification requirements of enterprise network operators.*

Specifically, the project will test the following sub-hypothesis:

1. *A traffic classification system built on SDN architecture can accurately classify traffic with static classification method*

2. *A traffic classification system built on SDN architecture can accurately classify traffic with identity classification method*
3. *A traffic classification system built on SDN architecture can accurately classify traffic with payload classification method*
4. *A traffic classification system built on SDN architecture can accurately classify traffic with statistical classification method*

## **2.7. Chapter Summary**

In this chapter, the problem of traffic classification in enterprise networks and requirements has been stated and traffic classification methods defined. Next relevant academic research has been summarised along with possible solutions. Finally, a hypothesis and sub-hypotheses have been posed to prove.

In the following chapter a design for a SDN-based traffic classification system is outlined, and a prototype system that was developed for this project is detailed.



# Chapter 3

## 3. Design

This chapter proposes a design for a SDN-based traffic classification system, and introduces the prototype system that was developed for this project.

### 3.1. Architecture

This project leverages SDN architecture by classifying new flows at the SDN controller layer, thus leveraging the software flexibility and processing power that SDN affords. Once a flow is classified, fine-grained classifiers are installed to the switch packet forwarding tables for efficient switching without further recourse to the controller.

Switch matches are fine-grained so that all new flows observed by a switch are sent to the SDN controller classifiers. This reactive approach facilitates system-wide flow visibility and application of policy to classifier configuration.

The system is a framework supporting multiple classification methods, and can thus be described as a *multiclassifier* (refer previous chapter). Classifiers can be *specific* (return a Boolean describing whether or not a match is made) or *general* (return parameters describing what they classified). Classifiers can be combined in a logical structure through use of a policy.

#### OpenFlow

OpenFlow [21] is a well-known architecture and protocol for establishing and maintaining control of the data plane. OpenFlow was chosen for the role of SDN protocol in the design due to its current popularity, large development community and non-proprietary nature. In the OpenFlow architecture, simple traffic classifiers, called flow entries, are installed onto switches. A flow entry contains *match fields* which vary dependant on the OpenFlow version. Where implemented in hardware, flow entry classifiers have the advantage of being relatively fast, but may have capacity and capability constraints [9]. As they occur within the data plane, their capabilities are dependent on the particular switch implementation, and they cannot directly leverage network knowledge outside of the switch view. The trade-off between latency and capability between hardware switch, software switch and SDN controller based classifiers is shown in Figure 3:

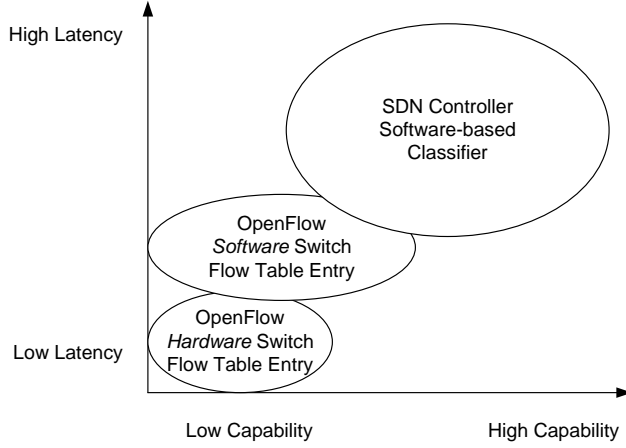


Figure 3 - Representative Comparison of Classifier Efficiency and Capability

Hardware switch classifiers are relatively fast, but their capability is often limited due to constraints of the ASICs on which they are built. Software switch classifiers may be slower than their hardware equivalents, but are likely to have better feature support as their development is not dependant on support in silicon. SDN classifiers are slower again due to the time taken to send packet(s) to the controller; however the benefits of software-development freedom, along with a system-wide view of flows, are judged to out-weigh the performance downsides in the architecture used by this project. All flows are classified initially by SDN controller classifiers and fine-grained classifiers are installed to switches once classification determinations are made.

### 3.2. Introduction to the Prototype System

A prototype SDN multiclassifier framework was developed for this project. It has policy-based classifier controls and produces enriched metadata output. The prototype system is called *nmeta*, short for **n**etwork **meta**data. It runs on top of the Ryu [22] SDN controller. Ryu was chosen due to use and familiarity within the Victoria University ECS faculty. Ryu is written in the Python [23] programming language and *nmeta* is also written in Python to take advantage of existing code development on Ryu.

The *nmeta* framework is novel in that it is the first solution (that the author is aware of) to employ a policy-based multiclassifier system on top of SDN architecture to provide extensible output in the form of enriched flow metadata.

The *nmeta* framework is a good solution to the functional requirements as:

- The capability to set policy statements specifying traffic feature match parameters and actions meets the *selective determinism* requirement
- The requirement for *agility* can be met by sending unmatched traffic to a statistical classifier
- *Application awareness* can come from policy that directs payload inspection on specific flow types (with knowledge of the protocol). Protocols that establish dynamic flows can be matched through payload inspection of the control packets.

- *Identity Awareness* comes from policy statements that reference identity metadata

### 3.3. Design Principles

Nmeta employs a modular design that decomposes major tasks into separate modules with public interfaces and hidden implementation (note that Python has limitations in this area [24]). This standard software design principle improves maintainability of code, since changes within a module are less likely to have unforeseen consequences outside the module.

The nmeta code has been written in partial compliance to the Python PEP-8 [25] coding conventions. Time limitations have prevented full compliance from being achieved.

Components of nmeta are grouped into *regions* that share a common purpose.

The *nmeta Core region* (refer orange shaded area in Figure 4) manages communications with switches (i.e. processing of packet-in and switch messages, adding flows etc) via Ryu and handles incoming REST API calls via the Ryu Python Web Server Gateway Interface (WSGI) libraries. It also reads in the main configuration file on initialisation. There is only one module in this region, *nmeta.py*. Packet-in messages are processed sequentially through the *\_packet\_in\_handler* function.

The *Traffic Classification region* (refer blue shaded area in Figure 4) classifies packets against a traffic classification policy and returns results to nmeta Core. The *tc\_policy.py* module reads in a traffic classification policy on initialisation, evaluates incoming packets against the policy and sends them to the appropriate classifier module (if required). The four classifier modules *tc\_static.py*, *tc\_identity.py*, *tc\_payload.py* and *tc\_statistical.py* are discussed in a following section.

The *Flow Metadata region* (refer purple shaded area in Figure 4) is called after forwarding decisions are made so that they can be incorporated in the resulting metadata. It stores the enriched metadata in a Python data structure called a *dictionary* [26], and controls the installation of flow match entries to switches.

The *Metadata Consumer - QoS region* (refer the red shaded area in Figure 4) is a simple stub that provides a QoS treatment (queue assignment) based on matching a QoS flow metadata tag against a simple QoS policy. Note that QoS treatment is not in scope for this project so this region has been implemented as just the bare minimum required to run the test use cases.

All communication from the traffic classification region to the flow metadata region is via the nmeta core region. This rule is to ensure that a future forwarding module has visibility of traffic classification status messages.

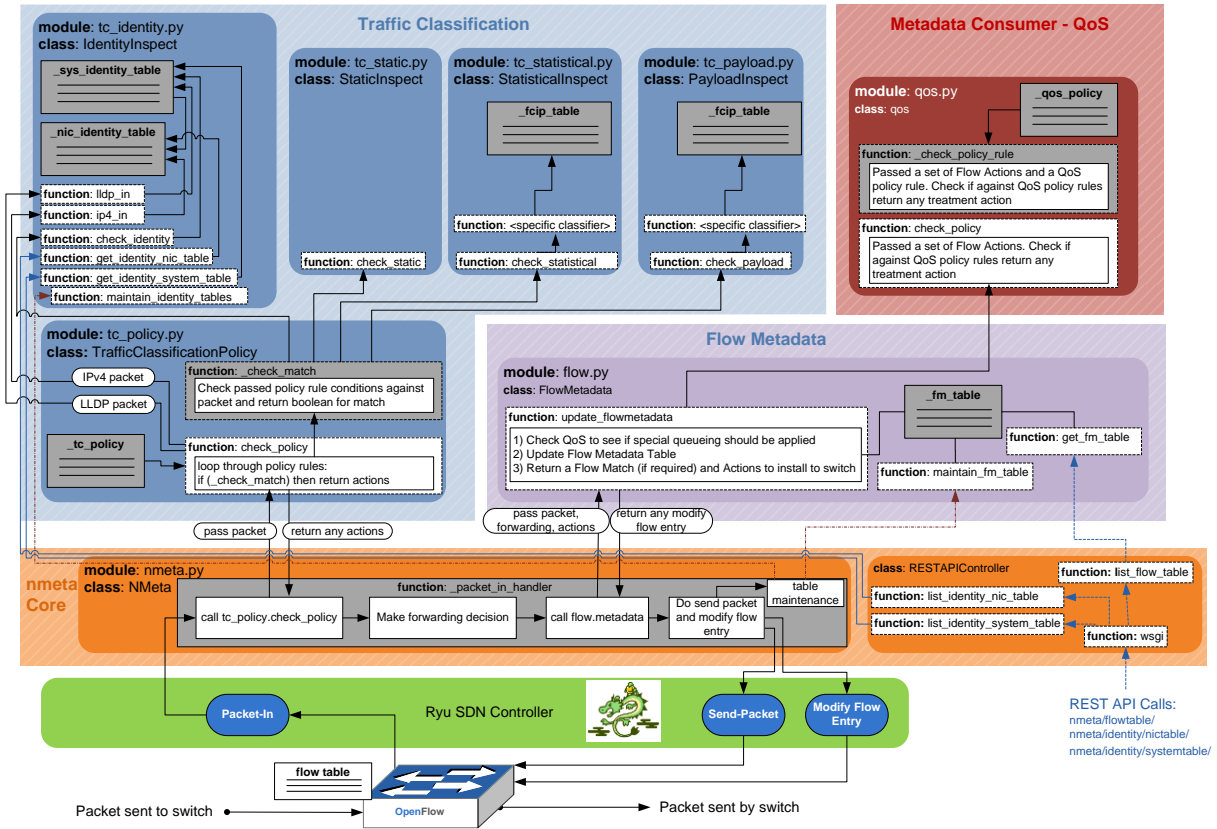


Figure 4 - nmeta logical architecture

### 3.4. Traffic Classification Modules

This section describes the modules with the traffic classification region.

#### Traffic Classification Policy Module

The traffic classification policy module reads in a policy configuration file called *tc\_policy.yaml* from the config subdirectory on initialisation. This file is in YAML format and describes the policy controlling the use of classifiers. The file is validated to ensure it contains only supported values. The public function *check\_policy* is called on every packet-in event so is written with efficiency in mind. It validates the incoming packet against the policy and decides what, if any, classifiers should be invoked. It also checks if packets need to be seen by the identity module and if so calls the appropriate identity function. Match results, and any other metadata, are returned to the nmeta core region.

#### Static Classification Module

Static classification is implemented as a simple *if/elif/else* Python block matching policy attributes, and checking their validity against the supplied packet. A Boolean is returned indicating the result of the match.



## Identity Classification Module

The identity classification module records the identity of endpoints that broadcast Link Layer Discovery Protocol (LLDP) messages. LLDP is widely supported, but not secure.

Identity classification can be set to match against values *chassisid* or *systemname* LLDP attributes. The match can be a partial match defined as a regular expression for *systemname*.

Identity information is stored in two dictionaries, one for Network Interface Controller (NIC) identities and the other for system identities. The system dictionary references entries in the NIC dictionary and vice versa. Two dictionaries are required since an endpoint may have multiple NICs. LLDP Packet-in events are used by the identity module to accumulate system information and likewise, IPv4 Packet-in events are used to accumulate MAC address to IPv4 address linkages in the NIC dictionary.

Matching against a *chassisid* or *systemname* value requires first checking if the value is present in the system dictionary. If present, the referenced NIC dictionary entry (caveat: code needs updating to deal with multiple NICs) is retrieved and the packet is compared to see if it matches against the MAC or IPv4 values. If it does, a True value is returned otherwise False.

## Payload Classification Module

It is often necessary to observe multiple packets in a flow before payload is present, and hence the payload classification module must understand flows [3]. Nmeta defines a bi-directional TCP flow as a 4-tuple of *ip\_a*, *ip\_b*, *tcp\_port\_a*, *tcp\_port\_b*. Packets can be matched as a flow in either direction as long as the TCP port numbers pair correctly with the IP addresses. A data structure called the *Flow Classification In Progress* (FCIP) table (a Python dictionary) is used to store flow classification state. A *continue\_to\_inspect* flag is used to indicate to the flow module that it should not install a flow to the switch as more packets need to be observed.

A single specific payload classifier is implemented in nmeta for matching FTP control and data traffic. This FTP payload classifier does an initial static match on packets with source or destination TCP port 21 to filter out FTP control traffic with minimal overhead. Matching packets are checked to see if they have a TCP payload and if they do this is checked for a match on the last 8 hex characters against pattern '504f5254'. Payload that matches this is dissected to obtain the FTP dynamic port number. If a dynamic port number is obtained this is added to the FCIP table so that packets in the dynamic port flow will be classified as FTP.

## Statistical Classification Module

Statistical classification also requires an understanding of flows. The statistical classification module has the same concept of flows, FCIP data structure, and ability to signal whether or not to install a flow as the payload classification module. Where it differs, is in the ability to return actions, rather than just a Boolean for a match. The ability to return actions is required to be able to indicate between multiple possible results, such as classifying a flow to one of *n*

traffic types. Arguably, payload inspection would require this same capability if a general purpose payload classifier was developed, and it would be simple to retrofit.

The following parameters of a bi-directional TCP flow are recorded by the module:

- Packet arrival times
- Packet sizes
- Packet directionality
- TCP flags
- TCP window size
- TCP acknowledgement numbers

Packet arrival times record when a packet arrives at the classifier module. Ideally, to remove variability due to backhaul transmission and controller processing, the value would instead be the arrival time of the packets at the switch. The accuracy of module arrival time was found to be sufficient for this project, but it is worth noting that OpenFlow Feature eXtraction (OFX), as proposed by Sanping Li *et al.* [18], could provide access to more accurate data in future if implemented by switch manufacturers.

TCP window size presented a challenge, since the SYN and SYN+ACK packets carry a directional *Window Scale* option in the TCP header [27]. Knowledge of this value, per direction, is required to be able to compute the actual TCP window size on subsequent packets from their indicated value. Packets with the TCP.SYN flag set are parsed for *Window Scale* option, and if present it is recorded in the FCIP flow record, noting the direction. These values are then used on subsequent packets to calculate the true window size.

A single demonstration statistical classifier, referred to as *statistical\_qos\_bandwidth\_1*, was developed for this project to demonstrate a basic statistical classification capability. It marks aggressive flows so that QoS can treat them as less than best effort class, thus protecting traffic in the default class.

The *statistical\_qos\_bandwidth\_1* classifier was limited to only analysing TCP traffic for reasons of simplicity. To develop the classifier, traffic flows of the following protocols were analysed:

- SSH (Interactive)
- SSH (SCP)
- Iperf (TCP)
- HTTP

The results showed that Iperf had a specific traffic profile characterised by a rapid increase in packet size at packet 5 and consistently low interpacket arrival time deltas from packet 4 onwards as showing in Figure 5 and Figure 6.

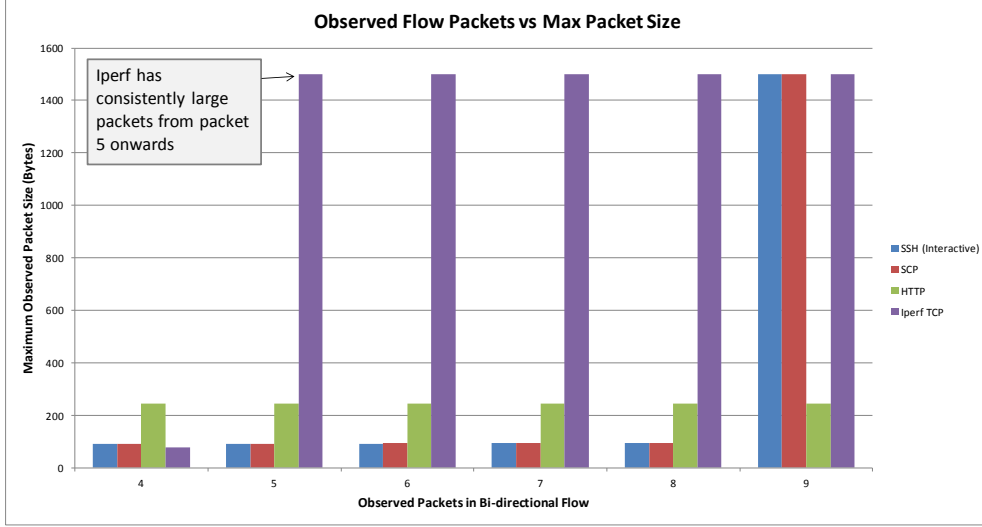


Figure 5 - Observed Packets vs Max Packet Size

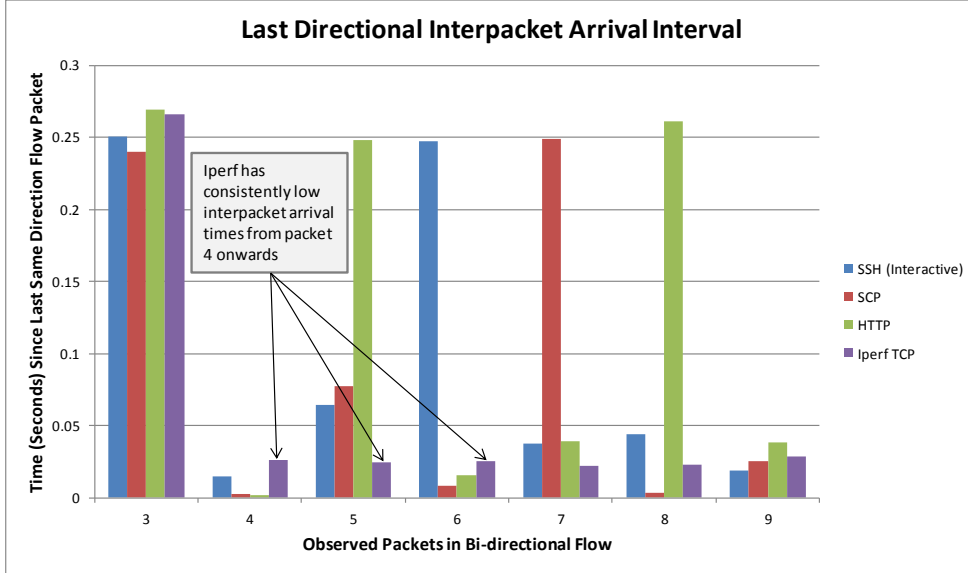


Figure 6 - Last Directional Interpacket Arrival Delta

Based on the above results, and drawing inspiration from papers such as [15], the statistical\_qos\_bandwidth\_1 classifier was configured as follows:

- Only match TCP flows (for simplicity)
- Carry out analysis of the flow after 5 packets observed:
  - Retrieve the maximum packet size value ( $P$ )
  - Calculate the minimum directional interpacket arrival time delta ( $D_{\min}$ )
  - Calculate the maximum directional interpacket arrival time delta ( $D_{\max}$ )
- Interpacket ratio ( $I$ ) calculated from  $D_{\min} / D_{\max}$ . Note the use of a ratio to reduce the influence of a base latency on the result.
- Threshold for maximum packet size ( $T_{P_{\max}}$ ) set to 1200 (bytes)
- Threshold for minimum interpacket ratio ( $T_{I_{\min}}$ ) set to 0.25

- If ( $P > T_{Pmax}$ ) and ( $I < T_{Imin}$ ) then return action specifying flow treatment as low\_priority otherwise default\_priority

The thresholds are set to match Iperf traffic with its very aggressive profile, but not to match other flows types.

Figure 7 shows an example statistical dictionary record for an Iperf flow:

```
{
  'direction':
    {1: 'forward', 2: 'reverse', 3: 'forward', 4: 'forward', 5: 'forward'},
  'window_size':
    {1: 29200, 2: 28960, 3: 29312, 4: 29312, 5: 29312},
  'max_packet_size':
    {2: 60, 3: 60, 4: 76},
  'number_of_packets':
    5,
  'bits':
    {1: 2, 2: 18, 3: 16, 4: 24, 5: 16},
  'TCP_SYN':
    {1: 'SYN', 2: 'SYN'},
  'ack':
    {1: 0, 2: 1697666226, 3: 1714517706, 4: 1714517706, 5: 1714517706},
  'ip_total_length':
    {1: 60, 2: 60, 3: 52, 4: 76, 5: 1500},
  'tcp_A':
    56945,
  'actions':
    0,
  'tcp_B':
    5001,
  'arrival_time':
    {1: 1410644589.143512, 2: 1410644589.315387, 3: 1410644589.412362, 4: 1410644589.421452, 5: 1410644589.464328},
  'finalised':
    0,
  'window_scale':
    {'forward': 7, 'reverse': 6},
  'ip_B':
    '192.168.56.12',
  'ip_A':
    '192.168.57.40'
}
```

Figure 7 - Example Statistical Flow Entry Dictionary (code v6.2)

### 3.5. Nmeta Supplementary Features

#### Configuration

Nmeta configuration is separated from code where practical to reduce need to modify code and allow customisation that is persistent through software upgrades. Configuration is stored in text files in the *config* subdirectory. Configuration files are written in YAML [28] format.

YAML was chosen as a format due to its concise nature, human readability and capability to represent arbitrary data structures. Nmeta leverages a Python YAML module to read in configuration files, ensuring their compliance to YAML standards, translating them into Python dictionaries.

Nmeta carries out additional checks to ensure that dictionaries representing configuration files contain only expected attributes and values. Where exceptions are found, they are logged with a clear explanation, and where necessary the program is halted at this point to prevent undesirable behaviour.

#### Data Management

Various dynamic data structures exist within nmeta requiring maintenance to prevent unchecked growth that compromises system performance and/or availability. Nmeta runs

table maintenance at the end of the packet in handler to minimise delays to packet out and flow install events. Timers are consulted, and if the delta from the previous maintenance is greater than the configured threshold then maintenance functions in modules are called to prune entries that are older than define maximum ages.

Time constraints prevented a more thorough data management regime from being implemented. Ideally, there should also be configurable maximum table size limits to prevent resource exhaustion, and also management of flow table sizes on switches.

## **REST API**

A REST API provides read-only access to flow and identity metadata.

### **3.6. Non-Functional Considerations**

The author chooses to out-of-scope all non-functional requirements, as project time constraints and restrictions on maximum report size prevent this sizeable area from being addressed properly. A few noteworthy considerations are listed in this section.

#### **Performance Considerations**

Nmeta is a single threaded Ryu application so is susceptible to blocking [29]. The REST API shares the same thread, so could cause performance degradation if called on a large dictionary while the system is under load.

There is an opportunity to improve performance for situations where classifiers are used that need to see more than the first packets in a flow (i.e. payload and statistical) on flows that cross multiple switches. The current nmeta behaviour is to require a packet-in from each switch in each direction until the classification has been made. Multiple switches result in duplicate packet-in events being sent to the controller that add no value. Nmeta is configured to ignore these duplicate packet-in events; however it is worth noting that they impact performance as they add load to the backhaul and controller. They also delay the forwarding of the packet until the controller has sent a packet-out message. If there are  $n$  switches and  $x$  packets must be observed then there will be  $x(n-1)$  duplicate packet-in events. To improve this situation, the controller could install flow table entries to all but one of the in-path switches, and update these entries if required based on the traffic classification determination.

#### **Security Considerations**

Enterprises take security very seriously. It is unlikely that SDN will take hold in enterprises until it can be shown to be as secure as monolithic networking. As detailed later in this document, it appears that SDN is lacking maturity in the area of security.

Note that OpenFlow traffic should be protected to ensure confidentiality and integrity. In nmeta OpenFlow traffic is passed in plain text, which is great for troubleshooting, but not for security.

### **Scalability Considerations**

It should be possible to scale controllers horizontally using some means to maintain loose consistency of data; however this has not been investigated by this project.

### **3.7. Chapter Summary**

In this chapter, a design for a SDN-based traffic classification system has been outlined, and a prototype system developed for this project has been presented in detail. The following chapter details the methodology and results of the evaluation of the prototype system and concludes by using the results to prove the hypothesis.

# Chapter 4

## 4. Evaluation

This chapter evaluates the functional performance of the *nmeta* prototype system in various traffic classification scenarios.

### 4.1. Evaluation Methods

Formal methods exist for evaluating the performance of traffic classification, such as overall accuracy, precision, recall and F-Measure [3] [4], however these are too rigorous for the requirements of a COMP489 project. Instead, evaluation will focus on the QoS use case as it can show clear and tangible benefits.

The prototype *nmeta* framework has been written for this project to test the validity of the hypothesis. It is used in all the evaluation tests as the traffic classifier framework.

### 4.2. Lab Environments

Lab environments are required to partially simulate an enterprise network with a WAN component, so that suitability and performance of the proposed solution can be tested and analysed.

Inclusion of a simulated WAN is important to demonstrating the feasibility of the solution as many enterprise networks connect geographically dispersed sites. WAN latency and bandwidth constraints pose design challenges that should be considered when evaluating solutions for enterprises.

The lab environments include one additional challenge - reticulation of the control traffic via the data plane. When an SDN deployment is limited to a single site, i.e. a data centre, it is feasible to use a separate network for the backhaul of traffic between the switches and SDN controller(s). In an enterprise with physically distributed sites, it is not likely to be practical or cost effective to run a separate out of band control network. For this reason the lab environments use the same data plane for standard network traffic to transport the control traffic to the SDN controller.

#### Virtual Lab Environment

A virtualised lab environment (referred to as *WAN3*) has been built within an Oracle VirtualBox hypervisor on top of a Microsoft Windows 7 PC as per Figure 8. Instructions for building the *WAN3* environment are included in Appendix A.

A virtualised environment has the advantage of being quick to configure and easy to make changes to. Downsides are that performance results may not be accurate due to variability in the virtualisation hypervisor and underlying host machine, and that virtualised switches may not accurately emulate the behaviour of hardware switches.

A central enterprise site is simulated by two Ubuntu guests. One guest provides a switching function and the other acts as a server for SDN control and testing functions. A WAN link is emulated by Dummynet [30] on a FreeBSD guest running as a router. It allows the setting of bandwidth, delay and packet loss values.

A remote WAN site is simulated by three Ubuntu guests. One guest provides a switching function and the other two are clients to support test functions.

Both switches run Open vSwitch software and are controlled via OpenFlow from the central SDN controller.

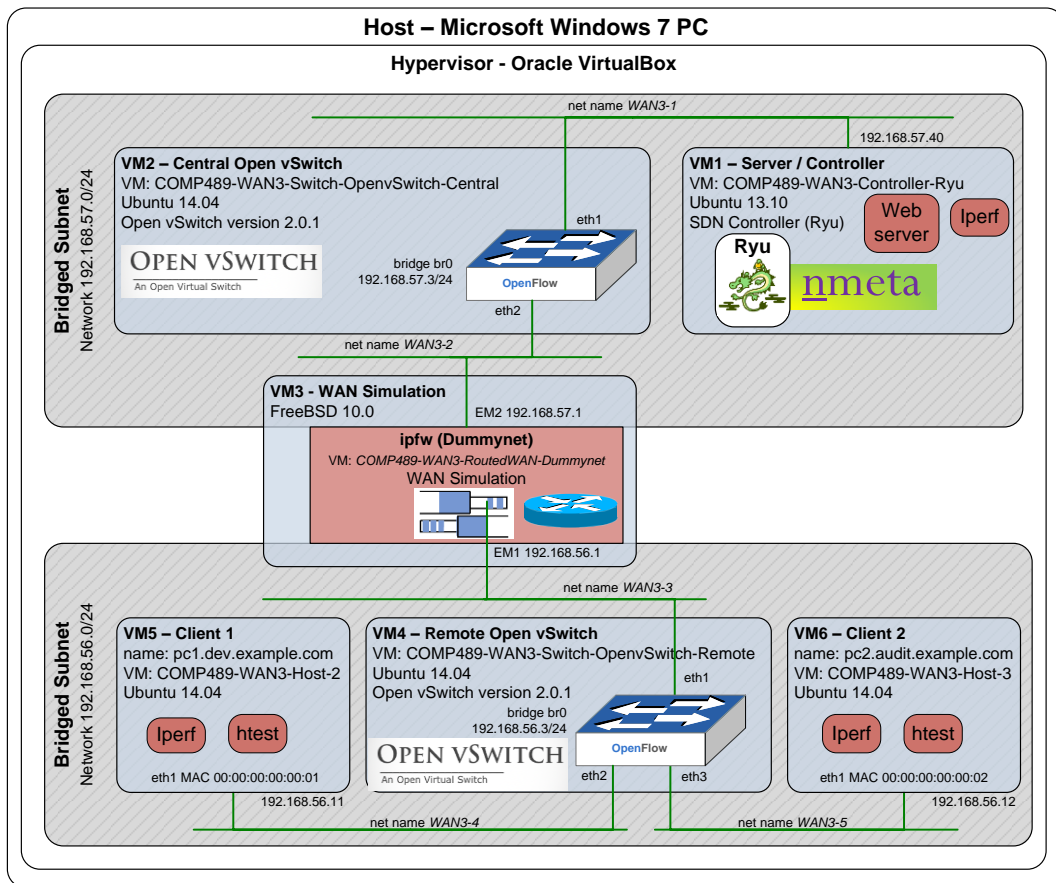


Figure 8 - WAN3 Test Environment

Five virtual networks are defined within Oracle VirtualBox for connectivity between guests. Note that while there are five virtual network connections, there are only two IP subnets present as the switches connect multiple segments into single subnets.

Important lessons were learned while building the environment:

- Guest network interfaces on the switches need to be set as *promiscuous* within Oracle VirtualBox. Without this being set, the guest interfaces will only receive packets destined for their MAC address or to the broadcast MAC address as the VirtualBox hypervisor runs internal networks as switches. In order for the guest to operate as a



switch (aka multi-port bridge) it must receive all packets on the network segment excluding those that it has sent.

- Open vSwitch is a lot easier to install on Ubuntu 14.04 than older versions due to in-tree kernel support
- FreeBSD is a good platform for Dummynet as it has built-in support for it.

## Physical Lab Environment

A physical lab environment was built utilising hardware SDN switches as per Figure 9:

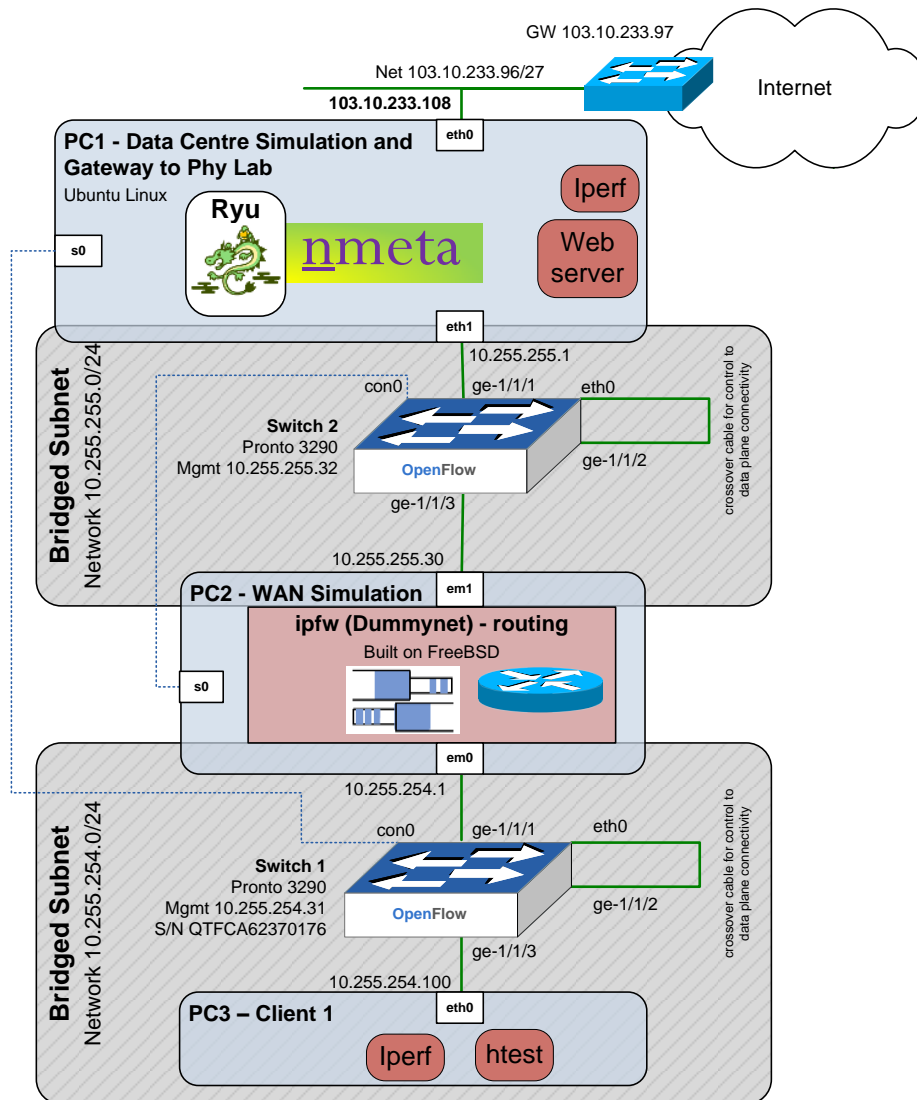


Figure 9 - Physical Lab Environment

Components:

- PC1 connects to the Internet, allowing remote access to the environment. It has been secured with Linux kernel firewall via iptables. Additionally, the SSH daemon has

been configured not to accept password connections, only specific SSH keys. PC1 simulates a central enterprise site and has applications running SDN control and server functions.

- A Pica8 Pronto P-3290 switch provides SDN hardware switch functionality connecting PC1 and PC2
- PC2 emulates a WAN link with Dummynet [3] on a FreeBSD guest running as a bridge, allowing the setting of bandwidth, delay and packet loss values.
- A Pica8 Pronto P-3290 switch provides SDN hardware switch functionality connecting PC2 and PC3
- PC3 is a client to support test functions

Asynchronous serial console cables are run from Linux PCs to Pica8 switches for out-of-band management connectivity for major reconfigurations.

Ethernet cross-over cables connect the management ports on the Pica8 switches to data plane ports. This inelegant workaround allows control plane traffic to traverse the data plane.

### **Bandwidth Congestion**

Iperf [31] is used to create network congestion on the WAN link, except for tests Payload-1 and Payload-2 where FTP is used.

### **HTTP Response Time Measurements**

The measurement of HTTP response times needs to be contained within a TCP session to avoid traffic classification overhead on subsequent GET requests. To achieve this result, the Keep-Alive header of HTTP/1.1 [32] was used. Additionally, HTTP content must not be cached as this would invalidate the results.

A simple python program was written for this project to automate HTTP load time testing, leveraging the Python *Requests* library [33] for HTTP/1.1 functionality. The program is called *htest.py* and it is passed a test URL (can include a port number) on the command line. It loops indefinitely with a one second sleep between tests. Results for *Requests elapsed.total\_seconds* and total elapsed test time and a time stamp are written to standard output and also to a text file in CSV format. Care was taken to ensure that the underlying HTTP adapter would retry multiple times to simulate a real browser connection and '*Connection*': '*keep-alive*' was specifically set in the HTTP header. The output to CSV file made it easy to import data into a spreadsheet for analysis.

Full details of evaluation methodology are detailed in [Appendix A - Test Details](#).

## **4.3. Test Use Cases**

Four sets of test use cases were developed for this project to evaluate the functional capabilities of the prototype system. Test use cases start off simple and increase in complexity, with traffic classification methods added in order of difficulty to allow for progressive and incremental code development.

The WAN link is set to a bandwidth of 2Mbps with a round-trip time of 40mS. These values are fairly arbitrary - they represent a rough mid-point between low bandwidth and high bandwidth WAN scenarios.

Clients are located at the remote site and servers at the central site. OpenFlow Protocol traffic between the remote site switch and the SDN controller traverses the WAN link.

The following QoS queueing policy is common across all use cases:

```
'PolicyRule 0':  
  comment: OpenFlow traffic  
  QoS_treatment: system_priority  
  output_queue: 0  
'PolicyRule 1':  
  comment: Default priority traffic  
  QoS_treatment: default_priority  
  output_queue: 1  
'PolicyRule 2':  
  comment: High priority traffic  
  QoS_treatment: high_priority  
  output_queue: 2  
'PolicyRule 3':  
  comment: Low priority bandwidth hungry traffic  
  QoS_treatment: low_priority  
  output_queue: 3
```

The policy defines queues into which traffic can be differentially assigned based on the value of the attribute '*QoS\_treatment*'. The rules work as follows:

- Flows tagged with '*system\_priority*' will be queued in queue 0. This queue is for OpenFlow Protocol traffic to/from the SDN controller, as it is the default queue used by switches before controller connectivity is established. Use of this queue for OpenFlow traffic prevents a bootstrap problem, as the initial OpenFlow Protocol traffic cannot be assigned to a queue as connectivity to the controller hasn't been established yet.
- Flows tagged with '*default\_priority*', and flows with no attribute '*QoS\_treatment*' tag with recognised value, will be queued in queue 1.
- Flows tagged with '*high\_priority*' will be queued in queue 2. This is intended for time-sensitive business-critical traffic.
- Flows tagged with '*low\_priority*' will be queued in queue 3. This queue is intended for bandwidth-hungry but non-time sensitive flows.

Lab switches were configured with four egress queues on ports that face the WAN simulator:

q0 = used for OpenFlow traffic, max rate 300Kbps

q1 = default, max rate 500Kbps

q2 = high priority, max rate 1Mbps

q3 = low\_priority, max rate 100Kbps

The queueing configuration is depicted in Figure 10:

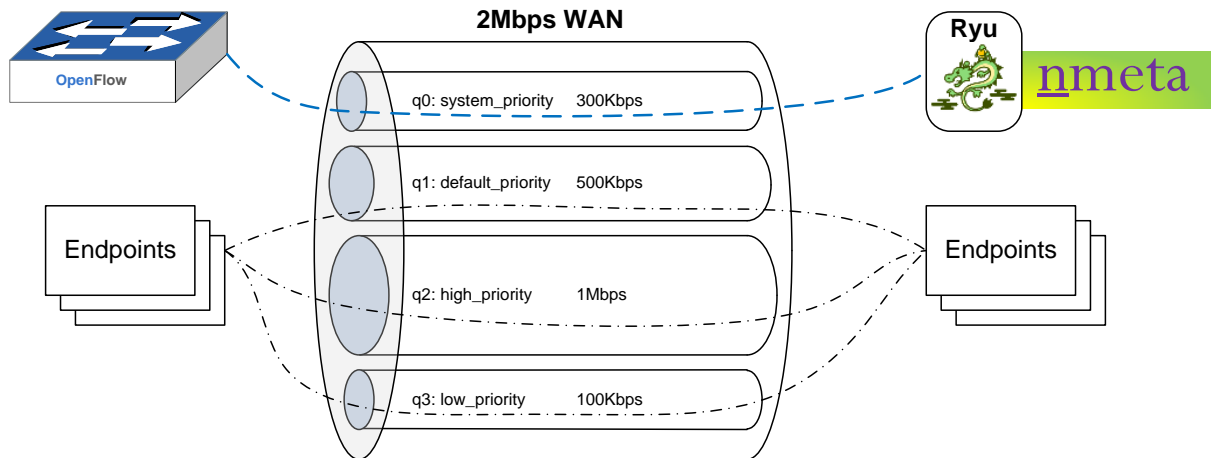


Figure 10 - Queueing Configuration

All queues are configured with a bandwidth ceiling of 1.8Mbps, which should allow bursting over the queue reserved bandwidth when other bandwidth is available, although this did not work in practice on either switch type.

#### 4.4. Test Use Cases Static-1 and Static-2

##### Goal

Demonstrate that basic static classification can classify and treat connectivity to a server on a specific port differently to other traffic.

##### Method

Client1 makes regular HTTP/1.1 connections to Server1 on tcp-1234 and tcp-80. Both connections are used to retrieve the same HTML object and are contained within persistent TCP sessions. Timing results are recorded.

After establishing a baseline, Iperf is used to congest the link in the default class in the server to client direction for a sustained period. This direction was chosen as it is the direction in which the majority of test traffic flows. Iperf is then terminated and the test runs for a further period to recheck baseline.

In test Static-1, the network is configured to classify and treat tcp-1234 and tcp-22 (SSH) connections as high priority, tcp-6633 (OpenFlow Protocol) as system priority and all other traffic as default.

Test Static-2 is a repeat of Static-1, with the only difference being an update of the traffic classification policy to treat tcp-80 as high priority instead of tcp-1234. This second test is used demonstrate that there were no other factors at play relating to the chosen TCP port numbers, other than the traffic classification and QoS treatment.

## Desired Outcome(s)

- 1) Load times for HTTP objects over tcp-1234 in test Static-1 are not materially affected by the link congestion (target less than 10% increase in response times)
- 2) Load times for HTTP objects over tcp-80 in test Static-1 are noticeably affected by the congestion (expect >100% increase in response times)
- 3) Load times for HTTP objects over tcp-80 in test Static-2 are not materially affected by the link congestion (target less than 10% increase in response times)
- 4) Load times for HTTP objects over tcp-1234 in test Static-2 are noticeably affected by the congestion (expect >100% increase in response times)

## Configuration

The following configuration was applied in the nmeta environment for test Static-1:

```
'PolicyRule 0':
  comment: OpenFlow Protocol Traffic
  match_type: any
  policy_conditions:
    tcp_src: 6633
    tcp_dst: 6633
  actions:
    set_qos_tag: QoS_treatment=system_priority
    set_desc_tag: description="OpenFlow Protocol Traffic"
'PolicyRule 1':
  comment: Use Case Static-1 - High Priority Business Traffic
  match_type: any
  policy_conditions:
    tcp_src: 1234
    tcp_dst: 1234
  actions:
    set_qos_tag: QoS_treatment=high_priority
    set_desc_tag: description="High Priority Business Traffic"
'PolicyRule 2':
  comment: SSH traffic
  match_type: any
  policy_conditions:
    tcp_src: 22
    tcp_dst: 22
  actions:
    set_qos_tag: QoS_treatment=high_priority
    set_desc_tag: description="High Priority SSH Traffic"
```

The following configuration was applied in the nmeta environment for test Static-2 (note the change to TCP ports in PolicyRule 1):

```
'PolicyRule 0':
  comment: OpenFlow Protocol Traffic
  match_type: any
  policy_conditions:
    tcp_src: 6633
    tcp_dst: 6633
  actions:
    set_qos_tag: QoS_treatment=system_priority
    set_desc_tag: description="OpenFlow Protocol Traffic"
'PolicyRule 1':
  comment: Use Case Static-1 - High Priority Business Traffic
```

```

match_type: any
policy_conditions:
  tcp_src: 80
  tcp_dst: 80
actions:
  set_qos_tag: QoS_treatment=high_priority
  set_desc_tag: description="High Priority Business Traffic"
'PolicyRule 2':
comment: SSH traffic
match_type: any
policy_conditions:
  tcp_src: 22
  tcp_dst: 22
actions:
  set_qos_tag: QoS_treatment=high_priority
  set_desc_tag: description="High Priority SSH Traffic"

```

## Results

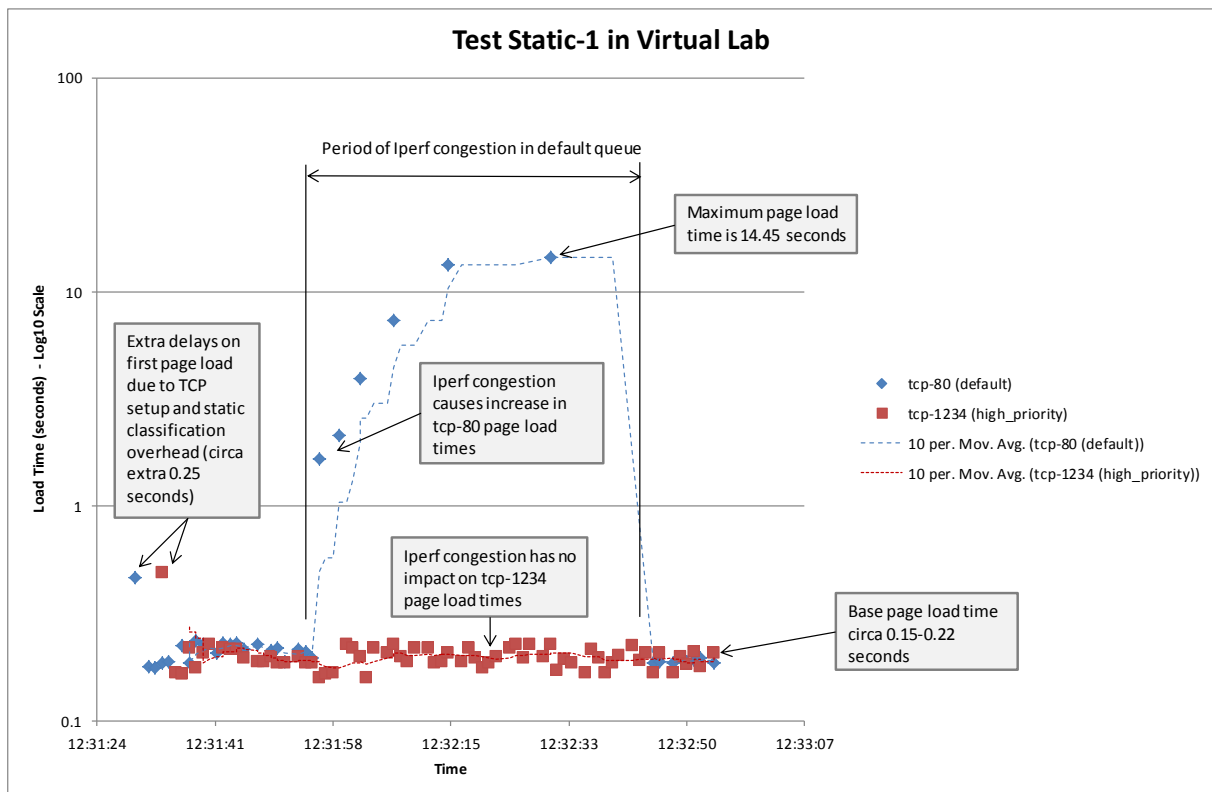


Figure 11 - Test Static-1 in Virtual Lab on code rev 5.6

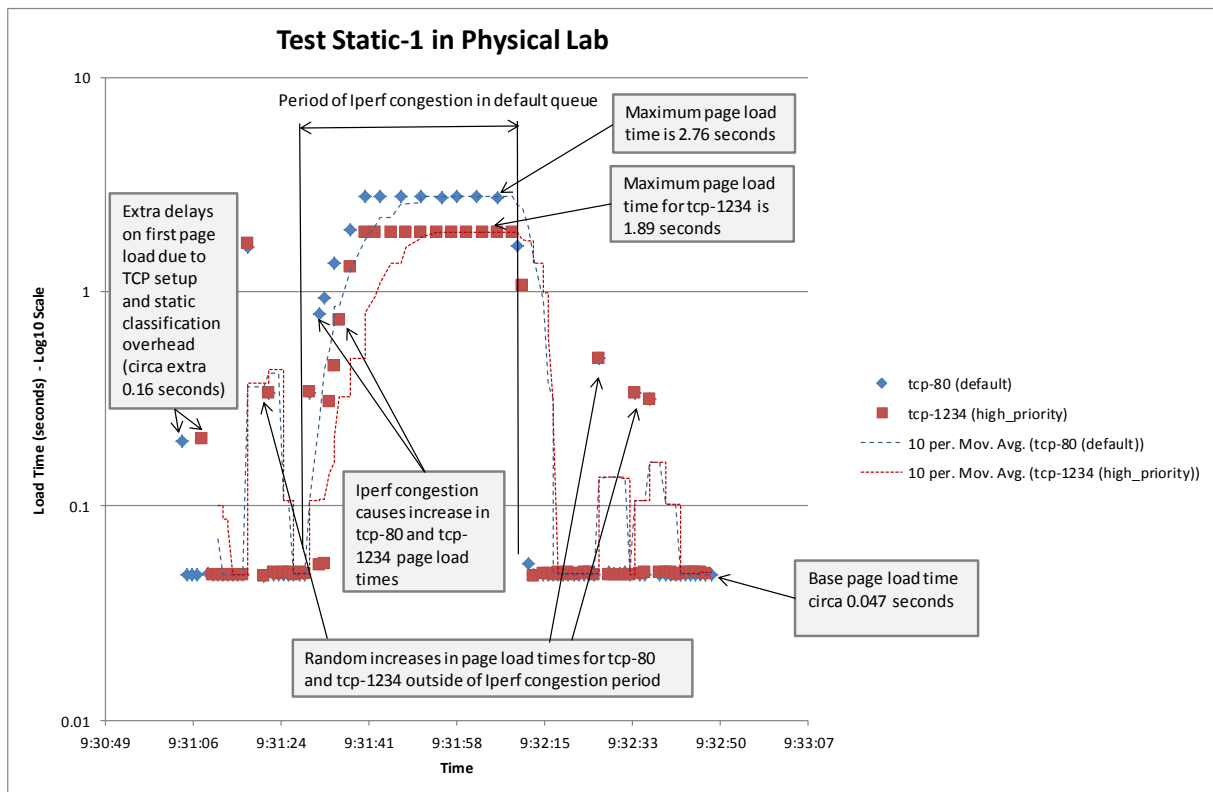


Figure 12 - Test Static-1 in Physical Lab on code rev 5.6

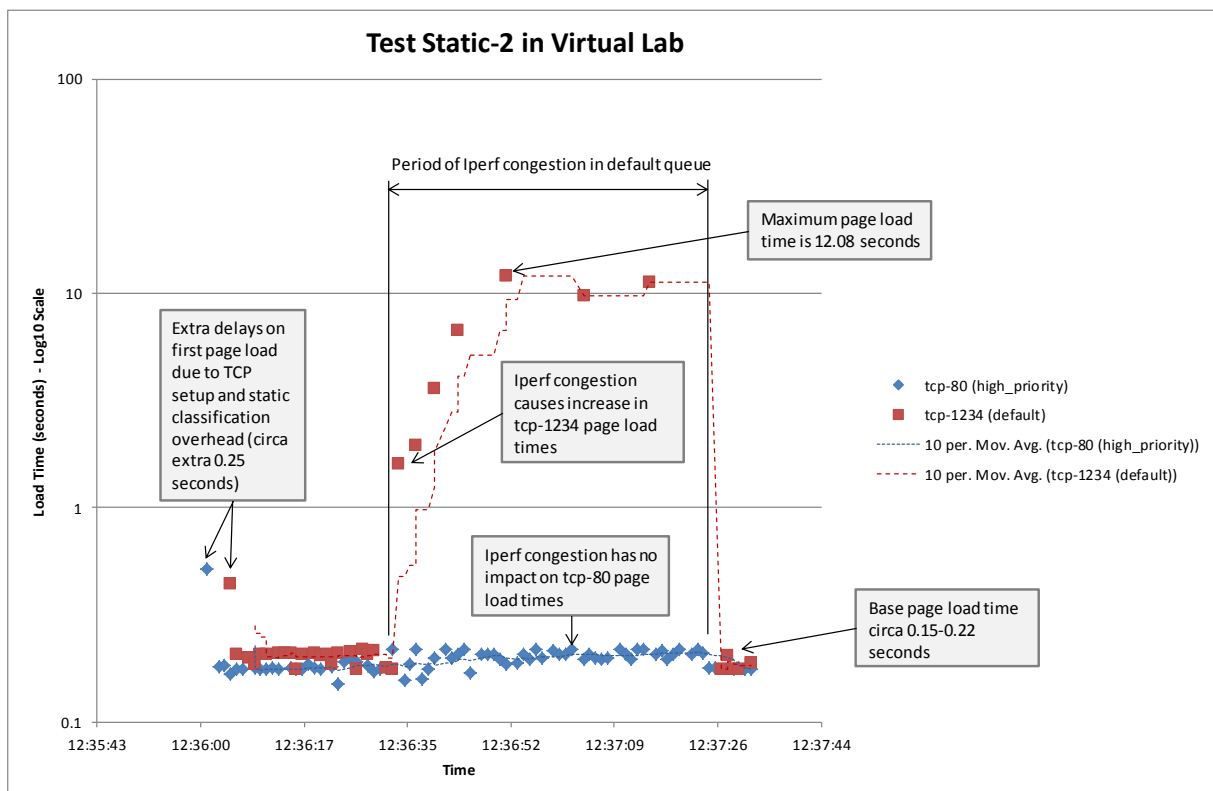


Figure 13 - Test Static-2 in Virtual Lab on code rev 5.6

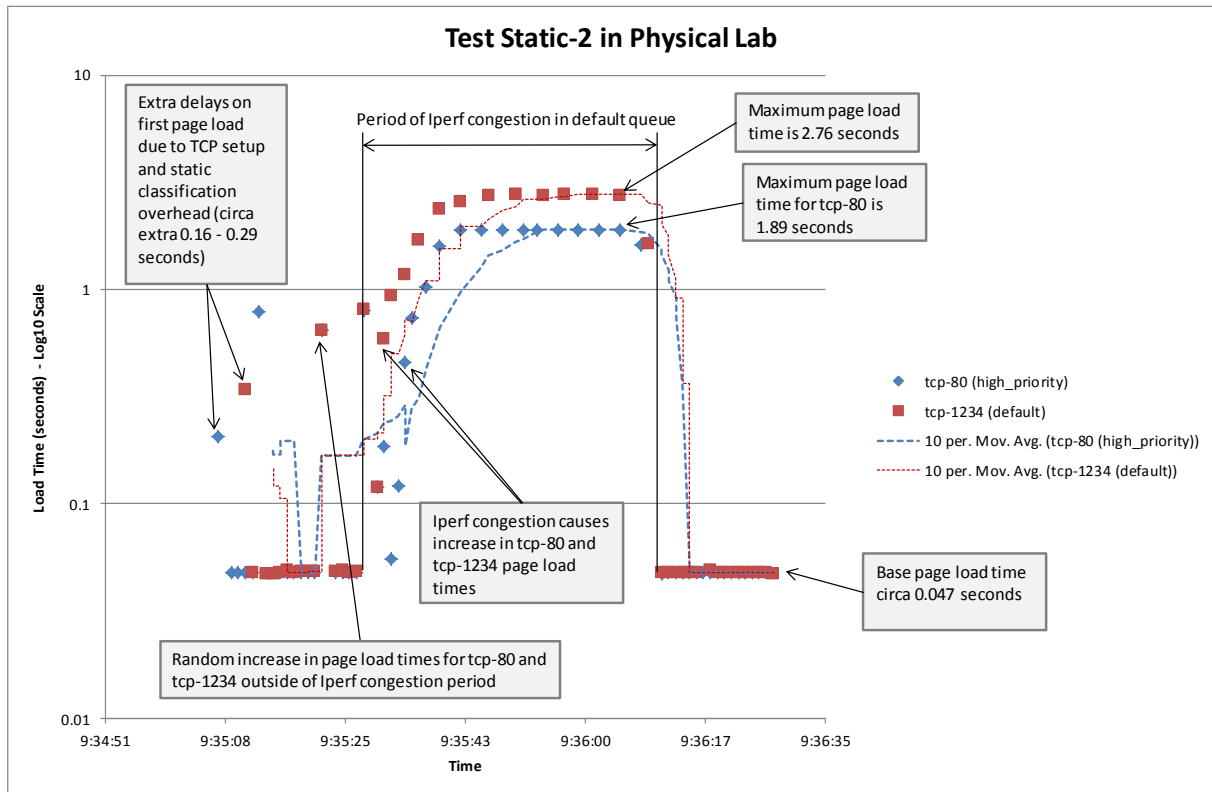


Figure 14 - Test Static-2 in Physical Lab on code rev 5.6

## Repeatability Test and Results

Additional iterations of test Static-1 with higher numbers of concurrent Iperf TCP streams were run to evaluate statistical significance of the results in the virtual lab. Ten tests were run with Iperf, starting with a set of ten concurrent sessions. Twenty samples of load time measured in seconds were recorded, then a further 10 tests with 50 concurrent sessions were run and the results are tabulated in Table 3:



		TCP 80 (20 Samples)					TCP 1234 (20 Samples)				
		Average (s)	Max (s)	Min (s)	Range (s)	StdDev (s)	Average (s)	Max (s)	Min (s)	Range (s)	StdDev (s)
10 Concurrent Iperf Sessions	Test 1	4.91	75.61	0.17	75.44	17.41	0.24	1.21	0.15	1.06	0.18
	Test 2	2.15	40.41	0.18	40.23	7.36	0.22	0.78	0.16	0.62	0.09
	Test 3	4.81	54.90	0.17	54.73	12.99	0.30	1.73	0.16	1.57	0.28
	Test 4	3.12	33.05	0.17	32.88	7.43	0.29	1.34	0.16	1.18	0.23
	Test 5	4.07	43.23	0.17	43.06	9.91	0.28	2.35	0.14	2.20	0.27
	Test 6	4.14	42.97	0.17	42.80	9.30	0.23	0.79	0.16	0.63	0.11
	Test 7	4.69	47.90	0.18	47.73	11.26	0.26	0.87	0.15	0.71	0.14
	Test 8	4.07	41.27	0.18	41.09	9.34	0.30	1.38	0.16	1.22	0.26
	Test 9	3.85	25.54	0.18	25.36	7.86	0.34	2.70	0.16	2.54	0.36
	Test 10	2.89	16.34	0.18	16.16	4.83	0.28	1.62	0.16	1.46	0.20
50 Concurrent Iperf Sessions	Test 11	9.84	91.01	0.18	90.83	23.91	0.32	1.17	0.16	1.01	0.24
	Test 12	5.19	76.93	0.18	76.75	18.14	0.39	3.40	0.17	3.23	0.44
	Test 13	6.89	63.03	0.16	62.87	19.02	0.40	2.43	0.16	2.26	0.37
	Test 14	7.34	55.65	0.18	55.48	16.94	0.33	2.00	0.16	1.84	0.30
	Test 15	5.44	51.11	0.18	50.94	14.38	0.41	2.40	0.17	2.22	0.37
	Test 16	6.39	56.31	0.18	56.13	15.69	0.46	4.13	0.17	3.96	0.57
	Test 17	16.29	206.35	0.17	206.18	48.10	0.41	5.65	0.17	5.48	0.73
	Test 18	2.61	51.05	0.18	50.87	9.41	0.52	3.70	0.16	3.54	0.68
	Test 19	10.62	128.23	0.17	128.06	32.52	0.36	1.67	0.16	1.51	0.34
	Test 20	9.33	132.15	0.17	131.99	32.15	0.39	3.55	0.15	3.40	0.48

*Table 3 - Statistical Analysis for Test Static-1 in Virtual Lab*

The statistical analysis in Table 3 shows that the classification results are consistent and reproducible. The average load time across all twenty runs for tcp-1234 is around 0.34s while it takes 5.9s for tcp-80. Moreover, the range (Max-Min) for tcp-80 connections is two orders of magnitude higher than those in tcp-1234. Finally, the variance for tcp-1234 is small and consistent across the first ten runs, and only slightly higher in the next 10 runs. All these three statistical measures strongly suggest that the classifier is performing correctly and consistently and further support the significance of the results obtained through the methods used.

## Analysis

- Response times for first tests in each series were higher due to overhead of TCP session establishment and flow classification via SDN controller. In the virtual lab the first tests took approximately 0.25 seconds longer, whereas in the physical lab they took approximately 0.16 seconds longer. The overheads of virtualisation are likely to have contributed to the higher first test page load time in the virtual lab when compared to the physical lab. Both the higher first page load times and the difference between virtual and physical are expected behaviour.
- Response times for HTTP connections on tcp-1234 in test Static-1 in the virtual lab were not materially affected by the link congestion, meeting the expectations of desired outcome 1.
- Response times for HTTP connections on tcp-80 in test Static-1 in the virtual lab were noticeably affected by the congestion. Maximum response time was an increase by a

factor of 80 over the pre-congestion time. This meets the expectations of desired outcome 2.

- Response times for HTTP connections on tcp-80 were not materially affected by the link congestion in test Static-2 in the virtual lab, meeting the expectations of desired outcome 3.
- Response times for HTTP connections on tcp-1234 in test Static-2 in the virtual lab were noticeably affected by the congestion. Maximum response time was an increase by a factor of 68 over the pre-congestion time. This meets the expectations of desired outcome 4.
- Physical lab results were unreliable (increased response times observed outside of Iperf congestion period) and desired outcomes 1 & 2 were not met. The traffic in the high\_priority queue was impacted by the Iperf congestion to within 68% of the increase observed in the default\_priority queue. The cause is the hardware queueing implementation on the Pica8 Pronto P-3290 switches, although the exact cause remains unknown.
- Repeatability experiment of test Static-1 in the virtual lab showed a very low level of standard deviation for the prioritised page load times over tcp-1234 demonstrating the reliability of the traffic classification, even under significant congestion. It also proved that the results in the virtual lab have a high degree of repeatability.

### **Summary of Static Traffic Classification Findings**

Test Static-1 passed in virtual lab but failed in the physical lab. The failure in the physical lab is as a result of the hardware queueing implementation on the Pica8 Pronto P-3290 switches, not the nmeta software. The physical lab test failure highlights the importance of choosing SDN switch hardware carefully, and testing it to ensure that it meets requirements.

Subsequent tests are run exclusively in the virtual lab as the failure of the physical lab tests would be repeated due to the commonality of queueing across the testing suite. The scope of this report is limited to traffic classification and thus the observed difficulties with QoS treatment on the physical switches are out of scope. Additionally, a comparison of the baseline and peak measurements between physical and virtual labs shows a comparable ratio and reliable no-congestion readings in the virtual environment. Repeatability tests also showed a consistency of results across multiple runs of the same test. For these reasons, the virtual lab is a suitable environment for the remaining tests.

## **4.5. Test Use Cases Identity-1 and Identity-2**

### **Goal**

Demonstrate that identity classification can classify traffic to provide differential treatment of connectivity to/from a particular endpoint.

## Method

Traffic classification is configured to treat as high priority any connections to or from hosts that have an LLDP system name of *\*.audit.example.com*.

Client1 with LLDP system name *pc1.dev.example.com* is not matched by the identity classification.

Client2 has an LLDP system name of *pc2.audit.example.com* and has its connections classified and treated as high priority based on the configured wildcard match for *\*.audit.example.com*.

Both Client1 and Client2 make regular HTTP connections to Server1 on tcp-80 and retrieve the same HTML object. Timing results are recorded.

After establishing a baseline, Iperf from Server 1 to Client1 and Client2 is used to congest the link in the default class for a sustained period. Iperf is then terminated and the test runs for a further period to recheck baseline.

Test Identity-2 is a repeat of Identity-1, with the only difference being an update of the traffic classification policy to treat *\*.dev.example.com* as high priority instead of *\*.audit.example.com*. This second test is used demonstrate that there were no other factors at play relating to the chosen TCP port numbers, other than the traffic classification and QoS treatment.

## Desired Outcome(s)

- 1) In test Identity-1, load times for HTTP connections from Client2 (*pc2.audit.example.com*) to the server are not materially affected by the Iperf congestion of the link (target less than 10% increase in response times)
- 2) In test Identity-1, response times for HTTP connections from Client1 (*pc1.dev.example.com*) to the server are noticeably affected by the congestion (expect >100% increase in response times)
- 3) In test Identity-2, load times for HTTP connections from Client1 (*pc1.dev.example.com*) to the server are not materially affected by the Iperf congestion of the link (target less than 10% increase in response times)
- 4) In test Identity-2, response times for HTTP connections from Client2 (*pc2.audit.example.com*) to the server are noticeably affected by the congestion (expect >100% increase in response times)

## Configuration

An identity rule carried out a regular expression match against the domain name portion of the LLDP system name. A rule was used to explicitly classify Iperf traffic on tcp-5001 into the default queue, as otherwise it would be set as high priority when sent to or from clients matching the identity rule.

The following configuration was applied in the nmeta environment for test Identity-1:

```

'PolicyRule 0':
  comment: OpenFlow Protocol Traffic
  match_type: any
  policy_conditions:
    tcp_src: 6633
    tcp_dst: 6633
  actions:
    set_qos_tag: QoS_treatment=system_priority
    set_desc_tag: description="OpenFlow Protocol Traffic"
'PolicyRule 1':
  comment: Explicitly set Iperf traffic to default class
  match_type: any
  policy_conditions:
    tcp_src: 5001
    tcp_dst: 5001
  actions:
    set_qos_tag: QoS_treatment=default_priority
    set_desc_tag: description="Default Priority Iperf Traffic"
'PolicyRule 2':
  comment: SSH traffic
  match_type: any
  policy_conditions:
    tcp_src: 22
    tcp_dst: 22
  actions:
    set_qos_tag: QoS_treatment=high_priority
    set_desc_tag: description="High Priority SSH Traffic"
'PolicyRule 3':
  comment: Use Case Identity-1 - High Priority Business Traffic
  match_type: any
  policy_conditions:
    identity_ldp_systemname_re: '.*\audit\example\.com'
  actions:
    set_qos_tag: QoS_treatment=high_priority
    set_desc_tag: description="High Priority Business Traffic"

```

The following configuration was applied in the nmeta environment for test Identity-2 (note the change in the regular expression in PolicyRule 3):

```

'PolicyRule 0':
  comment: OpenFlow Protocol Traffic
  match_type: any
  policy_conditions:
    tcp_src: 6633
    tcp_dst: 6633
  actions:
    set_qos_tag: QoS_treatment=system_priority
    set_desc_tag: description="OpenFlow Protocol Traffic"
'PolicyRule 1':
  comment: Explicitly set Iperf traffic to default class
  match_type: any
  policy_conditions:
    tcp_src: 5001
    tcp_dst: 5001
  actions:
    set_qos_tag: QoS_treatment=default_priority
    set_desc_tag: description="Default Priority Iperf Traffic"
'PolicyRule 2':
  comment: SSH traffic
  match_type: any
  policy_conditions:

```

```
    tcp_src: 22
    tcp_dst: 22
    actions:
      set_qos_tag: QoS_treatment=high_priority
      set_desc_tag: description="High Priority SSH Traffic"
'PolicyRule 3':
  comment: Use Case Identity-1 - High Priority Business Traffic
  match_type: any
  policy_conditions:
    identity_lldp_systemname_re: '.*\dev\example\com'
  actions:
    set_qos_tag: QoS_treatment=high_priority
    set_desc_tag: description="High Priority Business Traffic"
```

## Results

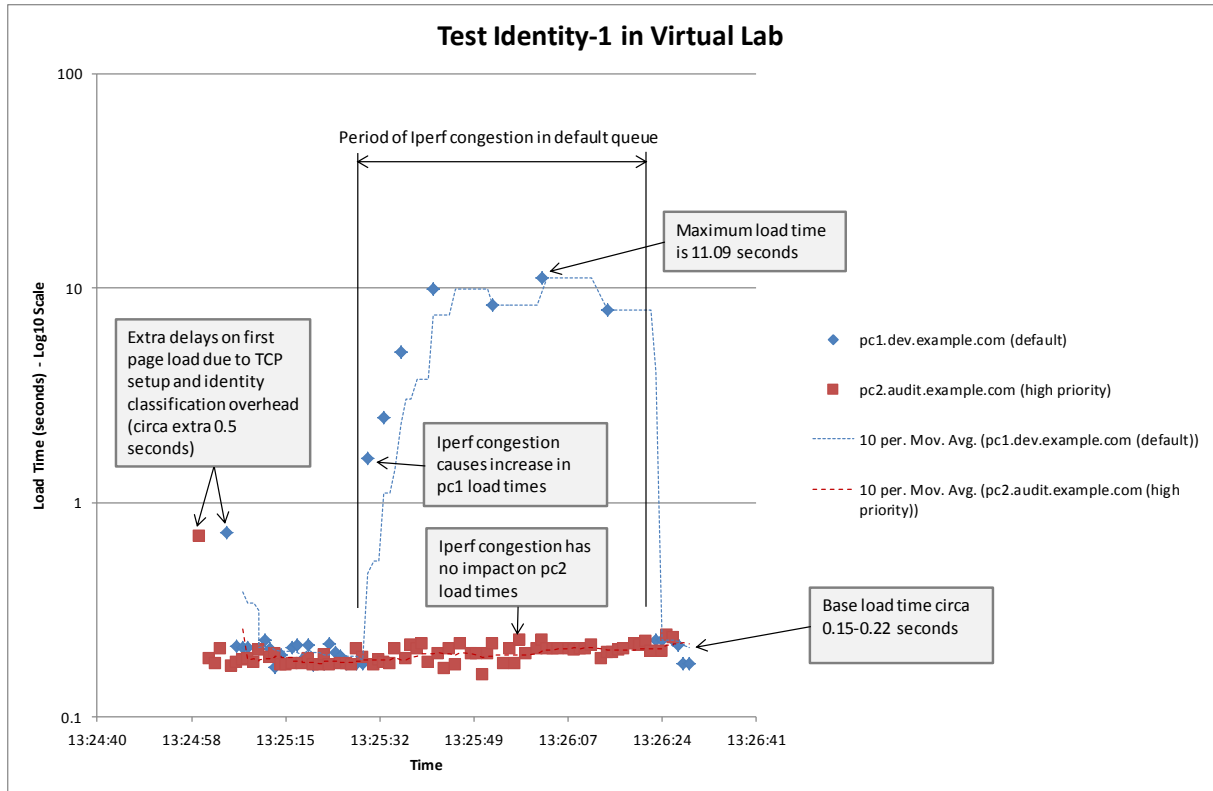
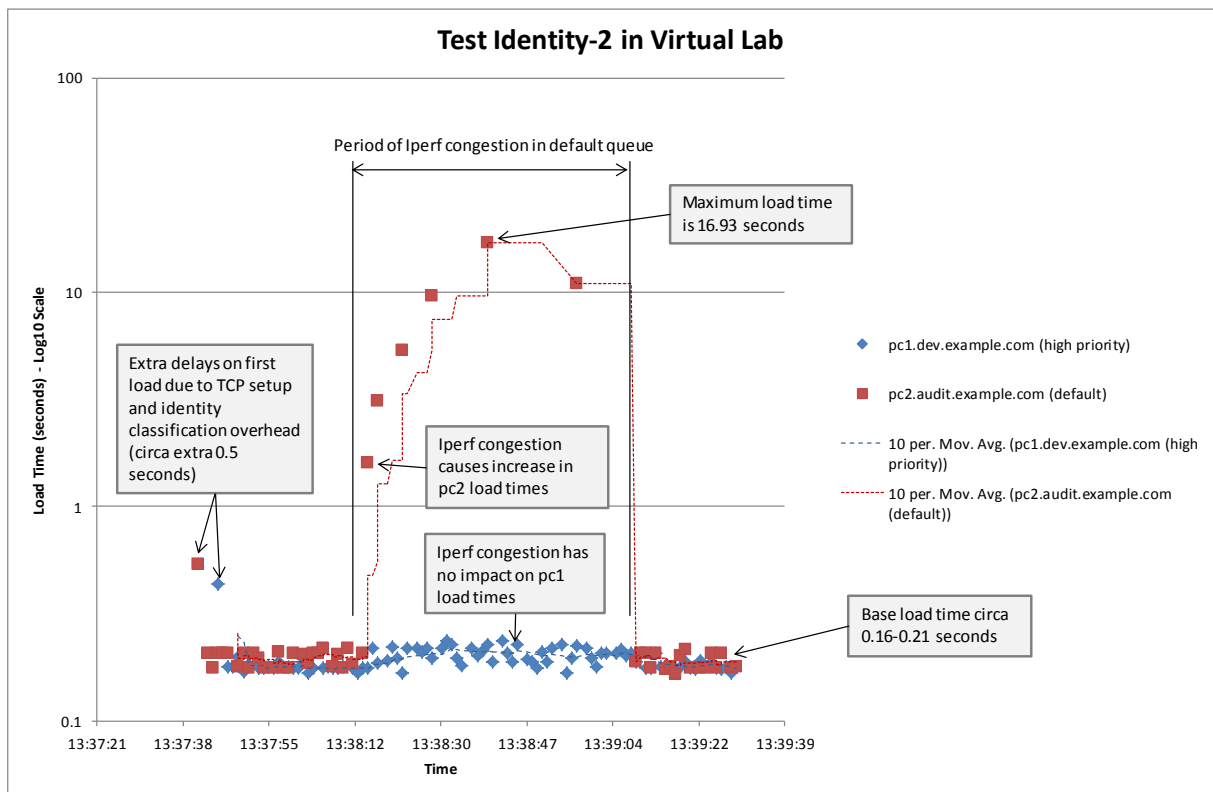


Figure 15 - Test Identity-1 in Virtual Lab on code rev 5.6



*Figure 16 - Test Identity-2 in Virtual Lab on code rev 5.6*

## **Analysis**

- Client2 (pc2.audit.example.com) HTTP object load times in test Identity-1 were not materially affected by the Iperf link congestion, meeting the expectations of desired outcome 1.
- Client1 (pc1.dev.example.com) HTTP object load times in test Identity-1 were noticeably affected by the congestion. Maximum load time was an increase by a factor of 55 over the pre-congestion time. This meets the expectations of desired outcome 2.
- Client1 (pc1.dev.example.com) HTTP object load times in test Identity-2 were not materially affected by the Iperf link congestion, meeting the expectations of desired outcome 3.
- Client2 (pc2.audit.example.com) HTTP object load times in test Identity-2 were noticeably affected by the congestion. Maximum load time was an increase by a factor of 85 over the pre-congestion time. This meets the expectations of desired outcome 4.

## **Summary of Identity Traffic Classification Findings**

Tests Identity-1 and Identity-2 passed, proving that it is possible to classify traffic based on identity in an SDN environment. The use of wildcard match on identity, as demonstrated, would be a desirable feature to operators dealing with scale issues.

Traffic differentiation was applied in both directions on the matched flows, including on the switch not directly connected to the identified device. This ability to make a system wide determination and apply it to all elements on the traffic path is an advantage conferred by SDN.

## **4.6. Test Use Cases Payload-1 and Payload-2**

### **Goal**

Demonstrate that basic payload classification can classify and differentially treat connectivity over a specific protocol, including traffic on a separate flow with dynamically assigned port.

### **Method**

Traffic classification is configured to treat any connections with payload match on FTP as low priority.

Client1 makes regular HTTP connections to Server1 on tcp-80 retrieving the same HTML object. Timing results are recorded.

Client2 makes an FTP connection to Server1 on the standard FTP port of tcp-21 and remains in the default *active* mode. Client2 requests a file download (t2.jar, size 2209984 bytes) and this is served over a dynamically negotiated connection from the server to the client.

A control test (Payload-2) is performed with same method but the payload match removed from the nmeta configuration.

### Desired Outcome(s)

- 1) Response times for HTTP connections on tcp-80 in test Payload-1 are not materially affected by the FTP congestion (target less than 10% increase in response times)
- 2) Response times for HTTP connections on tcp-80 in test Payload-2 are noticeably affected by the FTP congestion (expect >100% increase in response times)

### Configuration

The following configuration was applied in the nmeta environment for test Payload-1:

```
'PolicyRule 0':
  comment: OpenFlow Protocol Traffic
  match_type: any
  policy_conditions:
    tcp_src: 6633
    tcp_dst: 6633
  actions:
    set_qos_tag: QoS_treatment=system_priority
    set_desc_tag: description="OpenFlow Protocol Traffic"
'PolicyRule 1':
  comment: SSH traffic
  match_type: any
  policy_conditions:
    tcp_src: 22
    tcp_dst: 22
  actions:
    set_qos_tag: QoS_treatment=high_priority
    set_desc_tag: description="High Priority SSH Traffic"
'PolicyRule 2':
  comment: Use Case Payload-1 - Low Priority FTP Traffic
  match_type: any
  policy_conditions:
    payload_type: ftp
  actions:
    set_qos_tag: QoS_treatment=low_priority
    set_desc_tag: description="Low Priority FTP Traffic"
```

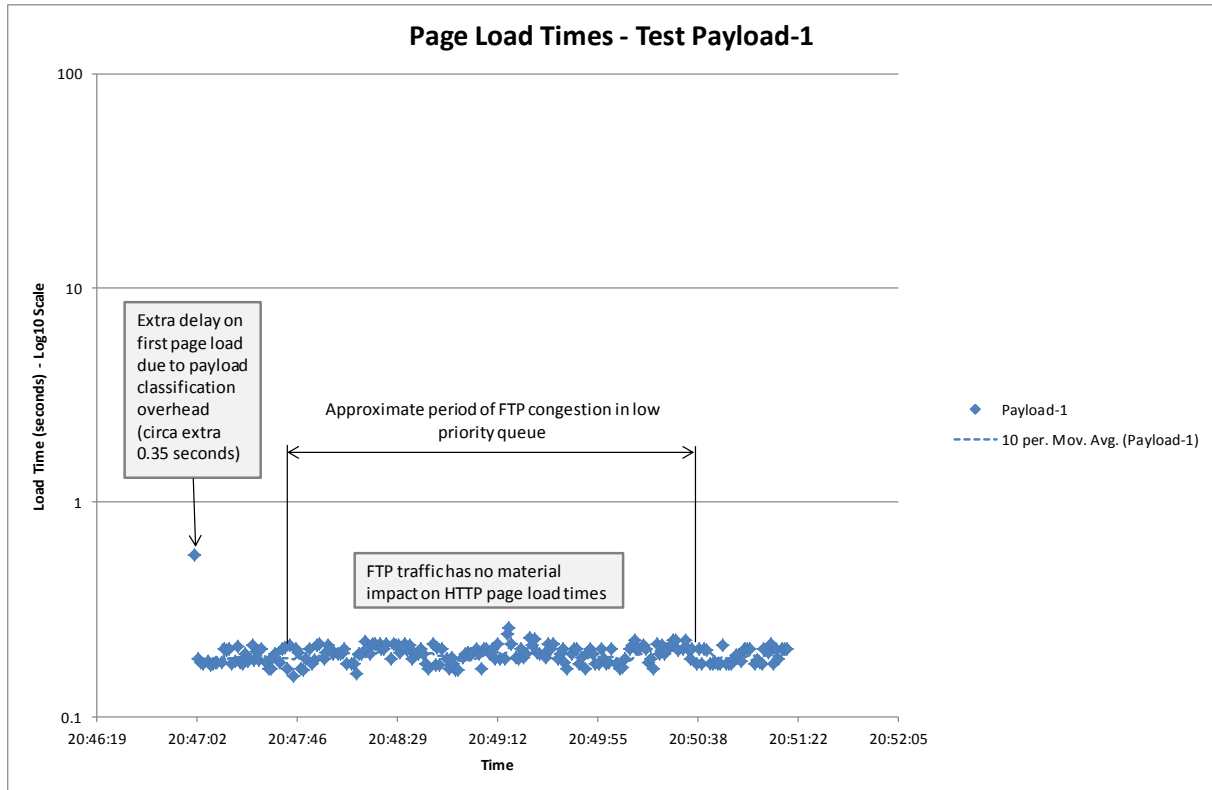
The following configuration was applied in the nmeta environment for test Payload-2:

```
'PolicyRule 0':
  comment: OpenFlow Protocol Traffic
  match_type: any
  policy_conditions:
    tcp_src: 6633
    tcp_dst: 6633
  actions:
    set_qos_tag: QoS_treatment=system_priority
    set_desc_tag: description="OpenFlow Protocol Traffic"
'PolicyRule 1':
  comment: SSH traffic
  match_type: any
  policy_conditions:
    tcp_src: 22
    tcp_dst: 22
  actions:
```



```
set_qos_tag: QoS_treatment=high_priority
set_desc_tag: description="High Priority SSH Traffic"
```

## Results



*Figure 17 - Test Payload-1 in Virtual Lab on code rev 6.5*

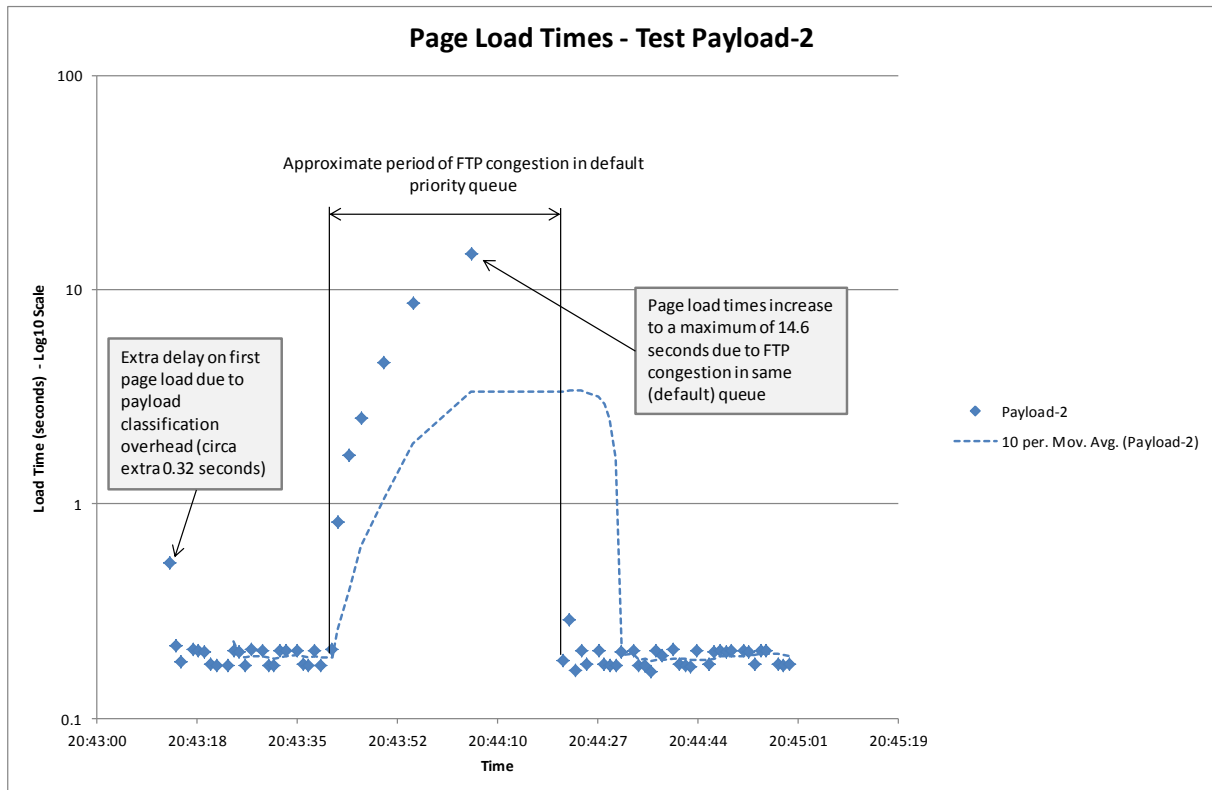


Figure 18 - Test Payload-2 in Virtual Lab on code rev 6.5

## Analysis

In test Payload-1, the FTP payload classifier successfully identified the dynamically negotiated destination TCP port of the flow to be set up from the server to the client and this information was used to classify this new flow into the low priority queue.

Response times for HTTP connections on tcp-80 in test Payload-1 were not materially affected by the FTP link congestion, meeting the expectations of desired outcome 1.

Test Payload-2 was a control test. Response times for HTTP connections on tcp-80 were noticeably affected by the FTP congestion. Maximum response time was an increase by a factor of 69 over the pre-congestion time. This meets the expectations of desired outcome 2.

It is worth noting that the same result can be achieved with static classification, for even though FTP active mode dynamically negotiates a port number for the data transfer, the source port number is always 20. The following snippet of nmeta traffic classification configuration achieves the same result as test Payload-1:

```
'PolicyRule 2':
  comment: FTP Control traffic
  match_type: any
  policy_conditions:
    tcp_src: 21
    tcp_dst: 21
  actions:
    set_qos_tag: QoS_treatment=low_priority
    set_desc_tag: description="Low Priority FTP Control Traffic"
'PolicyRule 3':
  comment: FTP Data traffic
```

```

match_type: any
policy_conditions:
  tcp_src: 20
  tcp_dst: 20
actions:
  set_qos_tag: QoS_treatment=low_priority
  set_desc_tag: description="Low Priority FTP Data Traffic"

```

This serves as an important reminder that simpler, more efficient static classifiers can sometimes fulfil the role of a specific payload classifier.

## Summary of Payload Traffic Classification Findings

Tests Payload-1 and Payload-2 both passed. This demonstrates the SDN capability to run payload inspection on multiple packets in a flow, extracting dynamic information and acting on it.

### 4.7. Test Use Cases Statistical-1 and Statistical-2

#### Goal

Demonstrate that statistical classification can classify based on the statistical profile of a traffic flow, and the results can be used to provide differential QoS treatment for the flow.

#### Method

Traffic classification is configured to treat SSH traffic (tcp-22) as high priority and all other traffic is passed through the statistical\_qos\_bandwidth\_1 statistical classifier.

Client2 makes regular HTTP connections to Server1 on tcp-80 and retrieves the same HTML object. Timing results are recorded.

After establishing a baseline, Iperf from Server1 to Client2 is used to congest the link for a sustained period. Iperf is then terminated and the test runs for a further period to recheck baseline.

A second test (Statistical-2) is run as a control, without the statistical classifier configured.

#### Desired Outcome(s)

The statistical classifier should classify the Iperf traffic into the low\_priority queue, based on its flow behaviour, and thus the Iperf traffic cannot impact the HTTP traffic since they are in different queues. Specific measures are:

- 1) Response times for HTTP connections on tcp-80 in the non-control test are not materially affected by the link congestion (target less than 10% increase in response times).
- 2) Response times in the control test for HTTP connections on tcp-80 are noticeably affected by the congestion (expect >100% increase in response times).

#### Configuration

Statistical-1 Configuration:

```

'PolicyRule 0':
  comment: SSH traffic

```

```

match_type: any
policy_conditions:
  tcp_src: 22
  tcp_dst: 22
actions:
  set_qos_tag: QoS_treatment=high_priority
  set_desc_tag: description="High Priority SSH Traffic"
'PolicyRule 1':
comment: Basic Statistical Classifier
match_type: statistical
policy_conditions:
  statistical_qos_bandwidth_1: on
actions:
  pass_return_tags: true

```

Statistical-2 (Control) Configuration:

```

'PolicyRule 0':
comment: SSH traffic
match_type: any
policy_conditions:
  tcp_src: 22
  tcp_dst: 22
actions:
  set_qos_tag: QoS_treatment=high_priority
  set_desc_tag: description="High Priority SSH Traffic"

```

SSH (tcp-22) traffic is included in the above configurations, but is not used in the testing.

The control configuration does not run the statistical classifier.

## Results

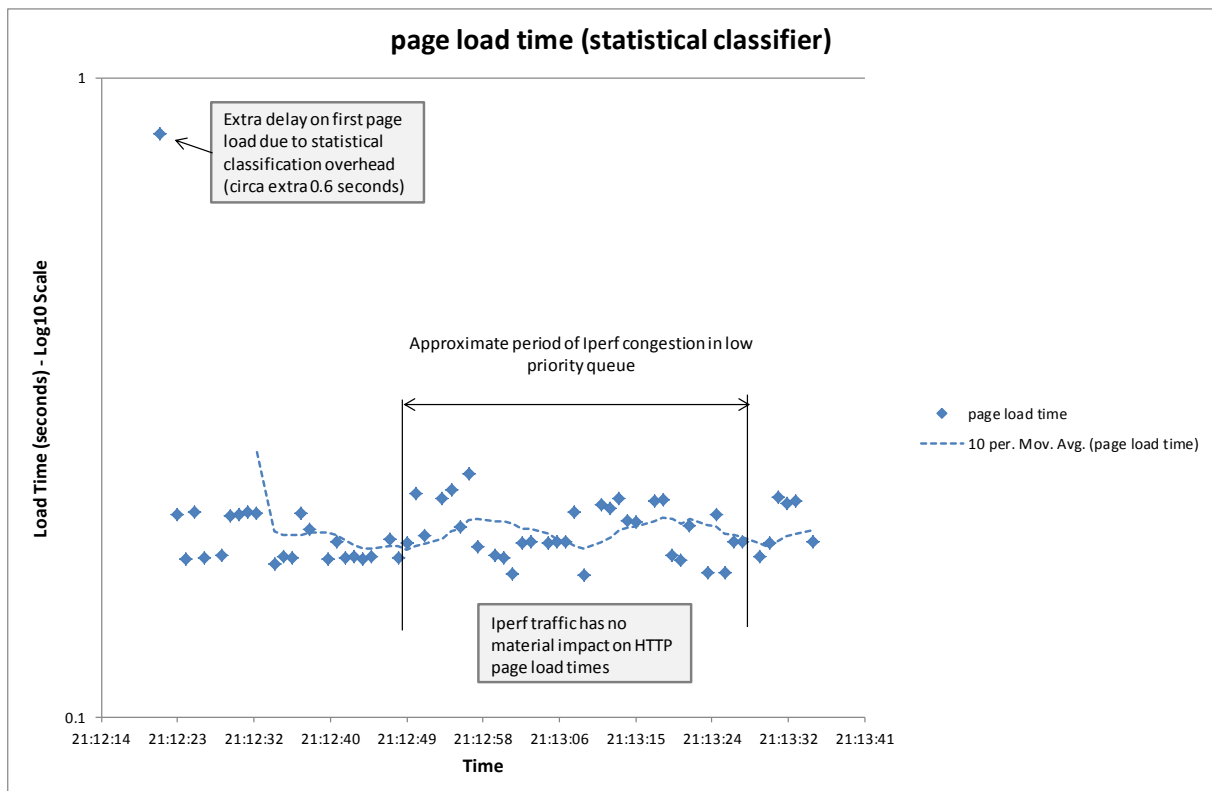


Figure 19 - Test Statistical-1 in Virtual Lab on code rev 6.2

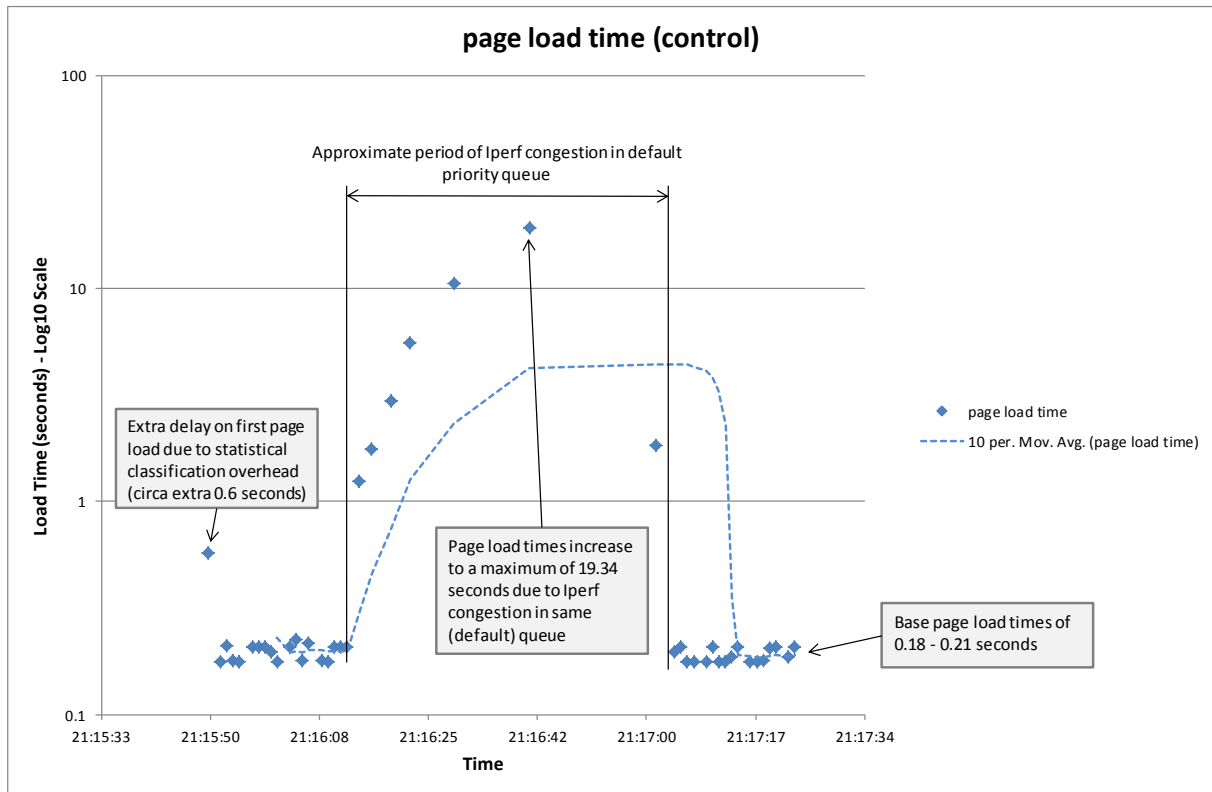


Figure 20 - Test Statistical-2 (control) in Virtual Lab on code rev 6.2

## Analysis

- In test Statistical-1, the Iperf traffic was matched by the statistical classifier and moved from the default\_priority to low\_priority queue. This change of queueing prevented Iperf from congesting the default\_priority queue and thus the HTTP load times were not materially affected by the Iperf congestion, meeting the expectations of desired outcome 1.
- Test Statistical-2 was a control test. Response times for HTTP connections on tcp-80 were noticeably affected by the FTP congestion. Maximum response time was an increase by a factor of 100 over the pre-congestion time. This meets the expectations of desired outcome 2.

Iperf classification is an unlikely use case for an enterprise network; however it does demonstrate the principle that statistical classification can be used to differentiate flows based on their behavioural profile.

Note that the statistical classifier does not reference the flow TCP port numbers, so Iperf could be run on any TCP port number (other than 22 since it is defined as a static classifier) and the results would be the same.

## Summary of Statistical Traffic Classification Findings

Tests Statistical-1 and Statistical-2 both passed. This demonstrates the SDN capability to run statistical analysis on flows with the results used in real-time (i.e. online) to modify network behaviour appropriately.

### 4.8. Evaluation of Hypothesis

Experimental results from the prototype nmeta SDN traffic classification system have shown the hypothesis posed by this project to be true for functional requirements. Four test use cases were evaluated and the results all met the desired outcomes, with the exception of those run in the physical lab. The failure of the physical lab is due to the limitations of the particular switch hardware and does not invalidate the hypothesis. We can thus conclude that the 4 sub-hypotheses are all proven true as results showed that the prototype nmeta traffic classification system built on SDN architecture can accurately classify traffic using static, identity, payload and statistical methods.

Tests Static-1 and Static-2 prove that an SDN traffic classification system can do policy-based static classification, meeting the requirement of *selective determinism*.

Tests Identity-1 and Identity-2 prove that an SDN traffic classification system can be used to apply QoS treatment based on the identity of an endpoint. This meets the requirement for *identity awareness*.

Tests Payload-1 and Payload-2 prove that an SDN traffic classification system can be used to identify the type of traffic contained in a flow and additionally to ascertain details of the set-up of a dynamic flow and apply QoS treatment to this new flow. This meets the requirement for *application awareness*.

Tests Statistical-1 and Statistical-2 prove that an SDN traffic classification system can make statistical classifications that differentiate flows based on their behavioural profile, meeting the requirement for *agility*. The Statistical-1 test also met the requirement for *timeliness* by showing that a flow can be classified before it has had time to ramp up to a point where it is causing congestion.

By proving the 4 sub-hypotheses are all proven true, and that the functional requirements are also true, the main hypothesis that *SDN architecture is a suitable foundation for development of systems that can meet the functional traffic classification requirements of enterprise network operators* is also proven to be true.

### 4.9. Chapter Summary

In this chapter, details the methodology and results of the evaluation of the prototype system have been presented, along with a test of the hypothesis that was shown to prove it to be true. The next and final chapter presents conclusions and other observations.



# Chapter 5

## 5. Conclusion

This project posed the hypothesis that SDN architecture is a suitable foundation for development of systems that can meet the traffic classification requirements of enterprise network operators. This hypothesis was proven by analysis of experimental results from the prototype nmeta system that was specially developed for this project. The fact that a functional prototype system can be built within the timeframe of a COMP489 project demonstrates the rapid innovation potential that SDN opens up to the networking community.

The prototype system produced enriched network metadata as per the 'Flow Enrichment' information in the right of Figure 1, but also extended left into the area of identity. This is a powerful result as it creates a foundation on which innovative applications can be built that mine the identity and flow data for both current and as yet unforeseen benefits.

### 5.1. Other Observations

#### Reticulation of the OpenFlow Protocol

The lab environments in this project featured backhaul of the OpenFlow protocol over the data plane. The remote switch OpenFlow traffic had to cross a simulated WAN link to reach the controller. This configuration was intended to push the limits of what could be achieved with SDN, but was found to be workable in the limited tests that were carried out. The author however would not recommend running OpenFlow protocol over a WAN due to the potential for performance and availability issues.

#### Denial of Service Vulnerabilities

It was found that LLDP packets passed to Ryu with the default packet-in maximum size of 128 bytes are truncated and cause Ryu to halt. This is a problem for both availability and security where it could be used to execute a Denial of Service (DoS) attack.

#### Non-Functional Requirements

Non-functional requirements were put out of scope due to the size of the task.

SDN developers can learn lessons regarding non-functional requirements from the development of monolithic networking, as they are very similar. Key non-functional requirements are availability and security. The author recalls an incident in 1998 when he upgraded the software on routers for an enterprise client. The upgrade was performed and passed testing but the client called up the next day the client complaining that their entire network was broken, with their business substantially impacted. The change was backed out and subsequent investigations showed that the new router software would stop receiving packets on the router LAN interface if a particular sequence of packets was received. This sequence had occurred overnight at all sites and thus none of the LAN interfaces were



operational the next morning resulting in no network connectivity between sites. The lesson from this story is that monolithic networking has matured significantly over the intervening 16 years as it would be highly unusual for a modern commercially-sourced router or switch to stop routing/switching due to a software fault of this nature. Also, this type of fault would now be classified as security vulnerability, since it presents a vector for a Denial of Service (DoS) attack. While it is now highly unusual for a monolithic router/switch to fail unexpectedly, it was found to be a reasonably common occurrence in the SDN lab environments. The author suspects that SDN software still has a long journey ahead to reach the non-functional maturity of monolithic networking, based on experiences gained during this project (i.e. previously mentioned LLDP DoS example), however this journey could be shortened if lessons are learnt from the development of monolithic networking software.

# Bibliography

- [1] M. Hayes, "Quality of Service Classification Mechanisms for IoT," Victoria University, Wellington, New Zealand, Term Paper 2013.
- [2] M. Hayes, B. Ng, and W. Seah, "Traffic Classification in Enterprise Networks within the Era of IoT," no. Submitted to IEEE Communications Magazine.
- [3] A. Dainotti, A. Pescapé, and K.C Claffy, "Issues and future directions in traffic classification," *IEEE Network*, vol. 26, no. 1, pp. 35-40, 2012.
- [4] H. Kim et al., "Internet traffic classification demystified: myths, caveats, and the best practices," in *Proceedings of the 2008 ACM CoNEXT Conference (CoNEXT '08)*, Madrid, Spain, 9-12 December 2008.
- [5] A. Dainotti, W. de Donato, and A. Pescapé, "Tie: A community-oriented traffic classification platform.," in *Traffic Monitoring and Analysis, First International Workshop*, Aachen, Germany, 2009, pp. 64-74.
- [6] J. Kosinski et al., "SLA monitoring and management framework for telecommunication services," in *Networking and Services, 2008. ICNS 2008. Fourth International Conference on*, Gosier, Guadeloupe, 2008, pp. 170-175.
- [7] H. Sundmaeker, P. Guillemin, P. Friess, and S. Woelfflé. (2010) Vision and Challenges for Realising the Internet of Things, March 2010. [Online]. [http://www.internet-of-things-research.eu/pdf/IoT\\_Clusterbook\\_March\\_2010.pdf](http://www.internet-of-things-research.eu/pdf/IoT_Clusterbook_March_2010.pdf)
- [8] M. Roughan, S. Sen, O. Spatscheck, and N. Duffield, "Class of Service Mapping for QoS A Statistical Signature based Approach to IP Traffic Classification," in *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement (IMC)*, Sicily, Italy, 25-27 October 2004.
- [9] B. Nunes, M. Mendonca, X. Nguyen, K. Obraczka, and T. Turletti. (2013, June) HAL - Inria. [Online]. [http://hal.inria.fr/docs/00/93/29/82/PDF/hal\\_final.pdf](http://hal.inria.fr/docs/00/93/29/82/PDF/hal_final.pdf)
- [10] T. En-Najjary and G. Urvoy-Keller, "A first look at traffic classification in enterprise networks," in *In Proceedings of the 6th International Wireless Communications and Mobile Computing Conference*, Caen, France, 2010, pp. 764-768.
- [11] R. Pang et al., "A first look at modern enterprise traffic," in *Proceedings of the 5th ACM SIGCOMM conference on Internet Measurement*, Berkeley, CA, USA, 2005, pp. 2-2.
- [12] Canalys. (2014, October) Network infrastructure, Worldwide, value (\$) by technology, Q2 2014 and Q2 2013. [Online]. <http://www.canalys.com/chart/index.html#display-314>
- [13] B. Park, Y. Won, M. Kim, and J. Hong, "Towards Automated Application Signature Generation for Traffic Identification," in *Network Operations and Management Symposium, 2008. NOMS 2008. IEEE*, Salvador, Bahia, Brazil, 2008, pp. 160-167.
- [14] G. Aceto, A. Dainotti, W. De Donato, and A Pescapé, "PortLoad: taking the best of two worlds in traffic classification," in *INFOCOM IEEE Conference on Computer*

*Communications Workshops*, San Diego, USA, 15-19 March 2010, pp. 1-5.

- [15] L. Bernaille, R. Teixeira, I. Akodkenou, A. Soule, and K. Salamatian, "Traffic Classification On The Fly," *ACM SIGCOMM Computer Communication Review*, vol. 36, no. 2, pp. 23-26, April 2006.
- [16] R. Alshammari and A. Zincir-Heywood, "A flow based approach for ssh traffic detection," in *Systems, Man and Cybernetics, 2007. ISIC. IEEE International Conference on*, Montreal, Canada, October 2007, pp. 296-301.
- [17] G. Xie, M. Iliofotou, R. Keralapura, M. Faloutsos, and A. Nucci, "Subflow: Towards practical flow-level traffic classification," in *INFOCOM, 2012 Proceedings IEEE*, Orlando, FL, USA, 25-30 March 2012, pp. 2541-2545.
- [18] S. Li, E. Murray, and Y. Luo, "Programmable Network Traffic Classification with OpenFlow Extensions," in *Network Innovation through OpenFlow and SDN : Principles and Design*, F. Hu, Ed. Boca Raton, FL, USA: CRC Press, 2014, ch. 13, pp. 269-299.
- [19] J. Khalife, A. Hajjar, and J Diaz-Verdejo, "A multilevel taxonomy and requirements for an optimal traffic-classification model," *International Journal of Network Management*, vol. 24, no. 2, pp. 101-120, January 2014.
- [20] K. Nichols, S. Blake, F. Baker, and D Black. (1998, December) Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers. [Online]. <http://tools.ietf.org/pdf/rfc2474.pdf>
- [21] Open Networking Foundation. (2013, October) OpenFlow. [Online]. <https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf>
- [22] Ryu Community. (2014, September) Ryu SDN Framework. [Online]. <http://osrg.github.io/ryu/>
- [23] Python Software Foundation. (2014, September) Python.org. [Online]. <https://www.python.org/>
- [24] Python Software Foundation. (2014, September) Private Variables and Class-local References. [Online]. <https://docs.python.org/2/tutorial/classes.html#tut-private>
- [25] G. van Rossum, B. Warsaw, and N. Coghlan. (2014, September) PEP 8 -- Style Guide for Python Code. [Online]. <http://legacy.python.org/dev/peps/pep-0008/>
- [26] Python Software Foundation. (2014, October) Data Structures. [Online]. <https://docs.python.org/2/tutorial/datastructures.html#dictionaries>
- [27] D. Borman, B. Braden, V. Jacobson, and R. Scheffenegger. (September, 2014) TCP Extensions for High Performance. [Online]. <http://tools.ietf.org/pdf/rfc7323.pdf>
- [28] C. Evans. The Official YAML Web Site. [Online]. <http://www.yaml.org/>
- [29] Ryu Community. (2014, October) Ryu application API. [Online]. [http://ryu.readthedocs.org/en/latest/ryu\\_app\\_api.html](http://ryu.readthedocs.org/en/latest/ryu_app_api.html)

- [30] M. Carbone and L. Rizzo, "Dummynet Revisited," *ACM SIGCOMM Computer Communication Review*, vol. Volume 40, no. Issue 2, pp. 12-20, April 2010.
- [31] J Dugan and M Kutzko. (2014, September) Iperf. [Online]. <http://sourceforge.net/projects/iperf/>
- [32] R. Fielding and J. Reschke. (2014, June) RFC 7230: Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. [Online]. <http://tools.ietf.org/pdf/rfc7230.pdf>
- [33] K. Reitz. (2014, September) Requests: HTTP for Humans. [Online]. <http://docs.python-requests.org/en/latest/>
- [34] H. Yin et al. "SDNi: A Message Exchange Protocol for Software Defined Networks (SDNS) across Multiple Domains", IETF Internet Drafts, 29 December 2012. [Online]. <http://tools.ietf.org/pdf/draft-yin-sdn-sdni-00.pdf>

## Appendix A - Test Details

This appendix contains configuration details for the set-up and running of the tests carried out for this project in the physical lab environment.

### Installation

#### Server (PC1) Configuration

Install Iperf:

```
sudo apt-get install iperf
```

Set up a web server. Copy the latest version of *websvr.py* (a Python program written as part of this project) into the home directory. Create a subdirectory called *static* and copy into it the file *index.html*.

Install the web.py library:

```
sudo easy_install web.py
```

#### Client (PC3) Installation and Configuration

Install PIP:

```
sudo apt-get install python-pip
```

Install python requests module:

```
pip install requests
```

Install Iperf:

```
sudo apt-get install iperf
```

Create HTTP Request Test Scripts. Copy the latest version of *htest.py* (a Python program written as part of this project) into the home directory.

#### Set up QoS Queues on Switches

Set up four egress queues on ports that face the WAN simulator:

q0 = used for OpenFlow traffic, max rate 300Kbps

q1 = default, max rate 500Kbps

q2 = high priority, max rate 1Mbps

q3 = low\_priority, max rate 100Kbps

All queues are configured with a ceiling of 1.8Mbps, which allows bursting over the queue reserved bandwidth when other bandwidth is available.

Physical Lab Central Pica8 Open vSwitch (Switch 2):

```

ovs-vsctl clear Port ge-1/1/3 qos
ovs-vsctl --all destroy qos
ovs-vsctl -- set Port ge-1/1/3 qos=@newqos \
-- --id=@newqos create QoS type=PRONTO_STRICT queues=0=@q0,1=@q1,2=@q2,3=@q3 \
-- --id=@q0 create Queue other-config:min-rate=300000 other-config:max-rate=300000 other-config:ceil=1800000 \
-- --id=@q1 create Queue other-config:min-rate=500000 other-config:max-rate=500000 other-config:ceil=1800000 \
-- --id=@q2 create Queue other-config:min-rate=1000000 other-config:max-rate=1000000 other-config:ceil=1800000 \
-- --id=@q3 create Queue other-config:min-rate=100000 other-config:max-rate=100000 other-config:ceil=180000

```

Physical Lab Remote Pica8 Open vSwitch (Switch 1):

```

ovs-vsctl clear Port ge-1/1/1 qos
ovs-vsctl --all destroy qos
ovs-vsctl -- set Port ge-1/1/1 qos=@newqos \
-- --id=@newqos create QoS type=PRONTO_STRICT queues=0=@q0,1=@q1,2=@q2,3=@q3 \
-- --id=@q0 create Queue other-config:min-rate=300000 other-config:max-rate=300000 other-config:ceil=1800000 \
-- --id=@q1 create Queue other-config:min-rate=500000 other-config:max-rate=500000 other-config:ceil=1800000 \
-- --id=@q2 create Queue other-config:min-rate=1000000 other-config:max-rate=1000000 other-config:ceil=1800000 \
-- --id=@q3 create Queue other-config:min-rate=100000 other-config:max-rate=100000 other-config:ceil=180000

```

Virtual Lab (WAN3) Central Switch:

```

sudo ovs-vsctl clear Port eth2 qos
sudo ovs-vsctl --all destroy qos
sudo ovs-vsctl -- set Port eth2 qos=@newqos \
-- --id=@newqos create QoS type=linux-htb queues=0=@q0,1=@q1,2=@q2,3=@q3 \
-- --id=@q0 create Queue other-config:min-rate=300000 other-config:max-rate=300000 other-config:ceil=1800000 \
-- --id=@q1 create Queue other-config:min-rate=500000 other-config:max-rate=500000 other-config:ceil=1800000 \
-- --id=@q2 create Queue other-config:min-rate=1000000 other-config:max-rate=1000000 other-config:ceil=1800000 \
-- --id=@q3 create Queue other-config:min-rate=100000 other-config:max-rate=100000 other-config:ceil=180000

```

Virtual Lab (WAN3) Remote Switch:

```

sudo ovs-vsctl clear Port eth1 qos
sudo ovs-vsctl --all destroy qos
sudo ovs-vsctl -- set Port eth1 qos=@newqos \
-- --id=@newqos create QoS type=linux-htb queues=0=@q0,1=@q1,2=@q2,3=@q3 \
-- --id=@q0 create Queue other-config:min-rate=300000 other-config:max-rate=300000 other-config:ceil=1800000 \
-- --id=@q1 create Queue other-config:min-rate=500000 other-config:max-rate=500000 other-config:ceil=1800000 \
-- --id=@q2 create Queue other-config:min-rate=1000000 other-config:max-rate=1000000 other-config:ceil=1800000 \
-- --id=@q3 create Queue other-config:min-rate=100000 other-config:max-rate=100000 other-config:ceil=180000

```

Configuration can be checked with the following commands:

```

ovs-vsctl list port <interface>

```

```

root@XorPlus# ovs-vsctl list port ge-1/1/3
_uuid                : 857a9066-b9fa-494d-810b-2381b3b6c3a5
bond_downdelay       : 0
bond_fake_iface      : false
bond_mode            : []
bond_updelay         : 0
external_ids         : {}
fake_bridge          : false
interfaces           : [7447dab3-5442-42a1-ac37-42edbe78fc07]
lacp                 : []
mac                  : []
name                 : "ge-1/1/3"
other_config         : {}
qos                  : 155f279d-88ab-4d67-a852-dfa562bcd9d6
statistics           : {}
status              : {}
tag                  : []
trunks               : []
vlan_mode            : []
root@XorPlus#

```

The reference in output above is displayed in the output below along with references for the individual queues:

**ovs-vsctl list qos**

```

root@XorPlus# ovs-vsctl list qos
_uuid                : 155f279d-88ab-4d67-a852-dfa562bcd9d6 Main reference
external_ids         : {}
other_config         : {}
queues               : {0=ae40cb37-97f2-4982-9d34-22cf7e49aab9, 1=4fa76adf-a898-4cd3-9a1c-95c427c8e830, 2=ee0d26ee-bbda-4fd2-9b52-26e8a8e6b7cf, 3=34bc3ad7-d446-4318-b204-2b4bc3d4add6}
type                 : PRONTO_STRICT

```

Check the individual queue configurations with the following command:

**ovs-vsctl list queue**

```

root@XorPlus# ovs-vsctl list queue
_uuid                : ee0d26ee-bbda-4fd2-9b52-26e8a8e6b7cf
dscp                 : []
external_ids         : {}
other_config         : {ceil="1800000", max-rate="1000000", min-rate="1000000"}

_uuid                : ae40cb37-97f2-4982-9d34-22cf7e49aab9
dscp                 : []
external_ids         : {}
other_config         : {ceil="1800000", max-rate="300000", min-rate="300000"}

_uuid                : 4fa76adf-a898-4cd3-9a1c-95c427c8e830
dscp                 : []
external_ids         : {}
other_config         : {ceil="1800000", max-rate="500000", min-rate="500000"}

_uuid                : 34bc3ad7-d446-4818-b204-2b4bc3d4add6
dscp                 : []
external_ids         : {}
other_config         : {ceil="1800000", max-rate="100000", min-rate="100000"}

```

Check queue statistics:

**ovs-ofctl queue-stats br0**

This command does not work on Pica8 switches due to a bug, so output shown from a switch in the WAN3 virtual environment:

```

bob@COMP489-OpenvSwitch:~$ sudo ovs-ofctl queue-stats br0
OFPST_QUEUE reply (xid=0x2): 4 queues
port 1 queue 0: bytes=382855, pkts=3373, errors=0, duration=?
port 1 queue 1: bytes=6641839, pkts=5760, errors=0, duration=?
port 1 queue 2: bytes=248069, pkts=1854, errors=0, duration=?
port 1 queue 3: bytes=0, pkts=0, errors=0, duration=?

```

## 5.2. Running Tests

### Test Use Case Static-1 and Static-2

On server (PC1), start web servers in separate terminal sessions:

```

sudo python webservr.py 80
(alias h80)

sudo python webservr.py 1234
(alias h1234)

```

On server (PC1), check that *qos\_policy.yaml* and *tc\_policy.yaml* are set appropriately

On server (PC1), start nmeta:

```

cd ryu
PYTHONPATH=. ./bin/ryu-manager ryu/app/nmeta/nmeta.py
(alias nm)

```

On client (PC3), start Iperf server:

```

iperf -s -i 1
(alias ipf)

```

On client (PC3), start the HTTP on TCP port 80 Python script:

```

python htest.py http://10.255.255.1:80/80
(alias ht80)

```

On client (PC3), start the HTTP on TCP port 1234 Python script:

```

python htest.py http://10.255.255.1:1234/1234
(alias ht1234)

```

On server (PC1), after waiting approximately 15 seconds, start Iperf:

```

iperf -c 10.255.254.100 -t 30
(alias ipf1)

```

(it will run for 30 seconds)

Wait an additional 15 seconds after Iperf completes then stop both htest.py instances and record their results.

### Test Use Cases Identity-1 and Identity-2

On server (VM1), start web server:

```

sudo python webservr.py 80
(alias h80)

```

On server (VM1), check that *qos\_policy.yaml* and *tc\_policy.yaml* are set appropriately

On server (VM1), start nmeta:

```

cd ryu
PYTHONPATH=. ./bin/ryu-manager ryu/app/nmeta/nmeta.py

```



(alias nm)

On clients (VM5 & VM6), start Iperf server:

```
iperf -s -i 1  
(alias ipf)
```

On client 1 (VM5), start the HTTP on TCP port 80 Python script:

```
python htest.py http://192.168.57.40:80/80  
(alias ht80)
```

On client 2 (VM6), start the HTTP on TCP port 80 Python script:

```
python htest.py http://192.168.57.40:80/80  
(alias ht80)
```

On server (VM1), after waiting approximately 15 seconds, start two Iperf sessions concurrently:

```
iperf -c 192.168.56.11 -t 30  
(alias ipf1)
```

In separate window:

```
iperf -c 192.168.56.12 -t 30  
(alias ipf2)
```

(they will run for approximately 30 seconds)

Wait an additional 15 seconds after the Iperf sessions complete then stop both htest.py instances and record their results.

## Test Use Cases Payload-1 and Payload-2

On server (VM1), check that *qos\_policy.yaml* and *tc\_policy.yaml* are set appropriately

On server (VM1), start nmeta:

```
cd ryu  
PYTHONPATH=. ./bin/ryu-manager ryu/app/nmeta/nmeta.py  
(alias nm)
```

On server (VM1), start web server:

```
sudo python webserv.py 80  
(alias h80)
```

On client1 (VM5), start the HTTP on TCP port 80 Python script:

```
python htest.py http://192.168.57.40:80/80  
(alias ht80)
```

On client2 (VM5), start an FTP to the server and retrieve the object:

```
Error! Hyperlink reference not valid.  
(alias ftp1)
```

```
<log in>
```

```
get t2.jar
```

## Test Use Case Statistical-1

### Pretests:

On server (VM1), check that *qos\_policy.yaml* and *tc\_policy.yaml* are set appropriately

On server (VM1), start nmeta:

```
cd ryu
PYTHONPATH=. ./bin/ryu-manager ryu/app/nmeta/nmeta.py
(alias nm)
```

On server (VM1), start web server:

```
sudo python webservr.py 80
(alias h80)
```

On server (VM1), start an Iperf server:

```
iperf -s -i 1
(alias ipfs)
```

On client (VM6), carry out the following tests and record the statistical analysis figures reported by nmeta:

### Iperf:

```
iperf -c 192.168.57.40 -t 30
(alias ipf)
```

### HTTP:

```
python htest.py http://192.168.57.40:80/80
(alias ht80)
```

### SSH (Interactive):

```
ssh bob@192.168.57.40
```

### SCP:

```
scp Downloads/t1.jpg bob@192.168.57.40:t1.jpg
(alias scp1)
```

Set the statistical classifier to use different values for the maximum packets to accumulate in a flow before making a classification (variable `_max_packets`)

## Main Tests

On server (VM1), check that *qos\_policy.yaml* and *tc\_policy.yaml* are set appropriately

On server (VM1), start nmeta:

```
cd ryu
PYTHONPATH=. ./bin/ryu-manager ryu/app/nmeta/nmeta.py
(alias nm)
```

On server (VM1), start web server:

```
sudo python webservr.py 80
(alias h80)
```

On client (VM6), start Iperf server:

```
iperf -s -i 1  
(alias ipf)
```

On client (VM6), start the HTTP on TCP port 80 Python script:

```
python htest.py http://192.168.57.40:80/80  
(alias ht80)
```

On server (VM1), after waiting approximately 15 seconds, start Iperf session:

```
iperf -c 192.168.56.12 -t 30  
(alias ipf2)
```

(will run for approximately 30 seconds)

Wait an additional 15 seconds after the Iperf session completes then stop htest.py record results.

Repeat test for control configuration of *tc\_policy.yaml*

## Appendix B - WAN3 Build Instructions

These instructions are included to assist the experimenter with building the WAN3 virtualised environment. They should hopefully save a lot of time that was needed to figure out obtuse features. See Figure 8 for a logical diagram of the environment. Note that these instructions are untested as they are based on build notes and have not been used for a fresh install, so some things may not work as advertised...

### Pre-Requisites

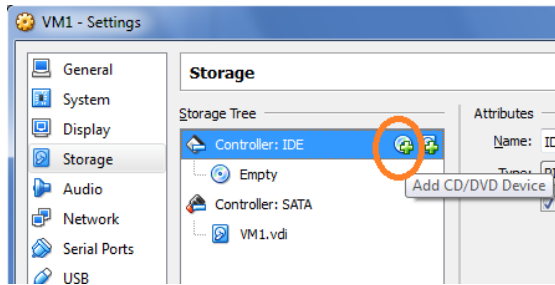
- Oracle VirtualBox hypervisor version 4.3.10 running on Microsoft Windows 7. Note: may work on other versions of VirtualBox and Host OS, but not tested.
- Host PC must have sufficient RAM (test PC had 8GB RAM)

### VM1 – Server / Controller

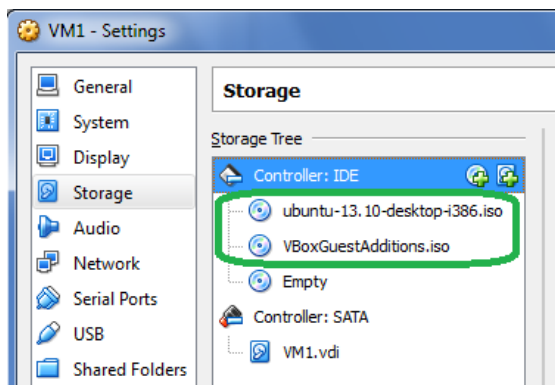
Download Ubuntu 13.10 ISO (32-bit desktop). Note that other versions of Ubuntu are probably fine to use, up to you if you prefer another one.

Create a new Ubuntu guest with 1024MB of RAM and 12GB of storage.

Go into the guest settings to configure it to boot off the ISO. Under *Storage*, click on the *Controller: IDE* row and then click on the *Add CD/DVD Device* Icon, "Choose disk" and browse to the ISO:

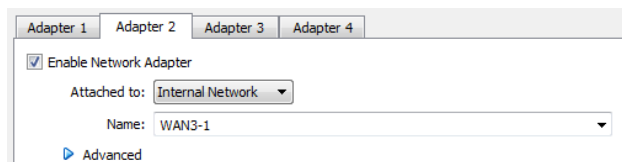


Do the same process to add the ISO for the Guest Additions. On Windows it is located in C:\Program Files\Oracle\VirtualBox\VBBoxGuestAdditions.iso

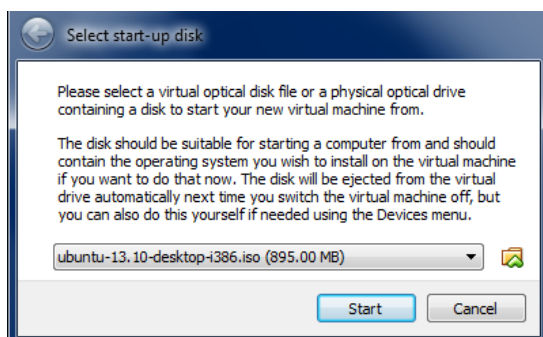


As above, there should now be two ISO files associated.

Leave Adapter 1 as per defaults to allow NAT access to the Internet (required for downloading software packages from the Internet). Configure Adapter 2 as per screenshot below to connect to Internal Network "WAN3-1":



Start the VM and install as per the defaults. Note that VirtualBox may ask you to confirm which ISO to boot from:



## Install VirtualBox Additions

Once built, start a terminal window (CTRL+ALT+T) and install the VirtualBox additions for improved host-guest integration:

```
cd /media
```

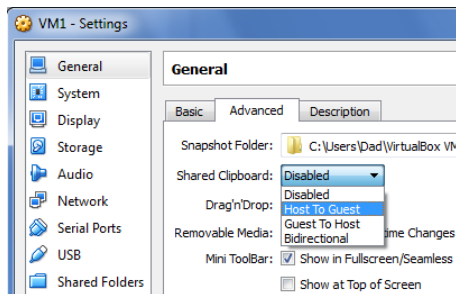
Look for the appropriate subdirectories that contain the correct additions version. Example:

```
bob@bob-WAN3-Host-2: /media/bob/VBOXADDITIONS_4.3.10_93012
bob@bob-WAN3-Host-2:~$ cd /media
bob@bob-WAN3-Host-2:/media$ ls
bob
bob@bob-WAN3-Host-2:/media$ cd bob
bob@bob-WAN3-Host-2:/media/bob$ ls
VBOXADDITIONS_4.3.10_93012
bob@bob-WAN3-Host-2:/media/bob$ cd VBOXADDITIONS_4.3.10_93012/
bob@bob-WAN3-Host-2:/media/bob/VBOXADDITIONS_4.3.10_93012$ ls
32Bit      runasroot.sh
64Bit      VBoxLinuxAdditions.run
AUTORUN.INF VBoxSolarisAdditions.pkg
autorun.sh VBoxWindowsAdditions-amd64.exe
cert       VBoxWindowsAdditions.exe
OS2        VBoxWindowsAdditions-x86.exe
```

Run the additions:

```
sudo ./VBoxLinuxAdditions.run
```

It will probably be necessary to restart the guest to get the additions running. You may want to change the copy/paste settings in VirtualBox to allow pasting to the guest:



## Configure Networking

Add the following to /etc/network/interfaces:

```
auto eth1
iface eth1 inet static
address 192.168.57.40
netmask 255.255.255.0
#
up route add -net 192.168.56.0/24 gw 192.168.57.1 dev eth1
```

Restart networking:

```
sudo /etc/init.d/networking restart
```

## Install Ryu

Install git:

```
sudo apt-get install git
```

Clone Ryu:

```
git clone git://github.com/osrg/ryu.git
```

Python stuff:

```
sudo apt-get install python-setuptools
sudo apt-get install python-pip
sudo apt-get install libxml2-dev
sudo apt-get install libxslt-dev
# Fix python-six (compatibility library) version issue:
sudo pip install six --upgrade
sudo apt-get install python-dev
```

Install YAML ("YAML Ain't Markup Language") for parsing config and policy files:

```
sudo apt-get install python-yaml
```

Install Ryu:

```
cd ryu
sudo python ./setup.py install
```

Run Ryu (simple switch) to check that it works:

```
cd ryu
PYTHONPATH=. ./bin/ryu-manager --verbose ryu/app/simple_switch.py
```

**Note:** Ryu version can subsequently be upgraded as follows if required:

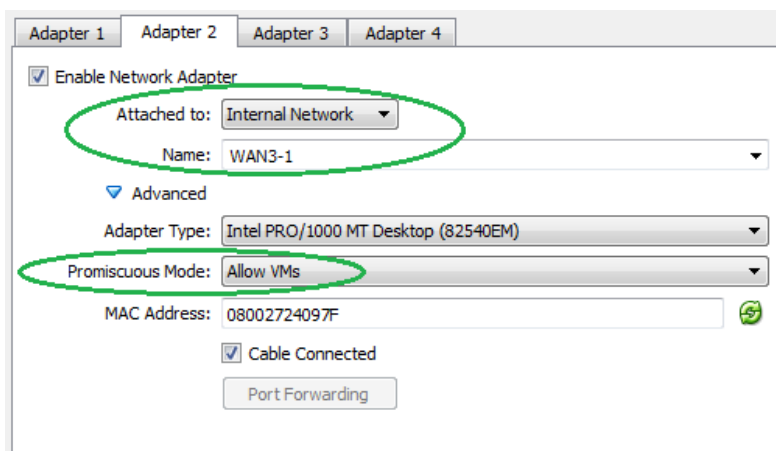
```
cd ryu
sudo git pull
```

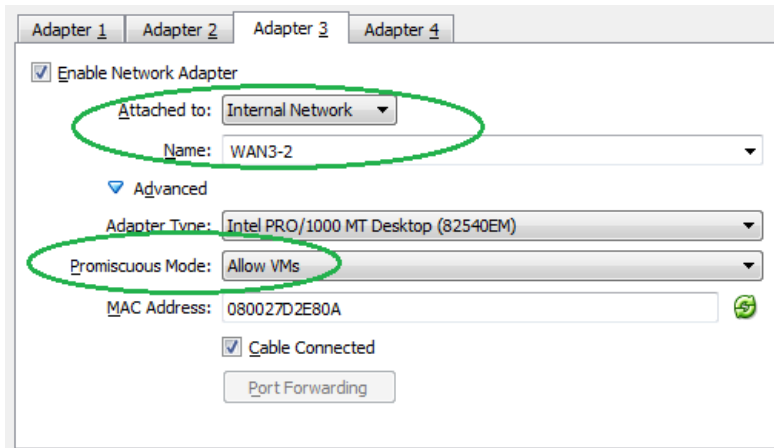
## VM2 – Central Open vSwitch

Download Ubuntu 14.04 ISO (32-bit desktop). Note that version 14.04 is a minimum requirement due to additional of in-tree kernel support for Open vSwitch from that version.

Create a new Ubuntu guest with 512MB of RAM and 8GB of storage as per instructions for VM1 including the association of ISO images.

Leave Adapter 1 as per defaults to allow NAT access to the Internet. Configure Adapter 2 and 3 as per screenshots below:





Note the use of named Internal Networks and selection of *Promiscuous Mode Allow VMs*. This mode is required to allow the guest to function as a switch. MAC addresses can be left as per their default values as long as they are unique within the environment.

Start the VM and install as per instructions for VM1 including guest additions.

### Install Open vSwitch:

Install openvswitch-common and openvswitch-switch (both in same package):

```
sudo apt-get install openvswitch-switch
```

Check that it is running:

```
sudo ovs-vsctl show
```

Set up bridge 'br0':

```
sudo ovs-vsctl add-br br0
```

Add physical interfaces:

```
sudo ovs-vsctl add-port br0 eth1
sudo ovs-vsctl add-port br0 eth2
```

### Configure Networking

Add the following to /etc/network/interfaces:

```
auto br0
iface br0 inet static
address 192.168.57.3
network 192.168.57.0
netmask 255.255.255.0
broadcast 192.168.57.255
gateway 192.168.57.1
#
up route add -net 192.168.56.0/24 gw 192.168.57.1 dev br0
```

Now do a full restart of the guest.

## Configure Open vSwitch

Check connectivity to the OpenFlow controller:

```
ping 192.168.57.40
```

Start OpenFlow Controller (on VM1)

```
cd ryu
PYTHONPATH=. ./bin/ryu-manager --verbose ryu/app/simple_switch.py
```

Set Open vSwitch to contact Controller (on VM2)

```
sudo ovs-vsctl set-controller br0 tcp:192.168.57.40:6633
```

Check Open vSwitch connectivity to OpenFlow Controller:

```
sudo ovs-vsctl show
```

Here is output showing successful connection with the controller:

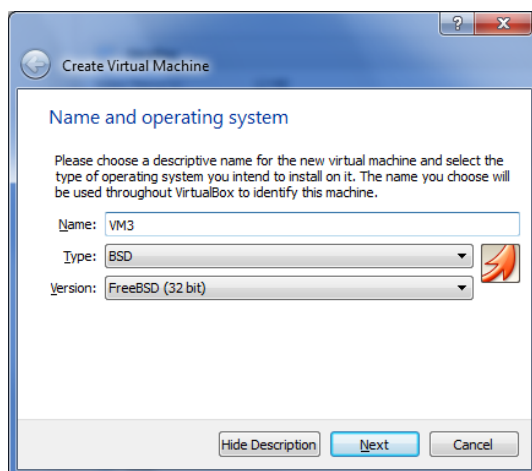
```
bob@bob-WAN3OVSCentral:~$ sudo ovs-vsctl show
0ef20c12-ec97-477f-9d94-abf90954662b
Bridge "br0"
  Controller "tcp:192.168.57.40:6633"
    is connected: true
  Port "eth1"
    Interface "eth1"
  Port "br0"
    Interface "br0"
    type: internal
  Port "eth2"
    Interface "eth2"
ovs_version: "2.0.1"
```

## VM3 - WAN Simulation

### Build Guest

Download FreeBSD 10.0 ISO

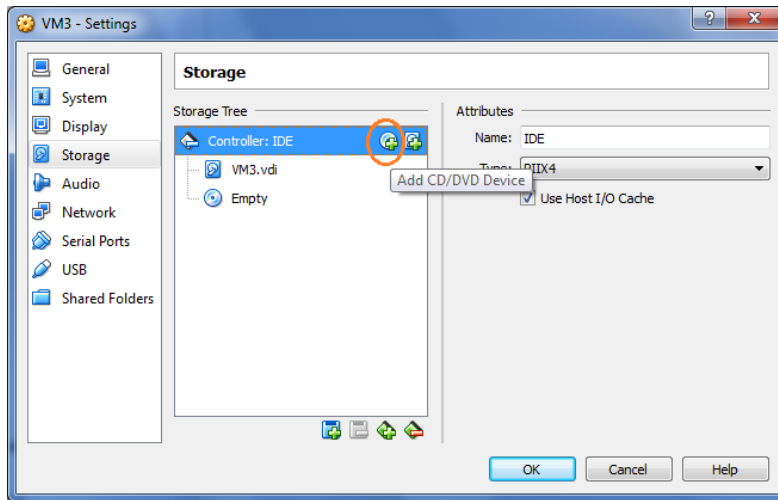
Create a new BSD guest:



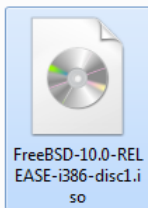
Chose 256MB of RAM and the default storage options and a new VM will be created.



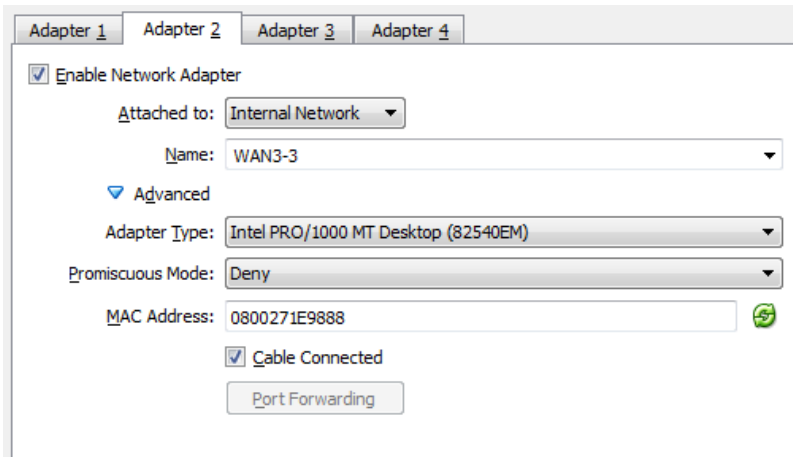
Go into the guest settings to configure it to boot off the ISO. Under *Storage*, click on the *Controller: IDE* row and then click on the *Add CD/DVD Device* Icon (circled in orange in screenshot below):

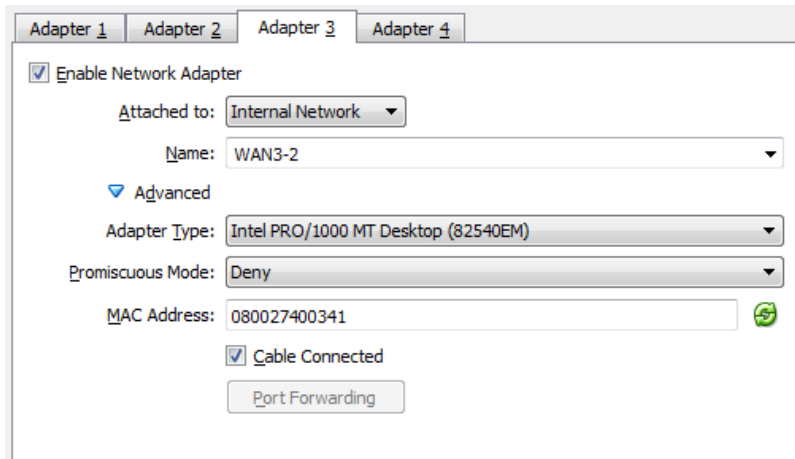


Browse to the location of the ISO:



Leave Adapter 1 as per defaults to allow NAT access to the Internet. Configure Adapter 2 and 3 as per screenshots below:





Note the use of named Internal Networks. MAC addresses can be left as per their default values as long as they are unique within the environment.

Start the VM and install as per the defaults. Once built, shut the VM down and remove the ISO from the *Storage* setting otherwise it will try and rebuild when started.

## Configure Networking

Start the VM and configure networking by editing */etc/rc.conf* and adding these lines:

```
ifconfig_em1="inet 192.168.56.1 netmask 255.255.255.0"
ifconfig_em2="inet 192.168.57.1 netmask 255.255.255.0"
#
# Enable IP Routing:
gateway_enable="YES"
#
# Enable IPFW (used for Dummynet):
firewall_enable="YES"
firewall_type="open"
firewall_script="/etc/ipfw.rules"
```

Enable Kernel Support for Dummynet by modifying the */boot/loader.conf* file:

```
dummynet_load="YES"
```

Configure Dummynet by creating a new file */etc/ipfw.rules*

```
ipfw -q flush
ipfw add pipe 1 ip from any to any
ipfw pipe 1 config delay 10ms bw 2Mbit/s plr 0
```

The bandwidth can be any of bit/s, Kbit/s, Mbits/s, Byte/s, KByte/s, MByte/s. A bandwidth of zero results in no bandwidth limitation

Note that the rule is applied 4 times as the request packet is received and sent out and the reply is received and sent out, i.e. 10ms configured is 40ms RTT.

Make config live:

```
service ipfw restart
```

Check with:

```
ipfw pipe 1 show
```

## VM4 – Remote Open vSwitch

Build as per VM2, but with the following differences:

- Adapter2 connects to network WAN3-3
- Adapter3 connects to network WAN3-4
- Adapter4 connects to network WAN3-5
- The config added to */etc/network/interfaces* is:

```
auto br0
iface br0 inet static
address 192.168.56.3
network 192.168.56.0
netmask 255.255.255.0
broadcast 192.168.56.255
gateway 192.168.56.1
#
up route add -net 192.168.57.0/24 gw 192.168.56.1 dev br0
```

## VM5 - Client 1

Build as per VM1, but with the following differences:

- Adapter2 connects to network WAN3-4
- The config added to */etc/network/interfaces* is:

```
auto eth1
iface eth1 inet static
address 192.168.56.11
netmask 255.255.255.0
#
up route add -net 192.168.57.0/24 gw 192.168.56.1 dev eth1
```

- The */etc/hostname* file was set as follows:

```
pc1
```

- The entry in */etc/hosts* file for 127.0.1.1 was updated as follows:

```
127.0.1.1 pc1.dev.example.com pc1
```

## VM6 - Client 2

Build as per VM1, but with the following differences:

- Adapter2 connects to network WAN3-5
- The config added to */etc/network/interfaces* is:

```
auto eth1
iface eth1 inet static
address 192.168.56.12
netmask 255.255.255.0
#
up route add -net 192.168.57.0/24 gw 192.168.56.1 dev eth1
```

- The `/etc/hostname` file was set as follows:

`pc2`

- The entry in `/etc/hosts` file for 127.0.1.1 was updated as follows:

`127.0.1.1 pc2.audit.example.com pc2`

## Appendix C - Troubleshooting

The following commands may come in useful for troubleshooting and diagnostics.

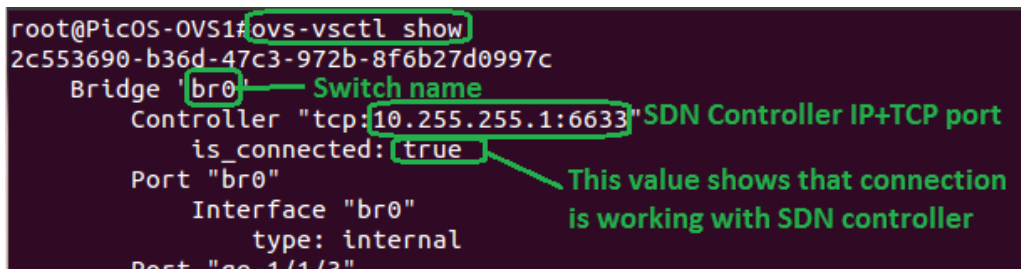
### Open vSwitch Troubleshooting

**Note:** may need to precede commands with 'sudo' on software implementations of Open vSwitch.

#### General Switch Commands

Show the general state of the Open vSwitch:

`ovs-vsctl show`



```

root@PicOS-OVS1# ovs-vsctl show
2c553690-b36d-47c3-972b-8f6b27d0997c
  Bridge "br0"
    Controller "tcp:10.255.255.1:6633"
    is_connected: true
    Port "br0"
      Interface "br0"
        type: internal
        Port "br0"
  
```

Annotations in the image:

- A green box highlights `ovs-vsctl show` in the command line.
- A green box highlights `br0` in the output, with a line pointing to the text "Switch name".
- A green box highlights `tcp:10.255.255.1:6633` in the output, with a line pointing to the text "SDN Controller IP+TCP port".
- A green box highlights `true` in the output, with a line pointing to the text "This value shows that connection is working with SDN controller".

Note that the switch name is used in some of the following commands, if the name of the switch is not 'br0' then replace with the appropriate name.

#### OpenFlow Commands

Show OpenFlow config:

`ovs-ofctl show br0`

Display OpenFlow flows:

`ovs-ofctl dump-flows br0`

Snoop the OpenFlow messages:

`ovs-ofctl snoop br0`

View switch port statistics:

`ovs-ofctl dump-ports br0`

#### Change OpenFlow Version

Change OpenFlow Version to just v1.0:

`ovs-vsctl set bridge br0 protocols=OpenFlow10`

Change OpenFlow Version to 1.0, 1.2 & 1.3:

```
ovs-vsctl set bridge br0 protocols=OpenFlow10,OpenFlow12,OpenFlow13
```

## Check Queueing

Check Queueing Configuration:

```
ovs-vsctl list port <interface>
ovs-vsctl list qos
ovs-vsctl list queue
```

Display Queue Stats:

```
ovs-ofctl queue-stats br0
```

## Pica8 Troubleshooting

### Logs

Logs are stored in */tmp/log/messages*

## Dummynet Troubleshooting

Check ipfw configuration:

```
ipfw show
```

Check pipe:

```
ipfw pipe 1 show
```

## Appendix D - nmeta Caveats

### Caveats

- Updates required to support identity for systems with multiple NICs
- As noted, further work is required on data management to prevent table size bloat.
- The system only supports OpenFlow version 1.0.
- YAML creates an unordered dictionary, require strict order for policy
- Written and tested on Python version 2.7.5. May not work as expected on Python 3.x

### Future Enhancements

- Complete data management work
- Consider event driven tidy-up too, i.e. port goes down, purge any port related data from tables

- Improve TC policy functionality by adding nesting ability etc.
- Add support in static module for IP address range and netmask matches
- Add support for IPv6
- Add support for IP multicast
- Add support for IP fragments
- Add support in identity module for IEEE 802.1x
- Consider moving tables to a database
- Improve API functionality
- Add support for OpenFlow versions 1.2 and 1.3 including meters
- Add support for VLANs and other similar network virtualisation features
- Add support for distribution of controllers such that flow metadata maintains loose consistency across the distributed system allowing horizontal controller scaling
- Add security features. Really this should be top of the list. How can DoS of the system be prevented? As the system receives packets from the network, is it vulnerable to exploits sent in network packets not directly to it? How can this be mitigated?
- Make the routing/switching configurable (currently just a basic switch). Leverage other systems that do this rather than writing something new.
- Make classifiers plug-ins so that they can be developed and added/removed without requiring changes to the main code.