Scalability and Performance Considerations for Traffic Classification in Software-Defined Networks

by

Matthew John Hayes

A thesis
submitted to the Victoria University of Wellington
in fulfilment of the
requirements for the degree of
Master of Science
in Network Engineering.

Victoria University of Wellington
2016

Abstract

Scalable network-wide traffic classification, combined with knowledge of endpoint identities, will enable the next wave of innovation in networking, by exposing a valuable layer of network metadata for applications to consume. We leverage the promising new paradigm of Software-Defined Networking (SDN) to create architecture for scalable traffic classification (TC). In this paper, we demonstrate scalability issues inherent with running full TC services over a traditional SDN architecture through test results. We then propose an architectural modification for scalable TC built atop of widely-available OpenFlow SDN switches, implementing higher-complexity classification functions on commodity hardware. We validate this approach through a prototype implementation, with experimental results that demonstrate the scalability of our approach.

Acknowledgments

Thank you to my supervisors, Professor Winston Seah and Dr Bryan Ng, for their encouragement, pertinent questions and advice. Thank you to InternetNZ for providing funding.

Contents

Chapter 1

Introduction

We live in an Information Age, where digital innovation is transforming all areas of our lives. We shop globally, from anywhere, through Internet commerce sites that never shut. Internet voice and video calls allow us to see and talk to people around the world as if they are in the next room, as well as sharing and collaborating in ways previously unimaginable. Virtual communities form, irrespective of the tyrannies of distance, exchanging ideas and building new knowledge.

Machines talk to machines too, exchanging information independently of humans, perhaps the electricity meter supplying a reading to the utility company, or a seismometer reporting seismic activity to a monitoring site. Aeroplanes talk to each other to avoid collisions, and cars will do so too in the near future.

Data networking is the foundation technology on which the Information Age is built. All of this progress relies on transmission of information over distance. Yet, until recently, data networking has not been open to innovation at the same pace as other areas of the Internet, due to the proprietary bundling of network hardware and software into vertically integrated products. This closed paradigm impedes innovation by preventing the unbundling of layers of the network stack, but is being rescinded by the emergence of Software-Defined Networking (SDN).

SDN is a recent network concept that decouples *control plane* decisions, on how to make and maintain connections, from high-speed forwarding of packets in the *data plane*. Through decoupling, each plane can evolve separately, encouraging innovation by breaking down traditional proprietary vertical integration and replacing it with standardised open programmatic interfaces. Innovators and academics can test new control plane applications and protocols, without having to also engineer a data plane, as they can leverage programmatic interfaces into existing SDN data planes.

The SDN approach has another key benefit; it allows for *logical centralisation* of the control plane. Whereas vertically integrated routers and switches each have their own limited view of network state, a logically centralised control plane can benefit from a *network-wide view*. Network applications can consume this information, once the preserve of isolated *middleboxes* such as firewalls, load balancers and proxy servers, delivering new services across the network.

Much of the recent innovation in SDN is built on OpenFlow [1], popular open SDN standard that defines a communications channel between the control and data planes. In OpenFlow, a *controller* is a software-based control plane system that manages one or more data plane *switch(es)* via an OpenFlow *channel*.

In previous work, we investigated [2] and demonstrated the functional benefits [3] [4] of OpenFlow SDN for *Traffic Classification* (TC) in enterprise networks, through a prototype system called *nmeta* [5]. TC is the glue that enables network applications to work at the flow level, rather than per packet.

The primitives for the use of any network can be summarised into classes of *participant* and *conversation*. The primary role of a network is to transport conversations, comprising of any number of *flows*, between

participants (labelled as P1 and P2), as per Figure 1.1.

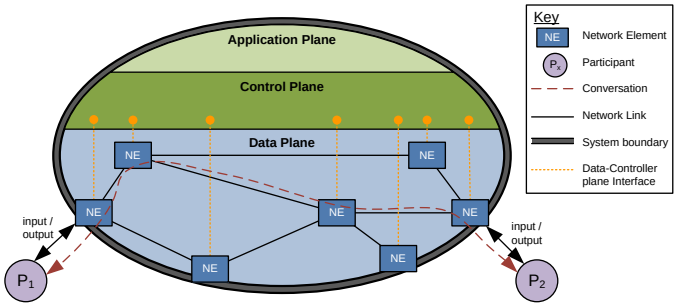


Figure 1.1: A Systems-Based Network View – flows and participants interact over different paths traversing multiple network elements.

With nmeta, we enhanced the network-wide view, with TC metadata about participants and types of conversation. This metadata has many potential benefits for network operators. It can be used to make traffic classification determinations based on metadata parameters not usually accessible, such as endpoint identity and flow statistical profile, and can do so in an online timescale such that actions can be taken before flows have had a chance to ramp up. This allows operators to write more abstract Quality of Service (QoS) and Traffic Engineering (TE) policies, referencing aspects of endpoint identity and flow behaviour, instead of network address and port numbers. Nmeta generates flow and identity metadata that can be used for online and offline use cases, as per Figure 1.2 [1].

The nmeta system leverages OpenFlow Packet-In message capability to send unmatched full length packets from the switch, via the controller, to the nmeta application. The packet headers and payload are thus available for inspection by traffic classification algorithms and other functions, such as forwarding engines. The ability to see specific full-length packets, along

[1] source: https://mattjhayes.github.io/nmeta/

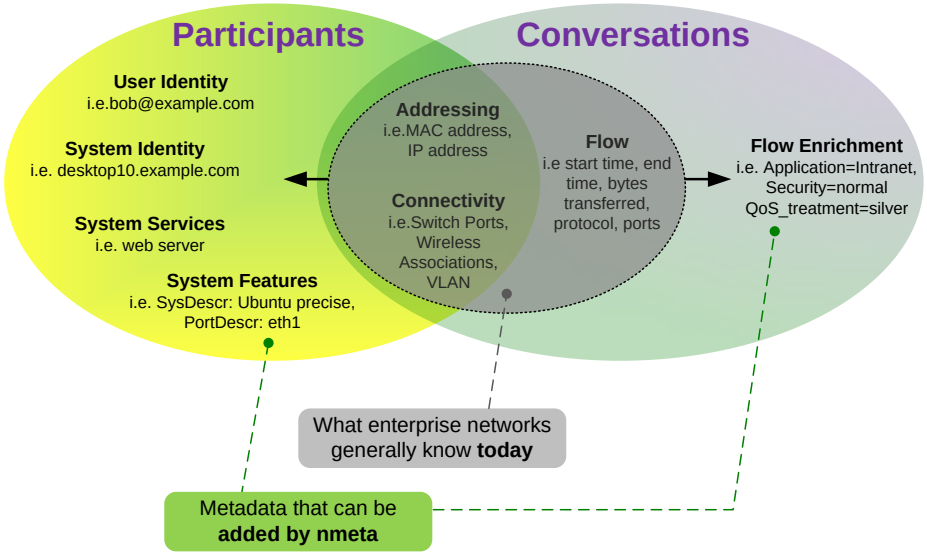


Figure 1.2: Nmeta Enhancements to Network Flow and Identity Metadata

with a network-wide view, provides nmeta with a powerful foundation for running policy-defined traffic classification.

Where the previous work demonstrated the functional benefits of SDN for traffic classification, it did not however investigate the non-functional requirements, such as security, availability, performance and scalability. This thesis extends the aforementioned previous work by investigating the *non-functional* engineering concerns, in particular, those of *performance* and *scalability*.

1.1 The Problem

The logical centralisation of SDN presents *challenges to scalability* when used for TC. Nmeta follows the SDN logically centralised architecture, however the nmeta fine-grained multi-classifier TC requires one to many packets per flow to be made visible to the application via the control plane. This in turn requires data plane packets to traverse the control plane, im-

pacting performance at many layers, most notably on the switch and the controller.

OpenFlow was designed as a means for controlling simple data plane switches, allowing the control plane to react to network events in a reliable manner and enforce a forwarding policy. It was however, not designed to transport large volumes of data plane packets to a TC network application.

In this thesis we investigate architectural changes to the traditional OpenFlow SDN model that maintain a network-wide TC view, but do not overload the OpenFlow channel or controller with data plane traffic.

1.2 Objectives

The objectives of this thesis are to:

- Investigate and document the performance and scalability limits of the nmeta system.
- Design a system that makes full-featured TC scalable using OpenFlow SDN.
- Validate that the revised architecture has acceptable performance under load, and is scalable.

1.3 Tasks

The tasks of this thesis are to:

- Survey related work in the area of TC scalability.
- Measure and analyse the limits of nmeta scalability in terms of new flow rate.

- Implement an SDN-based traffic classification system that can scale to handle peak new flow rates of an enterprise-scale deployment.

1.4 Contributions

The contributions made by this thesis are the design and experimental validation of an enhanced nmeta platform that can scale to meet the performance requirements of an enterprise-grade network platform in terms of peak new flow rate.

Note that a subset of this thesis is being considered for publication by the IEEE Systems Journal.

1.5 Thesis Structure

This thesis is organised as follows. Chapter 2 discusses related work in the context of our thesis. Next, in Chapter 3 we demonstrate that traffic classification in the control plane has a flow rate scalability problem, and we propose a revised solution architecture to solve this problem in Chapter 4. In Chapter 5, we detail the design of a prototype system that implements a subset of the revised architecture, discussing the design decisions and trade-offs. Next, in Chapter 6 we demonstrate improvements to performance and scalability through experimental results from our prototype implementation. Lastly we draw conclusions and propose future work in Chapter 7. Appendices provide supplementary detail on test methodology.

Chapter 2

Related Work

This thesis combines the topics of Traffic Classification and Scalability in Software-Defined Networking, both sizeable fields, thus there is a large volume of related work to consider. In this chapter we present an overview of some important related work.

2.1 Scalability of Software-Defined Networking

A recent survey paper by van Asten et al. [6] reviewed SDN scalability papers such as Hyperflow [7], ONIX [8], DevoFlow [9], Kandoo [10] and FlowVisor [11]. They concluded that scalability solutions fall into the general categories of *improving controller performance*, *reducing controller functions* or *partitioning controllers into separate domains*. We review related work against these categories below.

2.1.1 Improving Controller Performance

Controller design can be a critical factor that impacts the performance and scalability of an SDN system. Tootoonchian et al. [12] investigated controller optimisation and improved the performance of the NOX OpenFlow controller, by a factor greater than 30 times, by making it *multi-threaded*.

Similarly, Maestro [13] improves controller performance through parallelism, spreading controller workload across multiple cores for near linear scalability. Their system relies on there being no complicated dependencies between flow requests (i.e. Packet-In events are close to atomic). Note that this assumption may not hold for traffic classification, especially for situations where multiple packets in a flow must be inspected to make a classification determination.

Shalimov et al. [14] review the performance of OpenFlow controllers and conclude that current controllers (as of October 2013) are *'not ready to be used in production'* due to poor scaling over cores, and security deficiencies. They specifically mention that Ryu (the controller used by nmeta) does not have multi-threading, and thus cannot scale to use multiple cores, and say that it is *'more suitable for fast prototyping than for enterprise deployment'*.

2.1.2 Reducing Controller Functions

Various papers propose performance and scalability improvement schemes that localise and/or better distribute load by devolving functions from the controller to the switch.

Curtis et al. [9] present Devoflow, a concept that augments OpenFlow with an additional layer in the data plane to improve scalability. They see flows as differing types. 'Security sensitive' and 'significant' flows require further inspection, but 'normal' flows do not, and should be kept in the data plane. Switches are highlighted as performance and scalability bottlenecks due to slow control-data path and restricted FTE capacity. They introduce the concept of *'elephant flows'*, flows that are long lasting and transfer significant volumes of data. A trade-off is identified between full flow visibility with impaired performance against just visibility of elephant flows and improved performance and scalability. They present results from simulations rather than testing, as their scheme had not been

implemented in hardware, and conclude that it greatly reduces the number of FTEs and control messages.

Lin et al. [16] describe an architecture where they reduce traffic classification traffic to the control plane by extending the OpenFlow protocol and leveraging this to communicate with additional data plane components to affect classifications.

2.1.3 Partitioning Controllers

Various papers propose Network Operating System (NOS) that abstract the network state across multiple controllers. HyperFlow [7] is a system that allows horizontal scaling of controllers, with no change to OpenFlow. HyperFlow uses the WheelFS distributed filesystem to provide a partition-resilient publish/subscribe mechanism between controllers to keep network-wide state synchronised. Similarly, ONIX [8] is a closed-source NOS provides a distributed controller network-wide state via an API. ONOS [15] is an open distributed controller platform targeted at meeting the requirements of large service providers. ONOS provides an API into a distributed global network view of switch/link topology and network events, plus ability to install flows via intentions abstraction.

A different approach to scaling is taken in Kandoo[10]. A topology of local controllers bereft of network-wide state each serve only a small number of switches. A root controller provides a platform for applications that need network-wide state. Applications that do not need network-wide state are distributed across the local controllers to provide better response and scalability.

2.2 Traffic Profile Considerations

Traffic profiles are an important consideration when designing a scalable SDN solution. When Casado et al. [17] presented *Ethane*, a precursor to

OpenFlow, they identify broadcast traffic as a significant challenge since it resulted in 90% of the flows in their deployment.

The distribution of flow durations also requires consideration. Flow setup delay due to SDN processing can become more significant for shorter flows as a ratio. Benson et al. [18] observed that many data centre flows were less than 100ms in duration. For short flows such as these, increases in controller processing time may become a significant impediment to the flow.

Benson et al. [18] also observed an enterprise data centre where 80% of flows had inter-arrival times less than 1ms. This is significant for SDN scalability, as it causes high instantaneous rate of new flow load on the controller. The good news was the concurrency of flows was relatively low, meaning switches need not have overly large flow tables.

2.3 Traffic Classification

In our previous work [3], we divided TC techniques into the following categories:

- **Static**. Utilise packet header information and/or network context such as ingress port or Virtual LAN (VLAN).
- **Identity**. Classify based on the identity of one or more of the participants in the conversation.
- **Statistical**. Use flow features to make a determination. Can extend to Machine Learning (ML) techniques, although these have challenges with regards to training and ground truth.
- **Payload**. Look into the payload of one or more packets. Also known as Deep Packet Inspection (DPI). Can be computationally expensive and cannot cope with encrypted payload.

We define *full TC services* as the ability to carry out all four TC types listed above, and to be able to combine them in a hierarchical policy.

We use revisit these categories later in this thesis, analysing where they are best-suited to be run in the SDN architecture.

Qazi et al. [19] present a framework called Atlas that uses machine learning in conjunction with SDN to classify applications. They source flow to application ground truth for ML training from the network socket readings on end devices. They modify OpenFlow to store packet size statistics, and note that packets could be mirrored if this modification was not possible due to scalability issues, such as TCAM constraints. Their solution scales through distribution of the ML workload to the edge network device (in this case, access points). It is unclear how this solution can work with existing SDN infrastructure, or how timely the classification would be.

Li et al. [20] propose extensions to the OpenFlow standard, called OpenFlow Feature eXtraction (OFX), that implement programmable flow feature extraction. They use a Network Flow Processor (NFP) to accelerate the extraction of statistical flow features at the switch and run machine learning algorithms on the controller to classify the traffic. The centralised TC processing in this approach may not be suitable for online use cases that require actions in real time based on TC determinations due to control plane latency and it is unclear how the control plane could be scaled.

Recent work by Donato et al. [21] provides a retrospective on TIE (Traffic Identification Engine), an open source, non-SDN TC platform for use by research community to evaluate TC performance.

2.4 Summary

In this chapter, we reviewed SDN scalability work in the context of the general categories of *improving controller performance*, *reducing controller functions* and *partitioning controllers into separate domains*. Multiple approaches to solving SDN scalability exist in each of these categories, however most do not deal with the specific challenges of scaling TC on SDN, the exception of [16]. Our work is however, to the best of our knowledge, unique as it deals with the challenges of scaling full TC services in SDN.

Chapter 3

Scalability of SDN TC

In this chapter, we present experimental results demonstrating scalability and performance issues inherent in running full TC-services on standard SDN architecture. We describe our test methodology, experimental results, and analyse the root causes of the limitations.

3.1 Baseline Application - nmeta

We use nmeta as the baseline application to test performance and scalability of full TC services in a traditional SDN model. The nmeta application runs on the Ryu OpenFlow controller, as shown in Figure 3.1. Nmeta has a Northbound API providing access to identity and flow metadata, and performance metrics. Data structures in the nmeta application hold policy, identity metadata, flow metadata and traffic classification (TC) state.

Nmeta has a fine-grained *flow-based* view of network traffic, as opposed to coarse-grained address-based approaches. This point of difference is significant as it a) requires a means to identify packets to flows, b) requires visibility of packet features not available via existing OpenFlow standards, and c) has implications for data storage and scaling.

Currently, nmeta only supports a single controller (Ryu[24]), with a single nmeta application instance tightly coupled to it. The only scalability

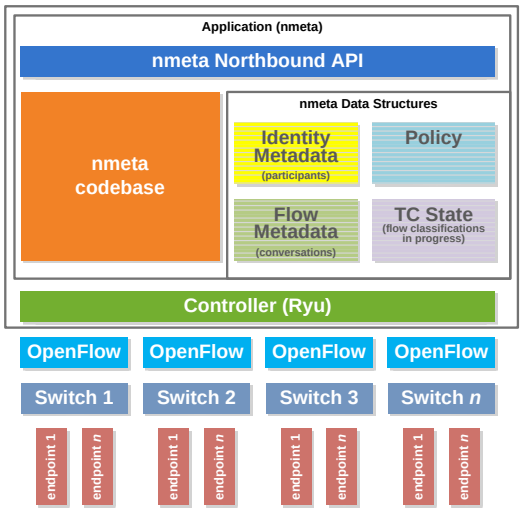


Figure 3.1: Nmeta Application on Ryu OpenFlow Controller Architecture

built into nmeta is the ability to scale the number of switches, as per Figure 3.2.

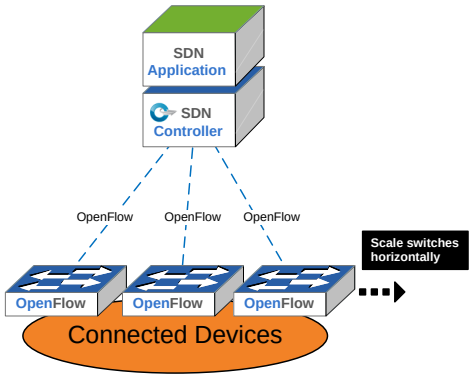


Figure 3.2: Nmeta Scales to Multiple Switches

3.2 Baseline Evaluation Methodology

The performance and scalability of the nmeta system was baselined with the methods outlined in this section.

We designed a simple experimental network topology to evaluate scalability constraints, as well as verifying traffic classification functionality.

3.2.1 Test Environment

A test environment was designed and built to run controlled repeatable tests. It runs on an Oracle VM VirtualBox[25] hypervisor, as per Figure 3.3.

The host machine for the hypervisor is an Arch Linux workstation with dual Intel E5 processors with a total of 8 cores and 64 GB of RAM. Multiple Ubuntu Linux guests (version 14.04.02 LTS 64 bit desktop) are instantiated within the hypervisor. Oracle VM VirtualBox was selected for the role of hypervisor due to familiarity with the product from previous work, and also its ability to host multiple isolated network segments. The latter point is important as it allows a guest to be run as a switch that connects multiple layer-1 segments.

3.2.2 Test Topology

We designed the test network topology for measuring the effects of load on network performance, as well as identifying constraints that contribute to deteriorating performance under load.

There are seven Linux virtual machine guests in the baseline test topology, as per Table 3.1. One instance acts as an SDN switch, and forms the centre of a four-legged star topology. Traffic between the four star-connected instances must pass through the switch. A separate Test Control instance has discrete connectivity to all instances, and the switch to controller connection is also via the out-of-band network. This design was chosen to minimise test control traffic impact on test results.

Separate cross-switch paths are used for load testing;

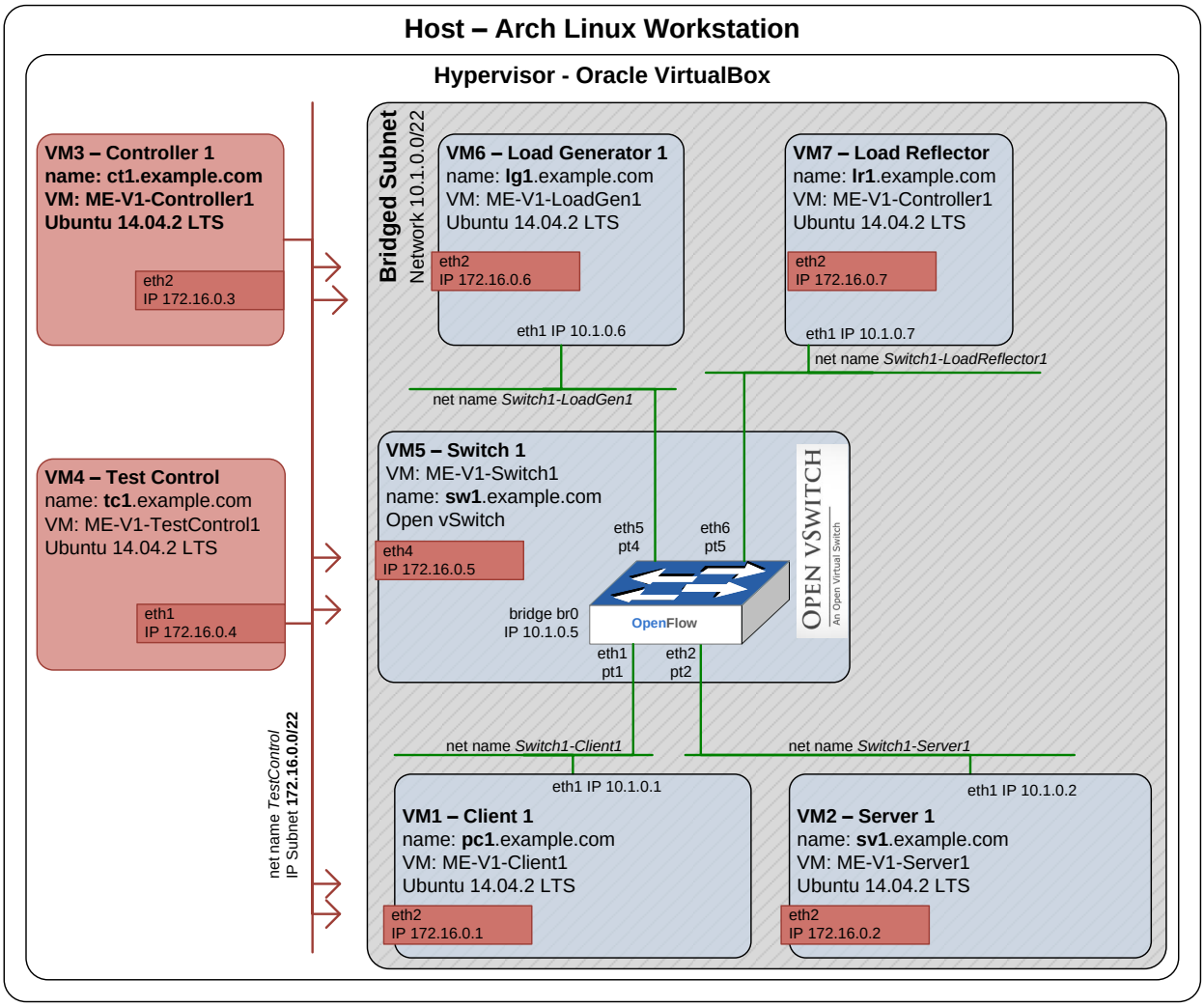


Figure 3.3: Virtual Test Environment

Name	Role	Description
pc1	Client	Runs performance measurement tests across the network
sv1	Server	Destination for performance measurement tests
lg1	Load Generator	Generates load across the network
lr1	Load Reflector	Acts as a sink for unidirectional load
sw1	Switch	OpenFlow SDN switch (Open vSwitch) connecting pc1, sv1, lg1 and lr1
ct1	Controller	OpenFlow Controller (Ryu)
tc1	Test Control	Automation and Orchestration of tests via separate out-of-band connection to other guests

Table 3.1: Baseline Test Guests

- Load Generator to Load Reflector. This path is used to add new flow load to the system.
- Client to Server. This path is used to measure performance.

3.3 Test Automation & Orchestration

During the planning phase, we recognised that time spent automating and orchestrating testing would be a worthwhile investment, due to the volume and complexity of test data generated, along with the need for significant numbers of test repetitions. Automation also has the benefit of reducing the risk of human error influencing test results. For these reasons, significant effort was put into automating and orchestrating the control and analysis of tests for this thesis.

Our guiding principles for the design of our testing system were:

- **Nonmonolithic**. Create (or download) simple reusable programs to

automate specific functions, and tie them together with orchestration software in such a way that we avoid building a monolithic application and retain flexibility to quickly and easily adapt to new test requirements.

- **Reuse**. Design programs to be reusable for multiple similar requirements, with a simple interface.
- **Telemetry**. Capture telemetry data in addition to the required result data so that test performance can be validated and unexpected results diagnosed.
- **Data Quality**. Result data should be easy to ingest (ideally CSV format), clearly identifiable as to the source, and have accurate timestamps
- **Test Atomicity**. All tests were designed to be self-contained, so that conditions from one test cannot influence the results of another test.

We derived a test architecture from these principles, that features an 'orchestration sandwich', as per Figure 3.4. The top layer is automation that iteratively calls an orchestration layer with multiple interleaved test types. The orchestration layer coordinates automated activities on a number of entities, retrieves results and requests automated post-processing of result data (if required).

3.3.1 Orchestration

Test orchestration (coordinating the timing and order of execution of tasks on multiple entities) is performed by the Test Control guest. It carries out the following high-level orchestration tasks per test:

1. Ensures that all guests are in the correct state to commence a test
2. Starts the various processes on guests to monitor test parameters

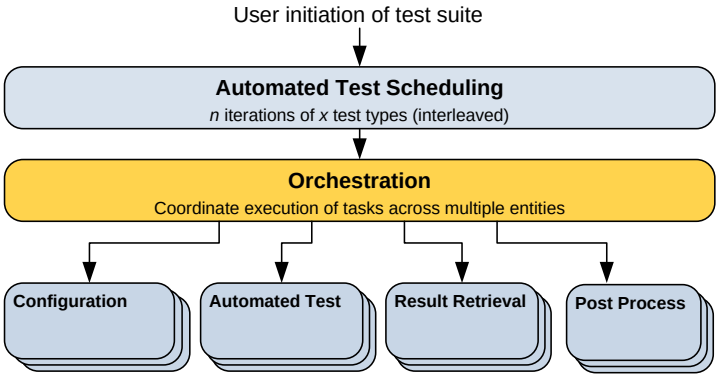


Figure 3.4: Test Architecture 'Orchestration Sandwich'

3. Starts the load test and waits for it to complete

4. Stops test processes

5. Retrieves result data and stores it in a directory structure

We chose Ansible® [26] software for our test orchestration function. Ansible is agentless free open source configuration management software, and uses SSH to communicate with managed nodes. Ansible abstracts the individual managed nodes through a construct called an *inventory*, and comes with a variety of built-in modules to handle a variety of common automation tasks. Tasks are described in a *playbook*, a YAML file that is used to control a set of automation tasks.

We use simple Python programs to run automated tests. They iteratively call an Ansible playbook with test-specific variables, interleaving test types to minimise the effect of external factors during the tests.

Ansible, while useful, is not perfectly suited to automating test suites. Its forte is automation of server builds, which generally requires scheduling and checking of many sequential tasks. The test suites that we run require many sequential tasks (example: checking that a process is running), but also many tasks that must run indefinitely in parallel. Examples

of such tasks are processes that regularly poll to gather performance metrics while a test is in progress. To work around the limitations of Ansible in this area, these tasks are executed in Ansible asynchronous mode, with polling disabled, and a maximum runtime that is greater than the length of the test. The Linux pkill command is used for ending these tasks at the completion of the test. While not elegant, this solution was found to work reliably.

The Ansible *fetch* module is used to retrieve result files and store them in a timestamped directory structure, along with information relating to the test context, such as program versions and the playbook contents. R [27] is used to analyse the result data and create output charts. The test configuration is summarised in Figure 3.5.

Result data files are read in by programs written in the R programming language, and *data frames* (data tables of equal length columns of variable measurements) are generated for the each heterogeneous source of result data.

It is necessary to index each test run by a field other than *Time*, so that results from all test runs can be combined on the same chart. Elapsed time within each test run is not a suitable index as there could be variations in when the load starts. Changes to test tools were considered such that each tool would poll filt to get the current actual load. This idea was discounted, as it would have been complicated to code, prone to error and could have influenced test results. Instead, a method was developed to post-process result data in R.

3.3.2 Test Tooling

Custom-developed Python command line tools were chosen as the preferred approach for test tooling, for the following reasons:

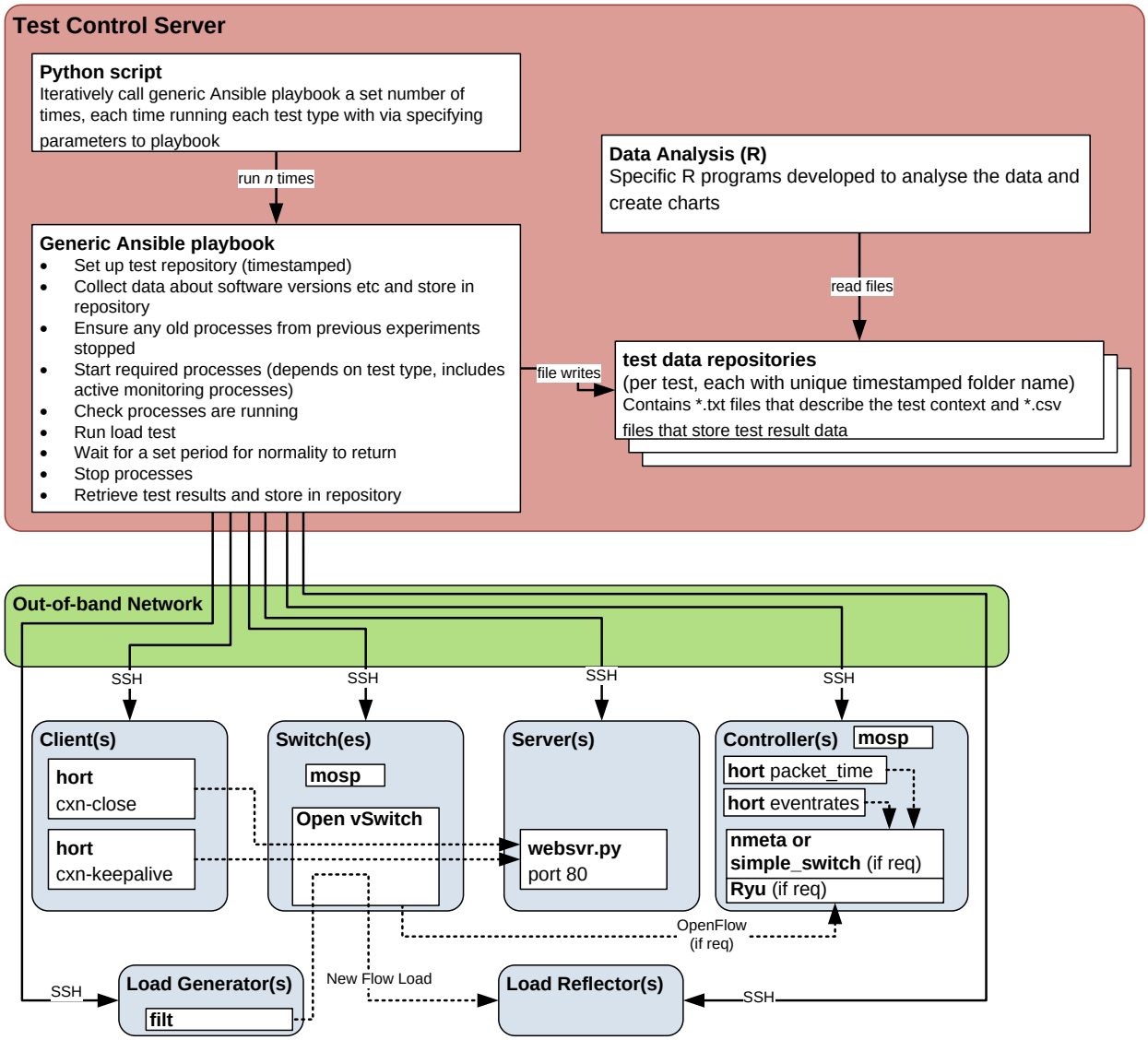


Figure 3.5: Test Configuration

- Custom development gives control over testing and measurement methodology
- Results can be generated in a common format that simplifies post-test analysis
- Development effort is minimised by leveraging existing Python libraries
- Direct control of configuration parameters from command line facilitates centralised test automation

3.3.3 Load Test Methodology

We hypothesised that nmeta performance would deteriorate under high rates of new flows due to stress on the controller and application due to the requirement to send initial n packets in each new flow to the controller for classification (where n is dependent on the classifier). New flows per second (NFPS) is thus the primary factor against which nmeta performance and scalability is assessed.

3.3.4 NFPS Test Types

Three types of test were devised to differentiate result caused by environmental, SDN and nmeta factors. They are as follows:

- No SDN Test (referred to as *nosdn*). In this control test, the switch runs in non-SDN (fail-open) mode, to give a baseline result without OpenFlow and nmeta traffic classification.
- Simple SDN Switch Test (referred to as *simpleswitch*). This test runs the switch in OpenFlow mode, controlled by a simple switch application on the Ryu controller that installs FEs for learnt MAC addresses, but does not run nmeta.

- nmeta SDN Test (referred to as *nmeta*). This test runs the switch in OpenFlow SDN mode and the nmeta application on the Ryu controller. Nmeta is run with a simple single rule static classification policy.

During shakedown testing it was discovered that nmeta was not installing flows for the NFPS load traffic, as the destination MAC was not known to it. This caused nmeta to flood the NFPS load packets, and thus not install a flow entry in the switch. This unintended behaviour was resolved by instantiating a separate guest to use as a load target (Load Reflector).

IPTables was configured on the Load Reflector guest to silently drop connections to the NFPS target TCP port. This was done to supress return TCP Reset packets that would otherwise effectively double the NFPS load, as each TCP Reset caused a packet-in punt to the controller, and a subsequent flow install and packet-out.

A step was added to the Ansible playbook to command the Load Generator to regularly PING the Load Reflector. The PING reply packet from the Load Reflector allows Nmeta to learn and maintain the Load Reflector MAC to switch port mapping and avoid flooding.

3.3.5 Functional Performance Evaluation

We evaluated functional network performance by carrying out regular measurement of HTTP object retrieval times. To do this, we developed a Python command line tool, called *hort* (HTTP Object Retrieval Test) [28]. Hort writes timestamped results to CSV file, and leverages the Python Requests [29] library for HTTP operations.

3.3.6 New Flow Rate Testing

Reactive SDN systems are known to have limitations in terms of maximum rate of new flows [30]. Nmeta was placed under increasing new flow rate load to ascertain the point at which functional performance metrics are materially impacted. A Python command line tool that puts the system under repeatable incremental new flow load, called *filt* (Flow Incremental Load Test) [31], was developed for this thesis. Filt writes timestamps and load rates to CSV file for comparison with other measurements, and has a stepped incremental traffic profile, as per Figure 3.6. Filt leverages the scapy [32] Python library for packet operations.

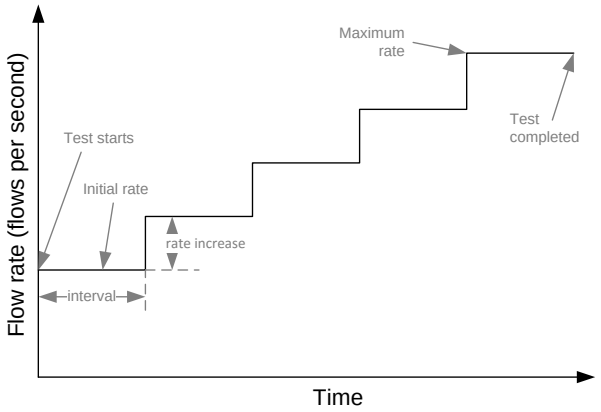


Figure 3.6: Example Filt Traffic Profile

More details of how we developed filt, and the challenges with performance and accuracy, are detailed in Appendix B

3.3.7 System Performance Measurement

To measure and record operating system performance parameters, we developed a simple Python script that leverages the psutil [33] library. We call this script *mosp* (Measure Operating System Performance) [34]. Mosp

records CPU load, memory swap and network volumes to CSV at defined regular intervals.

3.4 Baseline Validation Results

We carried out tests to ascertain the maximum NFPS load rate at which performance remains acceptable, which we arbitrarily define as latency no more than 50% above the no load rate.

3.4.1 Initial Results

Test results showed that nmeta in our virtual test environment scales to approximately 70 - 90 new flows per second, above which new flows are adversely impacted, as per Figure 3.7.

The *nosdn* control tests and *simpleswitch* tests were not impacted by the NFPS load. Flows in progress were not affected by the new flow load for any of the test types, indicating that the data plane forwarding performance was not impacted by the NFPS load.

Controller CPU in the *nmeta* tests plateaued at 100% utilisation from around 80 new flows per second, indicating a severe CPU constraint in the control plane for this test type.

Analysis of a packet capture on the controller showed the following:

- The point at which nmeta becomes ineffective (which we term *break point*) correlates with the point at which the controller reaches 100% CPU.
- The Controller Packet-Out rate, which was previously symmetric with the Packet-In rate, drops rapidly past the break point.
- OpenFlow Errors from the switch relating to invalid buffers do not occur until 15 seconds after break point, indicating that this test is

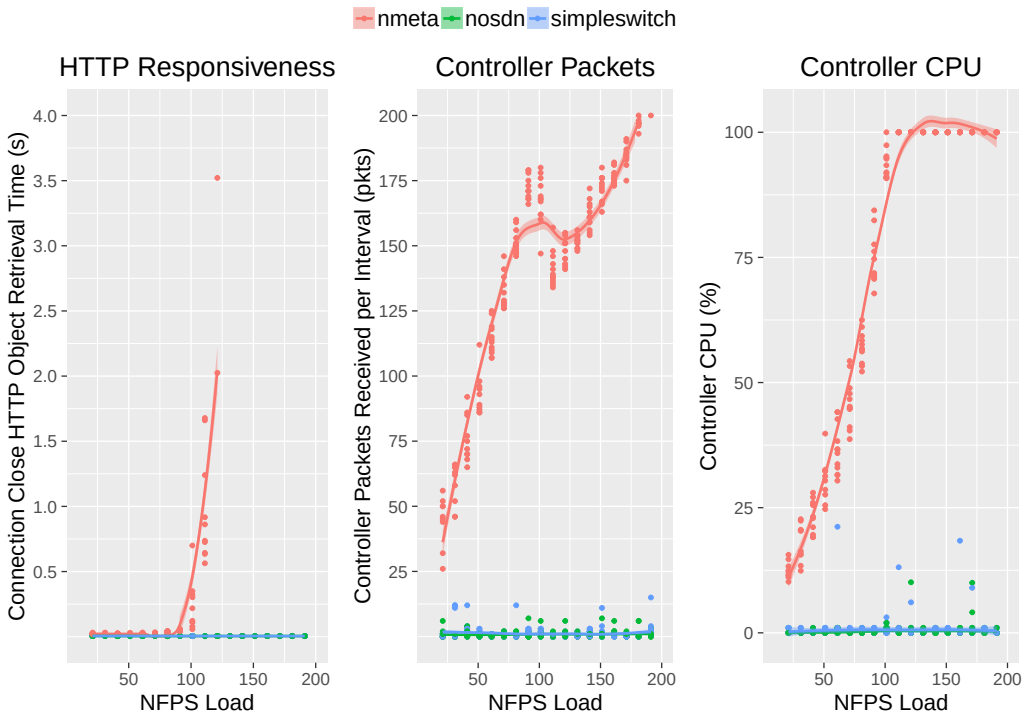


Figure 3.7: NFPS Load Impact on nmeta HTTP Responsiveness and Correlations (source: Table A.1)

not constrained by buffer exhaustion on the switch, since performance degradation occurred prior to errors occurring.

3.4.2 Controller CPU Analysis

We analysed Controller CPU usage by running the Linux *top* command on the Controller in batch mode with results written to file every 3 seconds. The results showed that a single Python process is the main consumer of CPU, and the consumption of CPU goes up sharply after 30 NFPS of load. This supports our hypothesis that the controller is CPU bound.

3.5 Summary

In this chapter, we described our experimental methodology and proved our hypothesis that nmeta performance would deteriorate under high rates of new flows through experimental results. These results showed that nmeta has a flow rate scalability problem resulting in a significant deterioration in performance when the rate of new flows per second exceeds an environment-specific threshold. The constraint that causes the problem is controller CPU. Solving this issue requires more than adding controller CPU resource, due to the severity of the constraint. In the next chapter, we propose solutions to this problem that involve moving the TC workload to other areas of the system, so that it can be more easily scaled to cope with peak new flow loads.

Chapter 4

Solution Architecture

In this chapter, we describe a minor revision to SDN architecture that solves the performance and scalability problem we demonstrated in the previous chapter.

4.1 ONF SDN Architecture

The Open Networking Foundation (ONF) are a not-for-profit industry-driven organisation that promotes the uptake of SDN through open standards[35]. They are responsible for maintaining and evolving the OpenFlow SDN standards, and also perform other roles including defining an architecture for SDN[36][37].

We use the ONF SDN architecture as our baseline, against which we propose changes. In the ONF SDN architecture, an *SDN Controller* component interfaces southbound with *resource groups* that may contain *network elements* (NE) in the data plane via *data-controller plane interfaces* (D-CPI), and northbound to *SDN applications* via *application-controller plane interfaces* (A-CPI).

4.2 Architecture Revision

We make a minor revision to the ONF SDN architecture through the addition of a new type of NE component to processes data plane workload for cases where it goes beyond the native capabilities of the switch. This component, which we call a *Data Plane Auxiliary Engine* (DPAE, pronounced *Dee-Pie*), is integrated with the network data and control planes, but able to scale separately.

The DPAE exists primarily in the data plane, with connectivity to the control plane for signalling. The DPAE is a Network Element (NE), as shown in a revision of an ONF diagram in Figure 4.1.

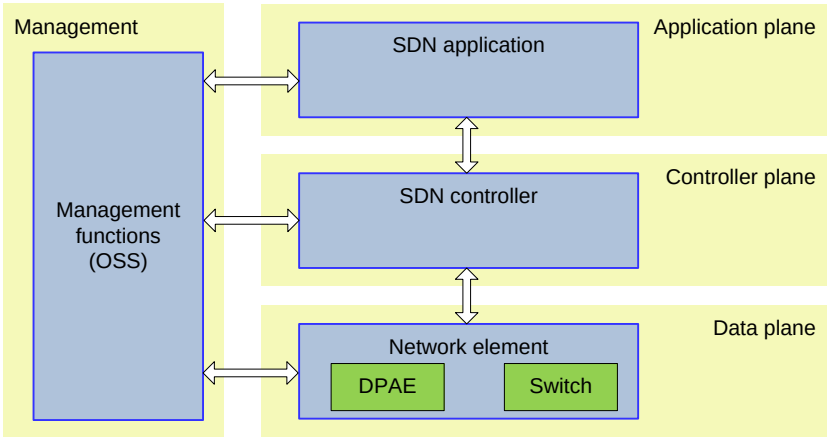
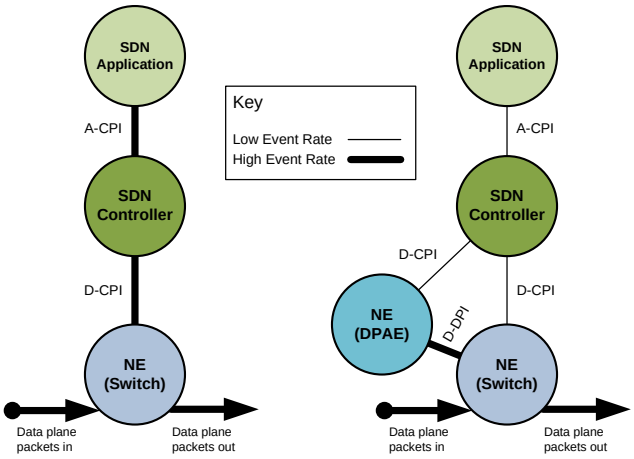


Figure 4.1: ONF SDN Architecture Diagram[38], Revised to Show DPAE and Switch

The DPAE reduces load on the controller and application by carrying out data plane workload locally, and sending only relevant updates to the control plane, as per Figure 4.2.

A new type of interface, *data-data plane interface* (D-DPI) is defined to connect DPAE to switches. The DPAE can be scaled separately to the switch, improving the scalability of the solution as a whole. The DPAE is also extensible to other roles that require packet processing capabilities beyond what the switch can offer.



(a) Standard solution (b) Our solution

Figure 4.2: Architectural Comparison: Increasing Scalability by Confining High Event Rates to the Data Plane.

By decoupling the switch from auxiliary data plane services, we reduce the breadth of the switch specification, and avoid creating a monolith. Where SDN decoupled monolithic vertically-integrated networking devices, by separating the data and control planes with a network, we decouple the packet forwarding functions of a switch from more advanced services, again with a network.

The DPAE can be scaled horizontally across a network. A single switch can balance its load across multiple DPAE, and a single DPAE can support one or more switches, as per Figure 4.3.

The ONF SDN architecture recommends that models and components are reused, to reduce standardisation and testing effort, and also that SDN must be deployable. We align with these principles by limiting the new components to only what is necessary, and building on commodity platforms that are simple to deploy.

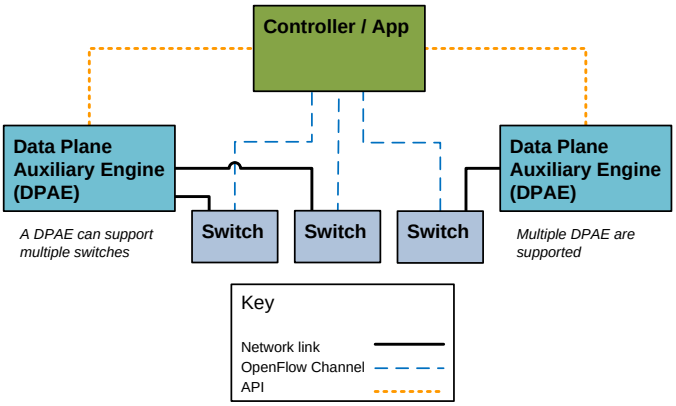


Figure 4.3: DPAE Scalability Options

4.3 Summary

In this chapter, we presented a revised architecture that solves the performance and scalability problems inherent in our previous architecture by relocating the majority of per-packet TC workload from the application to the switch and a new component, the DPAE.

In the next chapter, we present the design and build of a prototype implementation of our revised architecture.

Chapter 5

Design of Scalable TC with OpenFlow SDN

In this chapter, we present the design of our prototype implementation of the revised architecture. The prototype implements a subset of our architecture, sufficient to allow testing of performance and scalability. It is a complete rewrite of nmeta into a controller application called *nmeta2*[39], and a DPAE called *nmeta2dpae*[40]. Both are written in Python and available on GitHub. Figure 5.1 shows a high-level representation of the components of the prototype implementation.

5.1 Inter-Component Communications Design

In this section we discuss the design of communications between the DPAE, switch and controller/application.

There are two new communication interfaces introduced in our design for DPAE connectivity; the *Switch to DPAE D-DPI* and the *DPAE to Controller D-CPI*.

These communication interfaces must be efficient, to handle the load generated by full TC services. This applies in particular to the Switch to

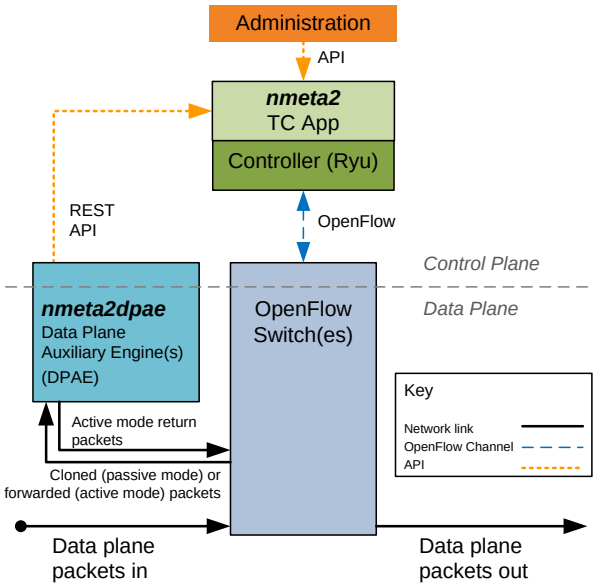


Figure 5.1: Overview of Prototype Implementation

DPAE D-DPI, as it may have a high rate of invocation and carry full packets.

5.1.1 Switch to DPAE D-DPI

Our design for switch to DPAE communications is deliberately light-weight, with minimal overheads. Packets are sent natively over Ethernet between the switch and the DPAE. There are benefits and drawbacks to our light-weight approach.

Other approaches[16][3][4] leverage the OpenFlow channel to carry data plane packets that require TC, via OpenFlow Packet-In messages. We term this the *OpenFlow-encapsulation* approach, as opposed to our light-weight *native-packet* approach.

The OpenFlow-encapsulation approach requires the switch to encapsulate the packet in an OpenFlow Packet-In message, within TCP, within an IP packet. In addition to the extra computational load, this approach also

adds network overhead, both where the packet size results in an encapsulated packet larger than the link's *maximum transmission unit* (MTU) thus incurring fragmentation, and when the packets are small resulting in the network overhead becoming proportionally large.

We found the OpenFlow-encapsulation approach adds 108 bytes of overhead per packet (66 bytes of Ethernet/IPv4/TCP, and 42 bytes of OpenFlow header, assuming an IPv4 backhaul). This is a high overhead ratio, especially for smaller sized packets. A 74 byte packet on the wire to the switch became a 182 byte packet in OpenFlow. This is a 146% increase in size. For large packets, this fixed per-packet overhead can put them over the MTU of the backhaul link, causing fragmentation.

The native-packet approach is a relatively efficient operation on the switch, when compared OpenFlow-encapsulation. The native-packet approach has no per-packet overhead for the transmission to the DPAE. It is also a relatively efficient operation on the switch, when compared OpenFlow-encapsulation, as it has lower CPU requirements as packets do not require OpenFlow and TCP encapsulation / decapsulation and the switch is only required to copy the packet buffer to a transmit queue.

The choice between the native-packet and OpenFlow-encapsulation approaches comes down to a trade-off between the speed and efficiency of the native-packet approach versus the reliability and capability of the OpenFlow-encapsulation approach.

The native-packet approach minimises processing overhead on the switch, does not suffer from undesirable TCP behaviours such as slow-start and does not have a per packet overhead, meaning fragmentation is not necessary (assuming same MTU). It also does not delay control plane traffic with head-of-line blocking. It is however vulnerable to packet loss due to queue tail drops, and without TCP there is no network-layer recovery.

There is also no flow control. The DPAE has no method to signal that it would like the rate to be reduced.

The OpenFlow-encapsulation approach has advantages, through metadata signalling packet context (reason, table id, match (including ingress port)), whereas the native-packet handoff of packets from the switch to the DPAE is blind. The DPAE does not know which port the packet originated from, only which DPAE interface the packet was received on and the switch name that this correlates to (via a discovery protocol). The DPAE also does not know the reason for the packet being sent to it. We work around the former limitation by deducing the original switch ingress port at the controller, based on the learned source MAC address to switch port mapping.

We address the latter by avoiding the need for the DPAE to understand why the packet was sent to it, by keeping DPAE packet processing generic.

5.1.2 DPAE to Controller D-CPI

Our prototype DPAE to Controller D-CPI is a new protocol built on *representational state transfer* (REST) over HTTP. Our choice of REST over OpenFlow for this D-CPI was based on expediency. Developing features in REST is simpler than extending the OpenFlow protocol, however this choice should be revisited in future work. It may be desirable to have a common D-CPI for both switches and DPAE.

Our prototype protocol automates the initial configuration of DPAE, including switch/port auto-discovery and negotiating capabilities. The same protocol is subsequently used to download optimised TC policy and communicate identity and traffic classification outcomes to the controller.

While our prototype design has the DPAE talking to our application via an API, we envision that this will later become a standardised native feature of the controller, and hence will be to the controller.

In Figure 5.2, we show a high-level representation of the phases that occur during a DPAE join to a Controller.

It is worth noting that the creation of this new protocol was made possible by the network programming capabilities provided by OpenFlow SDN.

5.1.3 Switch to Controller D-CPI

Switch to controller communcations is unchanged, continuing to use OpenFlow version 1.3 for this D-CPI.

5.2 Packet Forwarding Design

In this section, we present the packet forwarding design of our prototype implementation, along with discussion on trade-offs and other design considerations.

5.2.1 Passive and Active Modes

Our prototype implementation gives the network operator a choice between two modes of operation; packets are either forwarded (*active mode*) or cloned (*passive mode*) by the switch to the DPAE, where they are processed. In active mode, the DPAE returns the packet to the switch, to continue processing through the pipeline, whereas in passive mode the packet continued through the pipeline in parallel.

5.2.2 Flow Suppression

We reduce packet rate load on the DPAE by instructing the switches to send the traffic direct, once a TC determination has been made, by installing specific FEs that match the flow with instructions to bypass the DPAE.

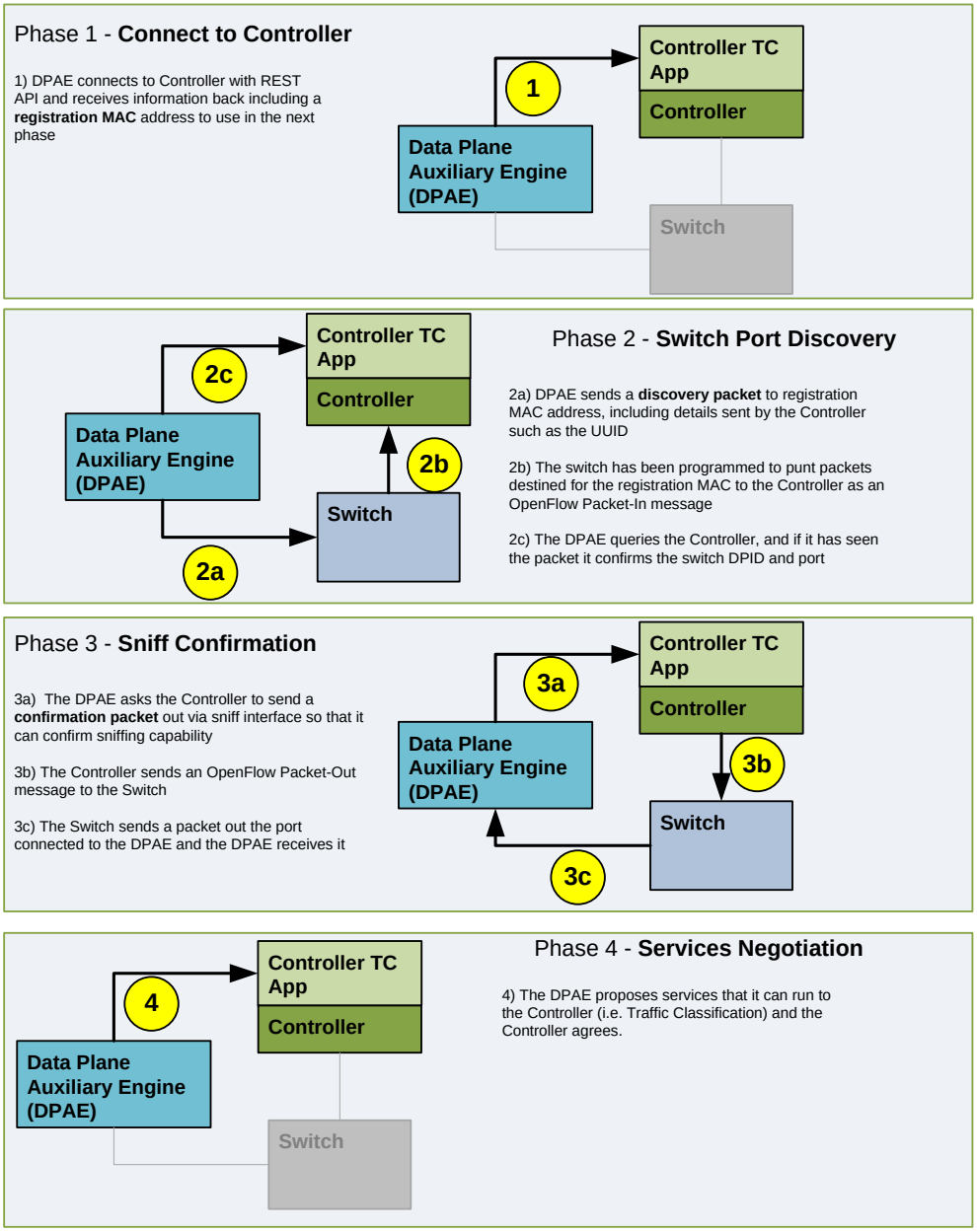


Figure 5.2: Overview of DPAE-Controller Join Protocol

Flow type is an important consideration when designing the suppression mechanism, since per flow suppression comes at a cost, adding load to the controller and switch. For this reason we want to suppress *elephant flows*, and ignore *mice flows*.

Elephant flows have a large number of packets, and generally comprise a small percentage of the total number of flows on a network, but a large percentage of data transferred. Mice flows, on the other hand, are generally short lived with few packets transferred, but are common [41].

Suppressing mice flows may not be worth the trade-off of load on the switch and controller to add FEs, especially if the flow finishes around the same time as this work is completed. An algorithm could dynamically adjust the packet count threshold to optimise for the relative load levels.

We leave the development of this algorithm to future work, and instead use a simple static value to define the per-flow packet count threshold, beyond which suppression should be initiated. We also define a resuppress threshold with a large value to catch any failure on the initial suppression.

5.2.3 OpenFlow Table Design

In OpenFlow, *flow tables* are collections of static classifiers, called *flow entries*. Flow tables evaluate packets against flow entries, sorted by priority order, and exit if a *match* is made. A match against an FE causes any associated *instructions* to be executed. There is no option to continue parsing in the same table after a match is made.

Multiple Flow Tables (MFT) are a feature of OpenFlow version 1.1 and later. MFT allows flow tables to be combined into a processing pipeline, enhancing the capabilities of OpenFlow by allowing multi-level logic and transformation to be applied to packets. We leverage OpenFlow MFT enhancements in our design.

There are good scalability reasons for wanting to leverage MFTs. Learning MAC address to port mappings is an $O(N^2)$ problem, but this reduces

to $O(N)$ with MFT [42] [43]. In a similar manner, we avoid Cartesian product scaling issues by using a flow table per function to reduce number of FEs, as per Figure 5.3.

For passive mode, we utilise a pairing of OpenFlow FE instructions, *Goto-Table* in conjunction with the optional instruction *Apply-Actions*, to clone a packet to the DPAE without affecting its normal passage through the pipeline. For active mode, we instead pass the packet to a filter table to ensure the destination MAC is known, and if so the packet is sent out the port to the DPAE.

Our design decouples the multiple functions of a TC switch into separate tables, and in doing so simplifies the packet forwarding pipeline and makes it scalable.

MAC Learning Considerations

Reinjection of packets from an active-mode DPAE into the switch has implications for MAC learning, as the OpenFlow pipeline OXM_OF_IN_PORT field for the packet becomes the DPAE switch port, not the original ingress port. Where the destination MAC is not known and flooding is required, this results in the packet being forwarded back out the original ingress port, as the OpenFlow FLOOD directive sends a packet out all ports except for the ingress port (as recorded in the OXM_OF_IN_PORT field) and blocked ports. The split-horizon principle for packet flooding is thus violated, the flooded packet egresses the ingress port, and the incorrect port to MAC mapping is learnt for the source MAC on switches in the path back to the MAC source.

We could solve this problem with FEs that update the OpenFlow pipeline field OXM_OF_IN_PORT with a set-field action, however this is prohibited by the standard[1], so we must use another means to prevent this occurring. We solve this problem instead by enforcing that packets are only sent to the DPAE if the destination MAC location is known. This has the minor drawback that TC cannot occur on a flow where the destination MAC

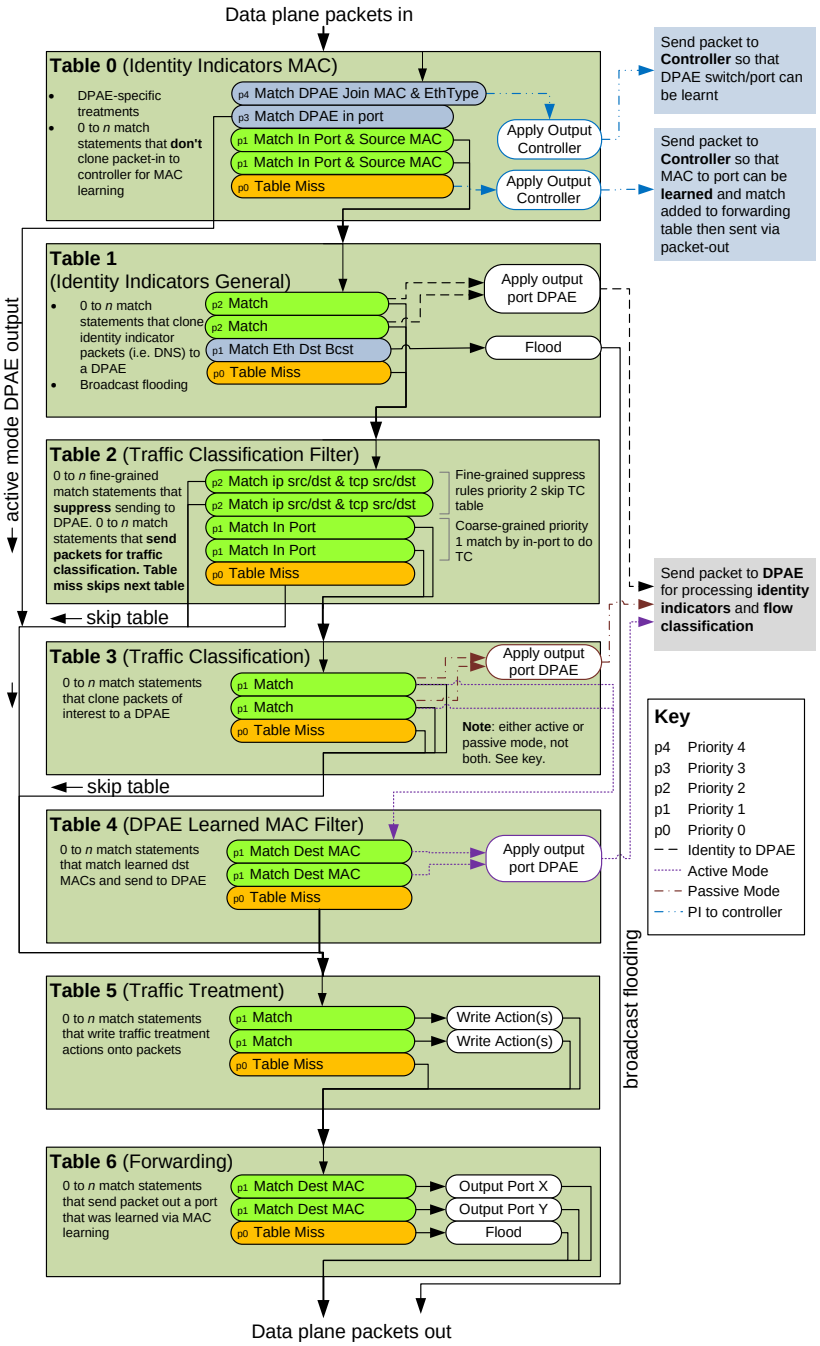


Figure 5.3: Solution Pipeline

location has not been learnt. In reality this is an unlikely scenario as ARP requests will be sent by end stations and thus learning will occur prior to the main traffic stream.

Our principles for the design of flow tables for the packet pipeline are:

- DPAE discovery packets must be sent to the controller, and not continue through the pipeline.
- Packets returned from the DPAE must bypass all tables except for treatment and forwarding to prevent incorrect learning and to improve pipeline efficiency.
- MAC learning is achieved via an exception process that sends packets that do not match higher priority source MAC and ingress port rules to the controller via a Packet_In message. Learned packets do not continue through the pipeline, the controller sends them via Packet_Out function out the learned destination port or via the FLOOD directive (ensuring the in_port is correctly set to the ingress port).
- DPAE harvesting of identity indicators where the destination is the broadcast MAC (FF:FF:FF:FF:FF:FF) or multicast MAC (where the least-significant bit of the first octet is set to 1) must be a clone operation as the DPAE must not forward packets destined for broadcast or multicast MACs. This operation must occur prior to the broadcast bypass rule.
- A broadcast bypass rule should output packets destined for the broadcast MAC early in the pipeline for efficiency, via the FLOOD directive.

MFT Packet Pipeline

The MFT packet pipeline design is as follows:

Table 0 - Identity Indicators General

Table 0 serves multiple purposes. Primarily, it is used to clone identity indicator packets to the DPAE. It also matches DPAE join packets and sends them to the Controller, as well as flooding Ethernet broadcast packets out ports, to reduce load on subsequent tables. Additionally, it matches active mode DPAE return packets and sends them to the traffic treatment table. Unmatched packets are sent to the next table to continue pipeline processing.

Table 1 - MAC

Table 1 is used for suppressing duplicate MAC learning messages to the controller. When the Controller learns a MAC addresses, it instantiates a flow entry in this table that matches the learnt MAC address as a source, with an instruction to goto the next table, and an idle timeout that deletes the FE when stale. The table miss rule clones the packet to the controller for MAC learning, as well as continuing pipeline processing with a goto next table instruction. This table thus serves as an efficient method for controller learning of source MAC address to port mappings, required for switching.

Table 2 - Traffic Classification Filter

Table 2 is used as an opt-in method of selecting which ports run full TC. Packets from matched ingress ports are sent to the next flow table for TC, whereas table miss skips the next table with the instruction to goto the Traffic Treatment table.

Table 3 - Traffic Classification

Table 3 is used send packets of interest to a DPAE for TC, or other DPAE processing. In active mode, the matched packets are forwarded directly to

the DPAE. In passive mode, matched packets are forwarded to the DPAE as well as being sent to the subsequent table to continue pipeline processing. All unmatched packets are sent to the next table.

Table 4 - DPAE Learned MAC Filter

Table 4 is used in active mode to filter packets such that only ones with learned destination MACs are sent to the DPAE.

Table 5 - Traffic Treatment

Table 5 applies treatment actions to packets, such as setting egress queues, by writing an action against the packet for later execution. All packets (including unmatched) are sent to the next table to continue pipeline processing.

Table 6 - Forwarding

Table 6 forwards based on learnt MAC addresses. Entries in this table are set with no idle timeout, as their freshness may differ from the learning MAC table FE. They can be shadowed by MAC learning entries when traffic is unidirectional inbound, causing them to age out, but never be relearnt. For this reason they are instead maintained by leveraging MAC table FE idle timeout eviction events to trigger the controller app to request the deletion of the relevant forwarding FE.

MFT Scalability Considerations

Our MFT design has $O(N)$ scalability in terms of total FE usage, as each use of an FE is either discrete, or as is the case with MAC address learning, requires use of three FEs per MAC address, as each learned MAC is used in three separate flow tables.

5.3 TC Design

In this section, we present design considerations from our prototype implementation that are specific to traffic classification.

5.3.1 State Retention

DPAE instances should minimise state retention to reduce complexity and improve scalability.

Flow state on a DPAE is only significant per flow, meaning it does not require synchronisation between DPAE, if all packets in the flow go to the same DPAE and resilience against DPAE failure is not required. We proceed with these assumptions.

Identity state needs to be synchronised across all DPAE and controllers, as an individual DPAE will not have visibility of all identity indicators, yet may need to classify based on this remote information. The DPAE that receives an identity indicator is responsible for translating this into a common format, communicating it via the distributed data store, and finally, aging it out when it goes stale.

We did not have time in the scope of this thesis to implement state sharing for identity data between DPAE. It is left to future work to investigate this element of the architecture, and should be achievable by leveraging an existing distributed system program.

Our architecture supports horizontal scaling of DPAE, where a single DPAE can service multiple switches. A limitation of our prototype solution is there cannot be more than one DPAE per switch, however this limitation could be resolved by implementing a mechanism to synchronise FCIP state between DPAE, and using the group table type *select* to balance packets over multiple DPAE.

5.3.2 TC Policy Design

We reused the syntax and structure of the nmeta main policy with some modifications. The QoS policy has been combined into the main policy to reduce complexity and and extra top level directive, tc_policies, is added to provide a higher level of abstraction. Port sets are used to define where TC should be run and the mode is selectable between active and passive.

5.3.3 Devolving Static Classification

Static classification matches against values present in packet headers, or environmental parameters such as ingress port or VLAN. OpenFlow switches already have extensive capabilities in this area (refer to '*Flow Match Fields*' in the relevant version of the OpenFlow Switch Specification standard). We follow our earlier principle, preferring switches over DPAEs, and instantiate simple non-nested static classifiers as flow entries on the switch.

5.3.4 Devolving Identity Classification

Identity classification requires tracking the state of various identity indicators, some of which are only visible to the switch directly connected to the device being identified (examples: LLDP, 802.1x). Others, such as MAC addresses, require knowledge of network topology, in order to understand if the device is directly connected, or via another switch. Our prototype does not have topology awareness, so we leave this feature to future work.

We clone identity indicator packets to the DPAE so that it can carry out the heavy-lifting required to parse them and maintain state regarding their freshness. The DPAE communicates updates to identity indicator states via REST API calls to the controller/application.

5.3.5 Devolving Statistical Classification

Statistical classification requires the ability to run an arbitrary program and feed it flow features, such as packet size, arrival time etc, and this requires compute and stateful flow capabilities, neither of which is not present in OpenFlow version 1.3. We thus send all packets that require statistical classification to the DPAE for processing. Accurate timestamping of packet arrival times presents a challenge as the classifier runs on the DPAE, not the switch, and variable delay may be introduced under load. A future version of OpenFlow may include native collection of flow features, as demonstrated in OFX [20], however the capability is not present in the version we employ. Our prototype solution uses the timestamp of the packet arrival time at the DPAE, however this can be subject to variability due to ingress queueing on the DPAE. Future work is required to assess the materiality of this to the accuracy of statistical classification.

5.3.6 Devolving Payload Classification

Payload classification requires inspecting packet payload in the first n packets, and may require complex stateful operations. We devolve payload classification to the DPAE as it is not suitable to run on the switch.

5.4 Performance Tuning

While developing the nmeta2 and nmeta2-dpae code, we carried out tuning to maximise performance, especially on per-packet operations. Here are some of the main optimisations that we made;

5.4.1 DPAE Concurrency Design

The DPAE needed a means to handle operations concurrently. We chose to use the Python multiprocess module for this purpose, it forks the process,

thereby allowing concurrency and avoiding global interpreter lock (GIL) issues inherent in a thread-based concurrency approach.

We split the DPAE into separate processes for data plane and control plane functions to minimise delays to packet processing. This is especially important in active mode, as packet processing directly impacts latency.

In our initial design for nmeta2dpae, we had packets passing through a Python multiprocess queue between the sniff and tc modules. We subsequently optimised the design by implementing a clearer demarcation between the data and control plane functions, and keeping packet flow within a single process. This optimisation resulted in a four times reduction in active mode latency, due to more efficient packet processing.

5.4.2 Packet Library

We initially used the scapy packet library, but changed to dpkt when we compared their performance and found dpkt to be faster.

5.4.3 5-Tuple Bi-Directional Hash and Database Indexing

We needed an efficient method to search and retrieve flow records, as this is a per-packet operation, adding latency in active mode. Our solution was a DPAE function that translates a TCP flow 5-tuple into a unique hash, matching packets in either direction on the flow. We used this hash as a key into a collection in a MongoDB database.

We configured an index of the hash field in the database collection, and the indexing resulted in a significant performance gain due, to reduced database query times.

5.4.4 Broadcast Optimisation

Broadcast traffic can comprise a significant portion of total network traffic [17]. We make an assumption that TC is not required for broadcast traffic,

and thus seek to optimise its path to reduce load on TC components. We do this by matching broadcasts in the first table and flooding them directly out ports.

5.5 Summary

In this chapter, we described the design and build of a prototype system that implements a subset of our revised architecture, along with lessons that we learned during the development lifecycle, including performance optimisations.

In the next chapter, we present experimental results from our prototype solution, and discuss how trade-offs can be managed to create a solution that meets reasonable requirements.

Chapter 6

Validation

In this chapter, we demonstrate improvements to performance and scalability of SDN TC through experimental results from our prototype implementation.

6.1 Test Method

We use the same test method as described previously, with the addition of a guest running the nmeta2dpae DPAE function to the test environment. We deliberately employ a worst-case scenario to test our solution to prove the viability of the most extreme case. We use an unqualified statistical classifier when testing our prototype implementation that sends initial packets from all new flows to the DPAE for TC.

We also do not optimise either nmeta or nmeta2/nmeta2dpae by turning off logging. Results therefore, should be worst case, and could be further optimised.

6.2 Performance

We tested the packet forwarding performance of existing solutions, and our new solution, under no load. Forwarding performance was assessed through two methods. Firstly, we used *hping3* to repeatedly measure a single TCP RTT, as per Figure 6.1. Secondly, we used *hort* [28] to retrieve an HTML object once a second, recording the total time taken, as per Figure 6.2.

We configured a statistical classifier that matches all packets for both types of nmeta2 test to test the worst case where the first n packets in a flow require TC inspection.

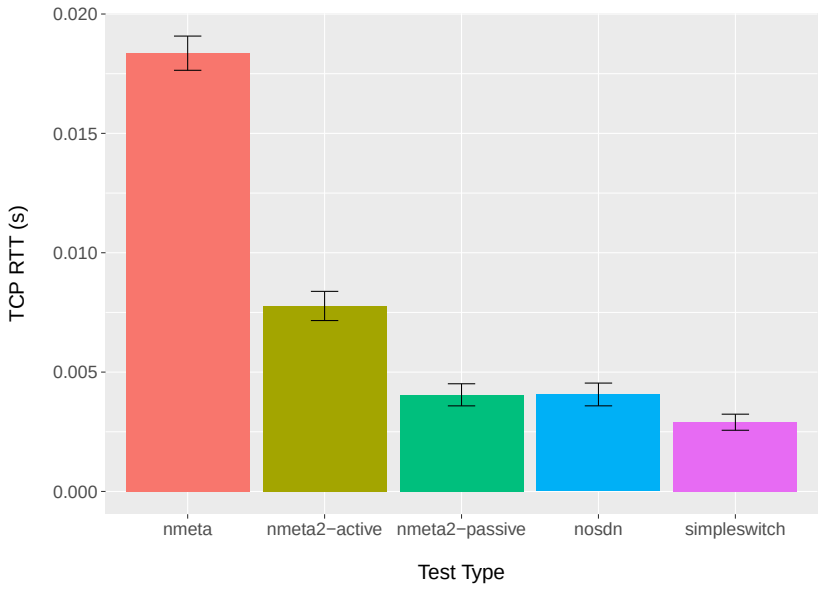


Figure 6.1: TCP RTT Performance No Load (Standard Error) (source: Table A.3)

The results show that nmeta has significantly worse forwarding performance for new TCP flows than the other solutions, with over 15ms TCP RTT and HTTP object retrieval time. The best performer in both tests was nosdn, which is to be expected, as everything is localised to the switch.

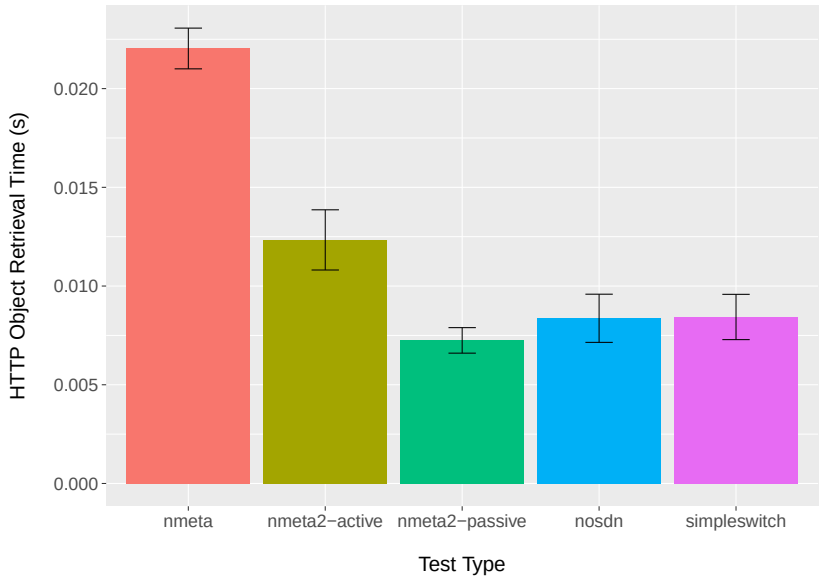


Figure 6.2: HTTP Object Retrieval Performance No Load (Standard Error) (source: Table A.3)

Neither nosdn or simpleswitch provide any TC services, so their good performance results are also expected. Their results are only included as a reference to the maximum performance of the environment without TC.

6.3 Scalability

In this section we validate the scalability of our solution to new flow rate by showing the reduction in load on the control plane in conjunction with ability to scale TC load separately to the control plane.

NFPS load is the primary dimension in which we measure the scalability of our solution. Ideally, scalability will be confirmed by performance that sustains reasonable levels under reasonable load.

6.3.1 New Flow Forwarding Performance Under Load

Our results, as per Figure 6.3 show that nmeta2 in passive mode provides comparable latency under NFPS load to the base switch (nosdn). When run in active mode, nmeta2 has a higher base latency, but this does not degrade significantly with increasing NFPS load. This performance improvement over nmeta is in addition to the added horizontal scalability of our solution.

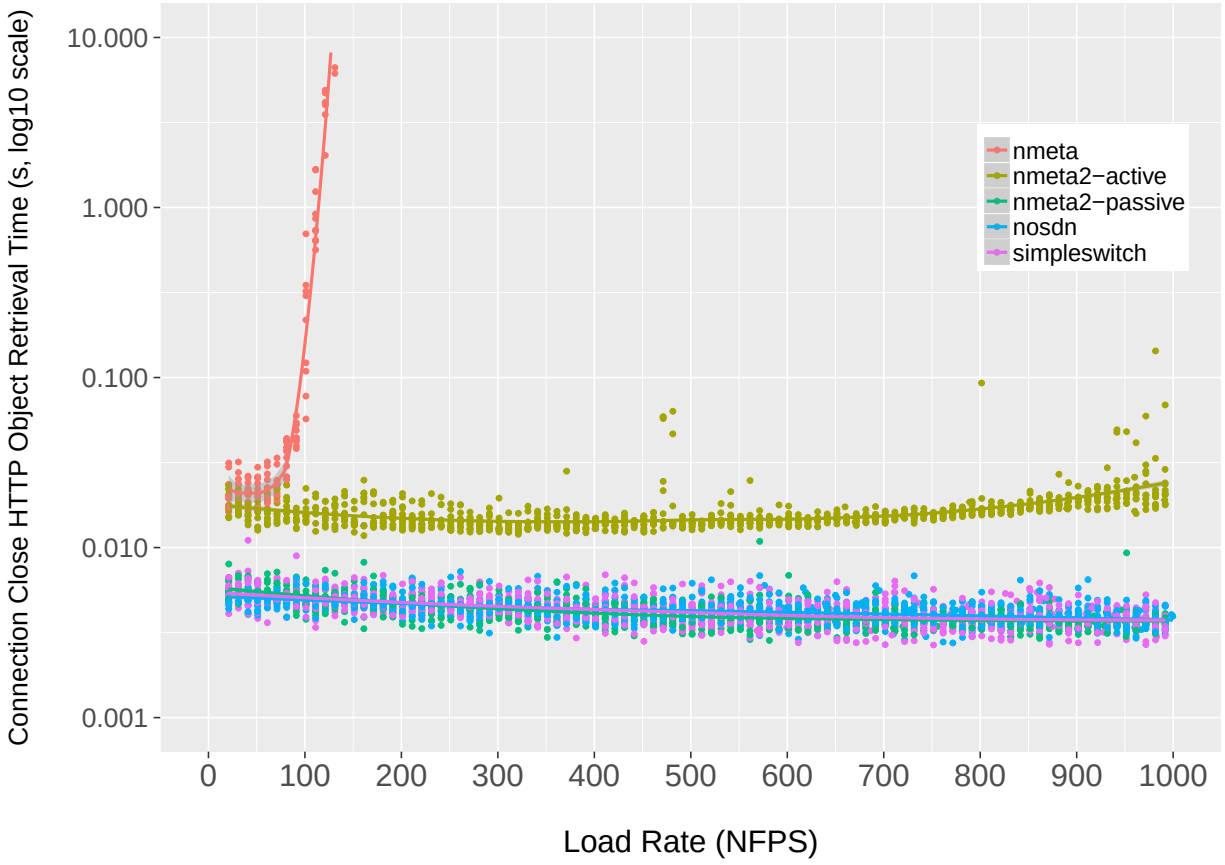


Figure 6.3: HTTP object retrieval times (new TCP connections) under NFPS load (source: Table A.1)

6.3.2 Existing Flow Forwarding Performance Under Load

The results for HTTP object retrieval times over existing TCP connections, as per Figure 6.4, show that NFPS load has no significant impact below 500 NFPS on any test type. This shows that forwarding performance for FEs in the switch is largely impervious to NFPS load.

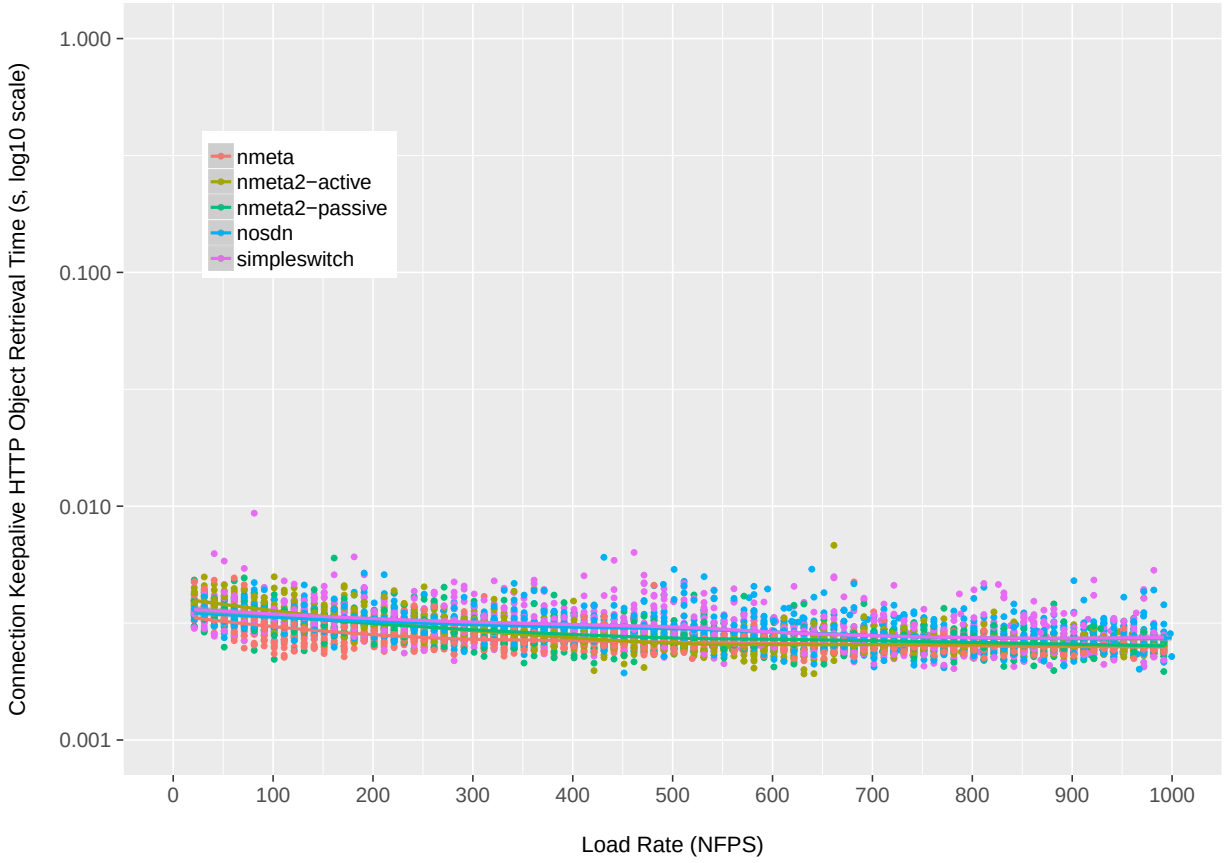


Figure 6.4: HTTP object retrieval times (existing TCP connections) under NFPS load (source: Table A.1)

In Figure 6.5 we see an order of magnitude lower packets to the controller in both modes of the nmeta2 solution when compared to nmeta. This correlates with a reduction in control plane CPU utilisation.

Figure 6.5: Controller Ethernet Packets In vs New Flows Load by Test Type (source: Table A.1)

Our prototype implementation moves the full TC services CPU workload from the controller to the DPAE, as per Figure 6.6. This chart clearly shows the improved efficiency of running the workload on the DPAE, as the DPAE CPU increase is linear and increases at a lower rate than the controller CPU for nmeta.

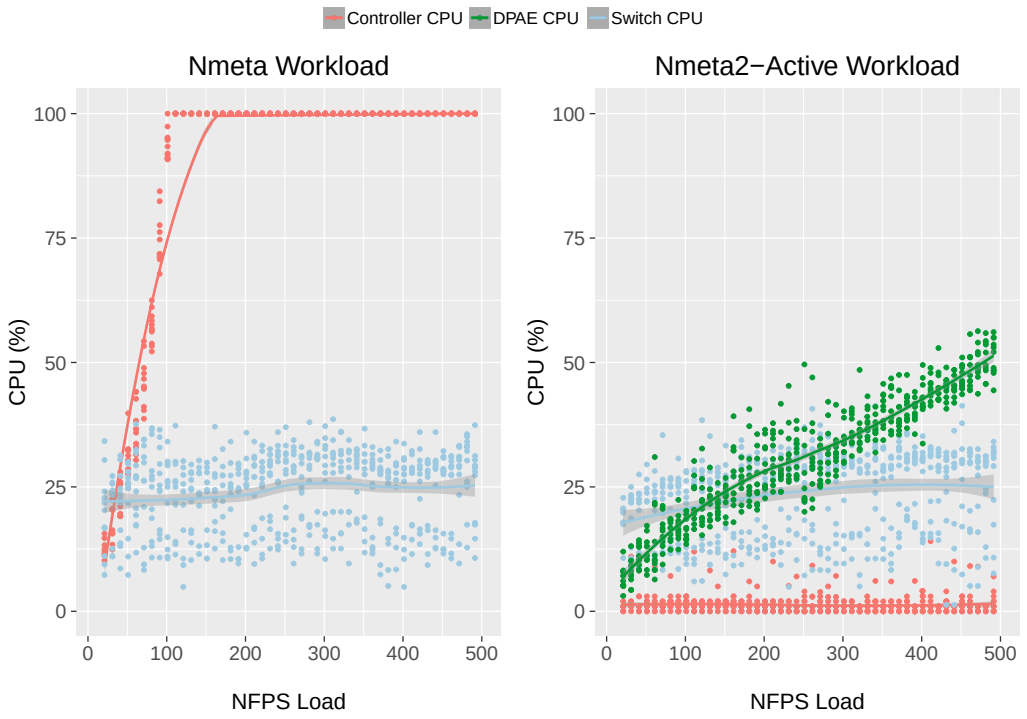


Figure 6.6: CPU Workload Comparison between nmeta and nmeta2-active (source: Table A.1)

6.3.3 Control Plane Timeliness Under Load

We evaluated the timeliness of control plane functions by assessing MAC learning performance for each test type at differing levels of NFPS load.

Our test methodology was as follows. We set the environment up for the particular test type, added the desired rate of NFPS, then sent 200 TCP

SYN packets (via a custom script) in rapid succession from the client to a special NIC on the server. This NIC had MAC address M, not previously seen in the environment, so not known to the switch or controller. We used a *macvlan* interface for this role, inside a separate network namespace, configured with an IP address belonging to the same subnet as the general server NIC. It was also necessary to change network adapter type in our virtual environment, as the default adapter did not support multiple MACs on a single guest.

We removed ARP from the test, as we observed that its performance was variable under NFPS load for some test types. We did this by using a static ARP on the server, and our script does not require ARP at the client end. There is thus no opportunity for the switch to pre-learn where M is in the topology, as there is nothing generating unsolicited packets from our special interface. MAC learning only occurs via TCP RST packets sent in response to the crafted SYNs of the test.

We measured the performance of the control plane response to the discovery of M in two dimensions. The first is MAC learning delay, which we measured as the time difference between initiating the sending of the first packet to M, and the time that the last packet to M is received by a participant (lg1) that should not be privy to the flow (i.e. last flooded packet receive time).

The second dimension is the number of packets to M that are flooded. The ideal result is that only a single packet is flooded, which is sufficient to learn the location of the target MAC via a response packet. Our test is unusual however, as it sends the packets unacknowledged with a very small interpacket delay (1ms).

Our results, as per Figure 6.7, show that control plane timeliness in nmeta is adversely affected by NFPS load. The MAC learning delay in

nmeta climbs almost two orders of magnitude by the time load reaches 150 NFPS, and correspondingly the number of packets flooded also increases significantly. Both of these results demonstrate that sending data plane packets over the control plane from the switch to the controller is detrimental to control plane timeliness.

In contrast, the performance of nmeta2 in active and passive modes remains stable under increasing NFPS for both MAC flooding delay and packets flooded. This shows the benefits of our design that removes TC load from the control plane. Note that the relatively high number of packets flooded for nmeta2 active and passive modes is an artefact of the prototype implementation, not the design, and will be reduced in a subsequent release of the code.

6.4 Scaling to Multiple Switches

We tested the scalability of our architecture to multiple switches by deploying our prototype into an enlarged test environment. We made an assumption that there would never be more than two DPAE in path between endpoints in a conversation as TC is a network edge function. We created an enlarged test environment for multiple-switch tests, as shown in Figure 6.8, employing the following design criteria:

- Tests can be varied between having 1 to n switches in path between clients and servers
- Out of band control network
- Number of switches can be changed without need to go outside hypervisor (i.e. Test Control server tc1 can enact the required topology by setting interface state on switch guests)
- Loop free. This is not a naturally loop free topology, so care needs to be taken setting the correct interface states on switch ports through

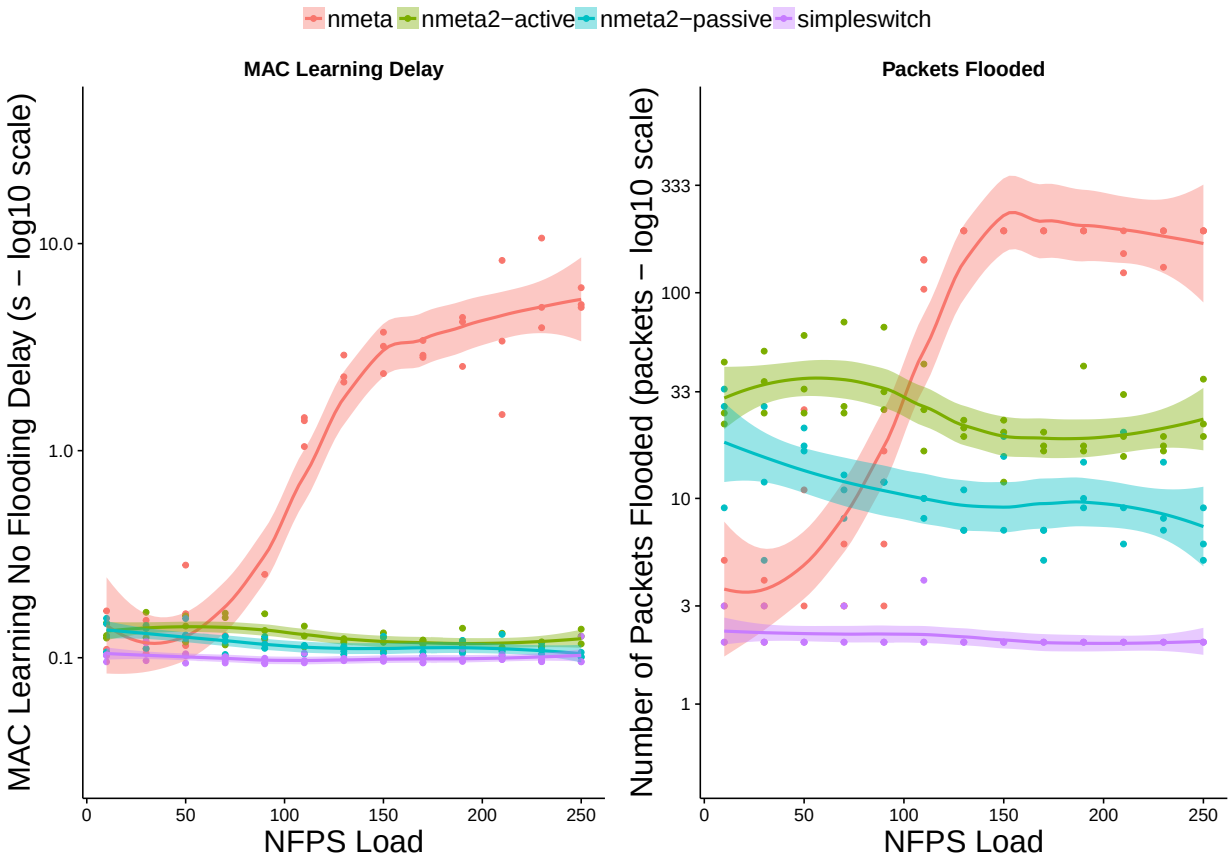


Figure 6.7: Control Plane Timeliness for MAC Learning Under NFPS Load (source: Table A.5)

automation.

Ansible playbooks enact the required network topologies by configuring the administrative state of switch Ethernet ports, such that the correct number of switches are in path, and the topology is loop-free.

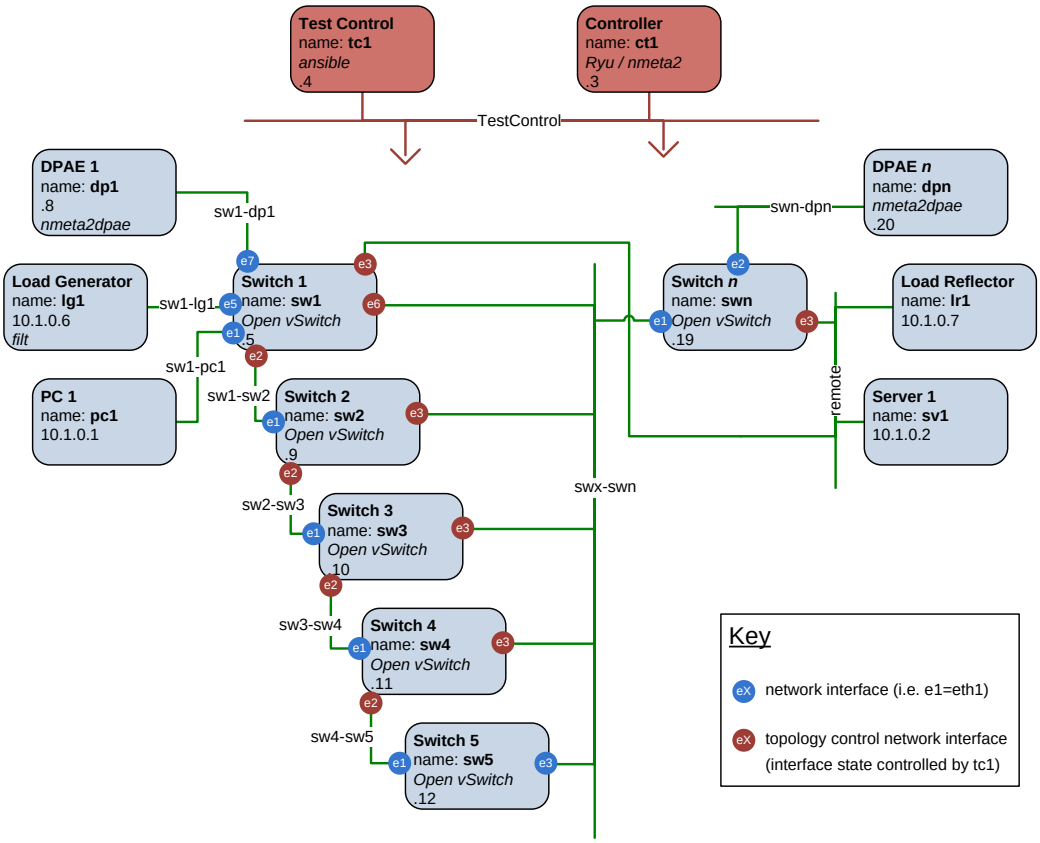


Figure 6.8: Multiple-Switch Test Environment

Performance tests of NFPS load with varying numbers of switches in path and active mode statistical classifier, as per Figure 6.9, showed acceptable performance that does not significantly degrade until past 750 NFPS.

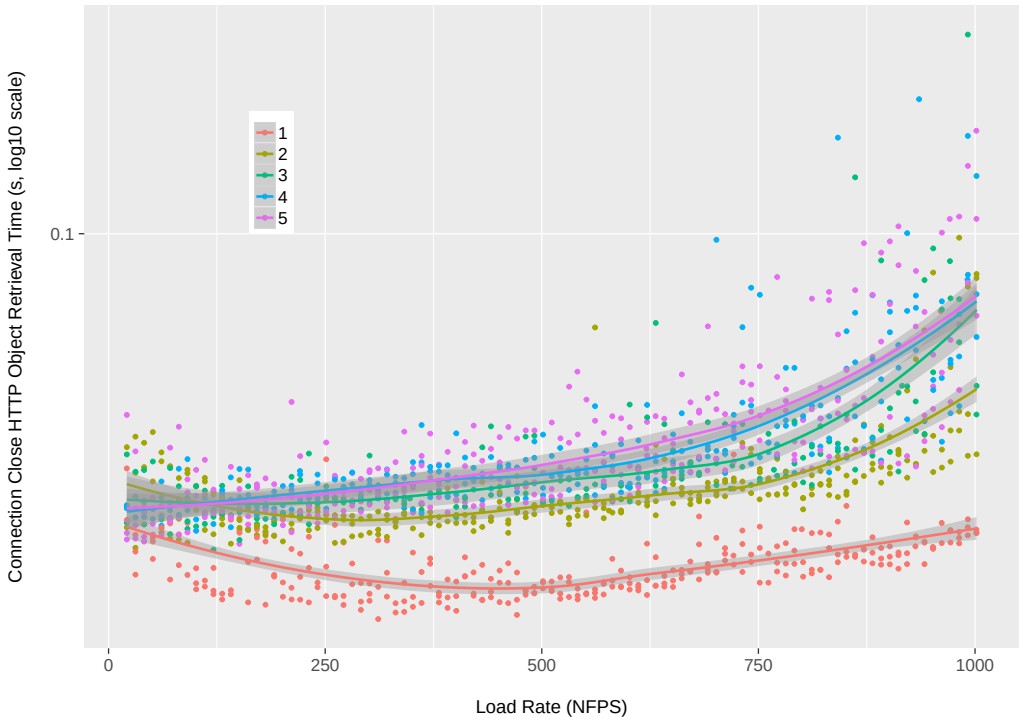


Figure 6.9: NFPS Load Impact on nmeta2 Active Mode New Connections by Number of Switches In-Path (source: Table A.2)

6.5 Capability

In this section we validate the traffic classification capabilities of our solution. We do this by validating the capabilities of our solution to handle the most challenging scenario, an online use case where all packets require inspection by an arbitrary program, and this program generates actions that must be applied to flows in close to real time. Specifically, our test is a statistical classifier that sets bandwidth restrictions on unusual bandwidth-hog flows based on their initial n packets. We evaluate the capability of our solution in two modes of operation, active and passive modes, comparing their capabilities in the dimensions of flow compliance to bandwidth constraint and packets to DPAE before flow suppression occurs.

6.5.1 Nmeta2 Classification Mode

Our testing showed that active mode is more effective for statistical QoS use cases. It provides a more timely classification in terms of elapsed packets until treatment is applied, resulting in more effective treatment for aggressive flows. This result demonstrated in Figure 6.10 [1], where we see active mode achieving a result close to the target bandwidth of 100,000 bps with packets to DPAE in the 4 - 10 range, whereas passive mode results in a higher average bandwidth and an order of magnitude more packets to the DPAE before treatment FEs are enacted.

Active mode achieves better classification timeliness due to the syncronicity between classification and control plane actions to set treatment.

We found that active mode achieved a three-fold improvement in compliance to the target bandwidth over passive mode, and sent an order

[1] lab: virtual, test: tc-timely-noload-statistical.py, results: /results/timeliness/statistical/20160507120638/

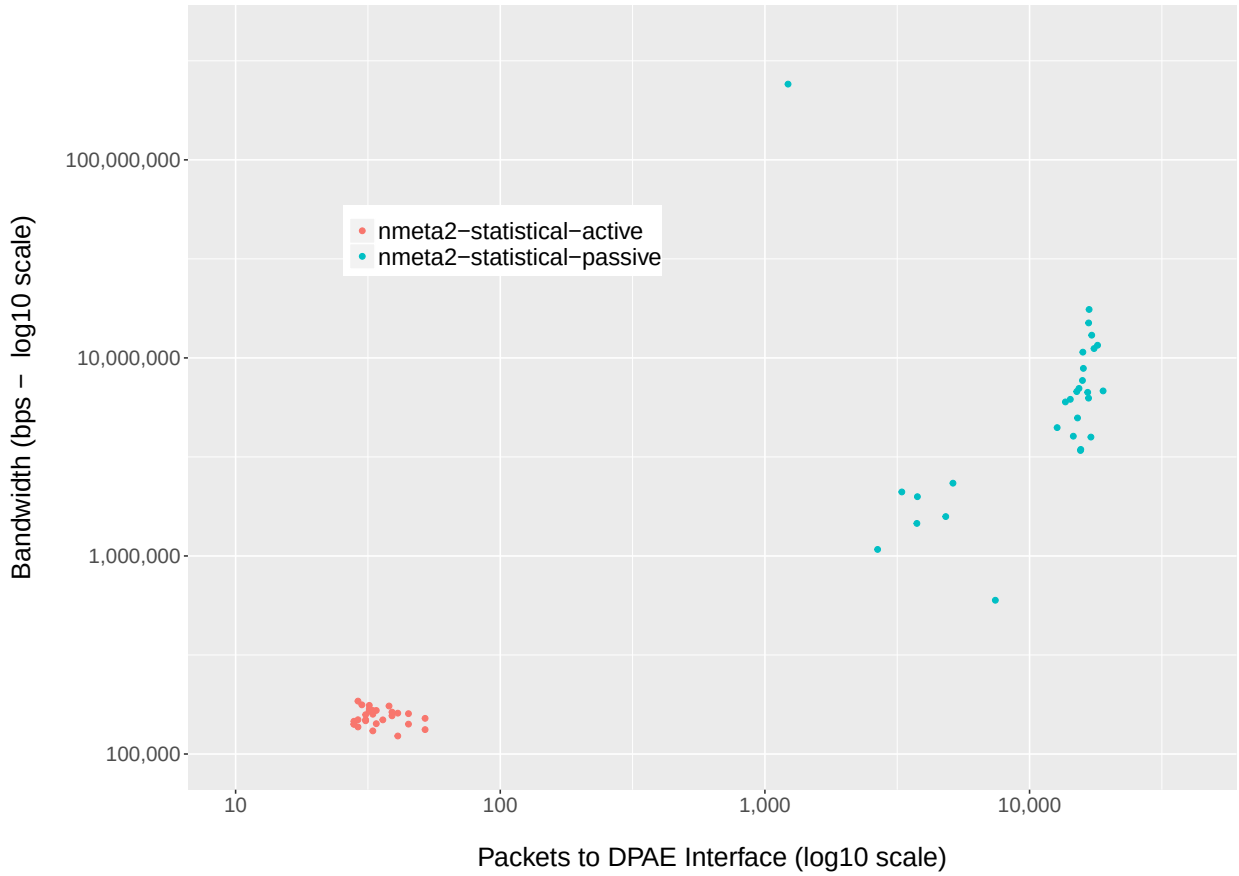


Figure 6.10: Nmeta2 Statistical Classifier - Iperf Average Bandwidth vs TC Packets to DPAE by Mode (source: Table A.4)

of magnitude fewer packets to the DPAE. This however, as per previous charts, is at the cost of NFPS load initial forwarding performance.

Active mode in our prototype is not fully synchronous; packets are returned to the switch in parallel to any advice to the control plane regarding classification and subsequent treatment. A fully synchronous design that waited to forward the packet until after confirmation from the control plane would improve treatment compliance, but at the cost of performance.

We thus have a trade-off for online use cases between the effectiveness of treatment and the forwarding performance. The effects of this trade-off can be mitigated by a) using TC policy to reduce the scope of packets sent to the DPAE for classification, b) minimising network latency between the DPAE, controller and switch and c) ensuring sufficient processing speed on the DPAE, controller and switch to handle peak load. Note that the performance impact on elephant flows is not significant, as forwarding occurs at the natural rate of the switch once suppression FEs are applied.

6.6 Summary

In this chapter, we have demonstrated that full TC services on SDN can be designed to scale, however there are trade-offs that must be considered. Our prototype solution showed performance improvements, as well as ability to scale separately to the controller and switch components. The prototype ran in a virtualised environment, with all flows to the DPAE, and employed an interpreted language (all worst case scenarios), so has considerable scope for further optimisation. Furthermore, our prototype solution proves the viability of building scalable TC atop unmodified OpenFlow 1.3, with only the addition of compute nodes running commoditised Linux.

Chapter 7

Conclusion

In this chapter, we draw conclusions and discuss areas for future work.

We have shown that full TC services over standard SDN architecture, where packets requiring application services must pass via the control plane, does not scale. Sending of packets that require TC inspection beyond the capabilities of the switch to a TC application via the control plane causes resource exhaustion on the control plane CPU, and this in-turn causes severe degradation of network performance. Our tests showed a new flow per second load rate in excess of a relatively low threshold, causes severe performance degradation for new flows. This is caused by per-packet TC workload on the controller and application, resulting in controller CPU exhaustion. Additionally, we showed that this constraint is detrimental to the timeliness of other control plane functions, in particular that of MAC learning.

We proposed a minor change to the standard SDN architecture to solve this problem by retaining the majority of TC workload in the data plane. We extend the SDN architecture, introducing a new component to the data plane, for offload of per-packet processing. This new component, the *Data Plane Auxiliary Engine* (DPAE), can be scaled separately to other compo-

nents. It also brings other advantages, freeing the switch from supporting a prohibitively large feature set, and providing extensibility to functions outside of TC.

Our solution showed performance improvements, as well as ability to scale, and ran in a virtualised environment with all flows to the DPAE and employed an interpreted language (all worst case scenarios), so should have considerable scope for optimisation. Furthermore, our prototype solution proves the viability of building scalable TC atop unmodified OpenFlow 1.3, with only the addition of compute nodes running commoditised Linux. Our solution also encourages innovation by making selected full packets available, in software, for other purposes through an extensible design.

Our prototype solution is simple to deploy as it does not require modifications to the OpenFlow protocol, nor does it require specialised hardware. Our solution leverages Linux on commodity x86 hardware, with the necessary logic in software. We reuse existing software components where possible. The compatibility of our solution with existing standards is a boon for manufacturers of OpenFlow hardware switches, as they are not required to add complexity to their hardware to support modifications to the OpenFlow standard. This in turn is a boon for operators of SDN systems as they have more choice of compatible hardware switches, and can scale other components on commodity compute hardware, or on compute virtualisation platforms.

We have demonstrated that full TC in SDN can be designed to scale, however there are trade-offs that must be considered.

The primary trade-off that we identified is between the effectiveness of treatment and initial forwarding performance. Online use cases that have a requirement for deterministic treatment must forego initial forwarding

performance while a TC determination is made and actions taken.

The effects of this trade-off can be mitigated by a) using TC policy to reduce the scope of packets sent to the DPAE for classification, b) minimising network latency between the DPAE, controller and switch and c) ensuring sufficient processing speed on the DPAE, controller and switch to handle peak load. Note that the performance impact on elephant flows is not significant, as forwarding occurs at the natural rate of the switch once suppression FEs are applied.

7.1 Future Work

We conclude this thesis by discussing potential areas of future work to extend our research.

In future work, we will investigate the sharing and synchronization of flow information across multiple DPAE. The current method of using a simple threshold to identify elephant flows can be enhanced with SDN-based heavy hitter detection techniques [44] to enable dynamic suppression algorithms that optimise resource usage based on current conditions. We intend to introduce security and availability features to our design, as well as improving the TC features and usability of the solution.

Appendices

Appendix A

Result Data

In this appendix, we provide information on the sources of result data used in the thesis.

Attribute	**Value**
Test Script	nmeta2-nfps-combined1000.py
Test Environment	Bordeaux
Repetitions	12 x 5 interleaved test types
Results File	nmeta2-combined-bordeaux-20160720225417.tar.gz
nmeta2 Version	0.3.4

Table A.1: NFPS 1000 Result Data Parameters

Attribute	Value
Test Script	multi-switch-nmeta2active-nfps-combined1000.py
Test Environment	Bordeaux
Repetitions	3 x 5 variations of numbers of switches in path
Results File	multi-switch-nmeta2active-nfps-combined1000-bordeaux-20160722131204.tar.gz
nmeta2 Version	0.3.4

Table A.2: Multi-Switch NFPS 1000 nmeta2 Active Mode Result Data Parameters

Attribute	Value
Test Script	performance-no-load-tests.py
Test Environment	Bordeaux
Repetitions	3 x 5 test types
Results File	performance-no-load-bordeaux-20160723154025.tar.gz
nmeta2 Version	0.3.4

Table A.3: Performance No Load Tests

Attribute	Value
Test Script	tc-timely-noload-statistical.py
Test Environment	Bordeaux
Repetitions	30 x 2 test types
Results File	tc-timely-noload-statistical-bordeaux-20160716115444.tar.gz
nmeta2 Version	0.3.3

Table A.4: TC Timeliness No Load Statistical Classifier

Attribute	**Value**
Test Script	cp-timely-load-statistical.py
Test Environment	Bordeaux
Repetitions	3 x 13 load levels x 4 test types
Results File	control-plane-timeliness-bordeaux-20160725203638.tar.gz
nmeta2 Version	0.3.4

Table A.5: Control Plane Timeliness No Load Statistical Classifier

Appendix B

New Flow Load Generator - filt

In this appendix, we provide details on how we developed filt, a new flow load generator, and the performance and accuracy challenges that we faced.

In filt, an outer loop increments the target rate by a set amount at predefined intervals, while an inner loop sends a single new flow packet per iteration. The algorithm must calculate how long to sleep for before continuing with the inner loop. Initially, it was assumed that a *basic* algorithm would suffice for generating flow load at the correct rate, as per the equation below:

$$d = \frac{1}{x}$$

Where:

- d is the sleep time in seconds
- x is the target rate

Tests in the virtual environment showed the simple flow rate algorithm underperformed on rate with increasing rate variability as target rate increased, and appeared to reach a ceiling beyond which more rate was not

achievable. This was initially assumed to be a limitation of the virtual environment. Physical environment tests however generated a very similar result set, indicating the bottleneck to be the algorithm, not the environment. The algorithm was updated to account for overhead time in the inner loop, which becomes more significant as target rate increases. The overhead is variable due to operating system resource demands made by other processes, so an algorithm is required that adjusts d to compensate, as per Figure B.1.

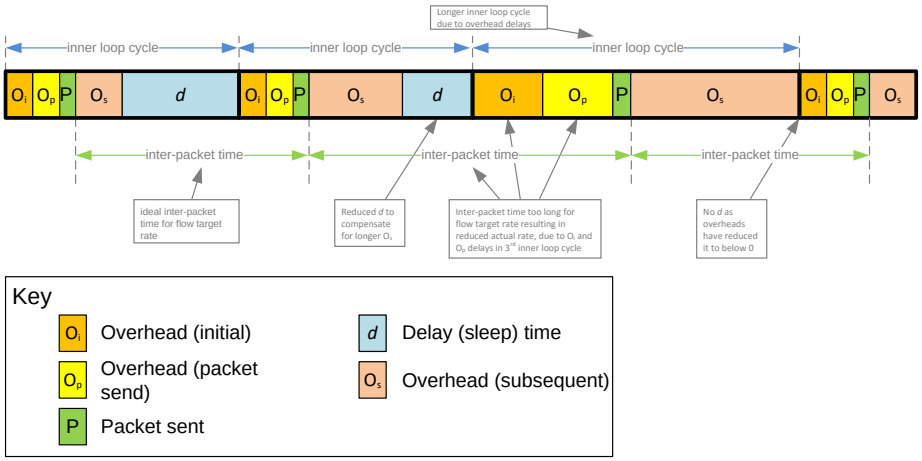


Figure B.1: Consistent Packet Rate Challenges

A new algorithm, called the *make-good* algorithm, was developed:

$$d = \max\left(0, \frac{t - \frac{1}{x}}{xi - q}\right)$$

Where:

- d is the sleep time in seconds
- x is the target rate
- i is the interval in seconds between flow rate increases

- q is the number of flows started in the interval
- t is the time remaining in the interval in seconds
- h is the average overhead time in seconds

A single inter-packet interval ($\frac{1}{x}$) is subtracted from the remaining time in the interval t, to compensate for packets being sent at the start of a cycle, to avoid finishing the outer loop with a packet send. The adjusted time is divided by the outstanding packets to be sent in the interval $((xi) - q)$ to give an ideal inter-packet time for the remaining interval. This time has overhead h subtracted so that the sleep time allows for the overhead.

Testing showed the make-good algorithm significantly outperformed the simple flow rate algorithm. A potential weakness however, of the make-good algorithm, is that it will attempt to lift the average rate of the interval if required by sending at a higher rate. This behaviour may not be desirable when testing requires an upper limit on the maximum rate to avoid overdriving the system earlier than the reported actual rate.

An alternative algorithm, christened *flat-top* as it is less likely to overdrive the output rate, is shown below:

$$d = \max\left(0, \frac{1}{x} - j\right)$$

Where:

- d is the sleep time in seconds
- x is the target rate
- j is the *minimum* overhead time in seconds

This equation is simpler than make-good, as it does not take into account the remaining time or packets to send in the interval, and instead focuses on calculating the correct sleep delay based solely on the target rate and measured overhead. The relative performance of all three algorithms in the virtual environment is shown in Figure B.2 [1].

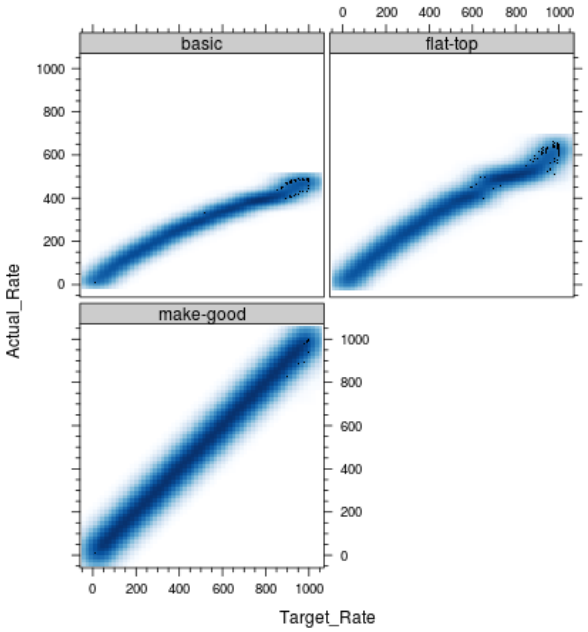


Figure B.2: filt target vs actual rates by algorithm

An ideal performance is a bottom-left to top-right line with no variability. We see the basic algorithm fall further below the target rate as the rate increases. The basic algorithm reaches a plateau around 400 NFPS.

The flat-top algorithm starts well, but drifts below target rate once past 200 NFPS and deteriorates markedly from 400 NFPS.

The make-good algorithm exhibits good behaviour up to 800 NFPS, after which point the variability increases.

[1] Tests in virtual environment, 30 tests of each algorithm type, data set 20150912122339-filt-unit-tests-30x3-types.tar

We decided to investigate sleep time accuracy in the virtual environment as a contributor to variability and rate drop-off and found that sleep calls always ran longer than the requested time period, and there was considerable variability to the oversleep amount, as per Figure B.3.

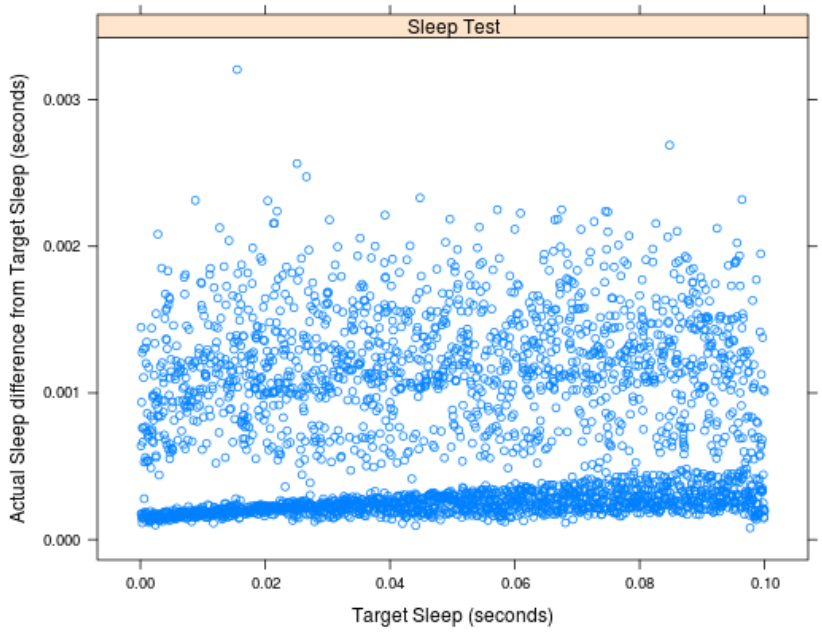


Figure B.3: Absolute Sleep Error by Target Sleep Time

The impact of oversleep on the interpacket delay accuracy becomes more significant as the sleep time decreases, as per Figure B.4, where percentage error is represented on the y-axis

When viewed as a percentage, we see sleep accuracy rapidly degrade below 5 milliseconds in the virtual environment, explaining the variability in the filt results at higher packet rates.

We decided to use the make-good algorithm for our tests due to its superior performance over the other two algorithms. It does have a risk of overdriving the rate, however we manage this risk by maintaining small increment intervals.

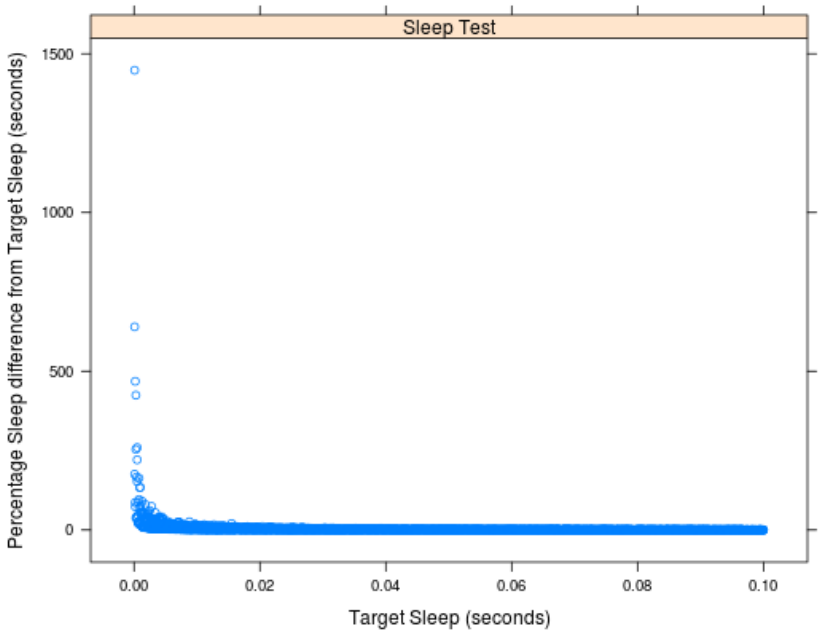


Figure B.4: Percentage Sleep Error by Target Sleep Time

Note that it should be possible to create a better performing algorithm, based on our knowledge of sleep time accuracy, however we chose the make-good algorithm for generating our NFPS test load as it has sufficient accuracy for our purposes at loads below 800 NFPS. Also, packet rate algorithms are not the core contribution of this thesis. We leave this subject open for future work. A researcher in this field may also choose to investigate use of a real-time operating system in place of Ubuntu, or a field-programmable gate array (FPGA), to provide better scheduling (sleep) accuracy.

We have since found prior work [45] that examined packet generator accuracy. The mechanism that we use to cater for oversleep in the make-good algorithm, is referred to as an *Inter Departure Time (IDT) recovery mechanism* in their paper. They observe, as we did, that an IDT recovery mechanism allows average throughput to more closely meet the target rate, but at a cost of reduced accuracy in packet distribution.

Bibliography

[1] Open Networking Foundation, "Openflow switch specification version 1.3.5. [online]." https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.3.5.pdf, April 2015.

[2] M. Hayes, "Quality of service classification mechanisms for IoT," tech. rep., Wellington, New Zealand, 2013.

[3] M. Hayes, "Traffic classification in enterprise networks within the era of IoT. technical report ecstr 14-06," tech. rep., Wellington, New Zealand, November, 2014.

[4] B. Ng, M. Hayes, and W. K. Seah, "Developing a traffic classification platform for enterprise networks with SDN: Experiences & lessons learned," in *Proceedings of IFIP Networking 2015*, (Toulouse, France), May 2015. http://dl.ifip.org/db/conf/networking/networking2015/1570065779.pdf.

[5] M. Hayes, "nmeta github repository [online]." https://github.com/mattjhayes/nmeta/, May 2015.

[6] B. J. van Asten, N. L. van Adrichem, and F. A. Kuipers, "Scalability and resilience of software-defined networking: An overview. [online]." http://arxiv.org/pdf/1408.6760.pdf, August 2014.

[7] A. Tootoonchian and Y. Ganjali, "Hyperflow: A distributed control plane for openflow," in *USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE)*, (San Jose), 2010.

[8] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathany, Y. Iwataz, H. Inouez, T. Hamaz, and S. Shenker, "Onix: A distributed control platform for large-scale production networks," in *OSDI Vol 10*, (Vancouver, BC, Canada), pp. 1–6, 2010.

[9] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "Devoflow: Scaling flow management for high-performance networks," in *Proceedings of the ACM SIGCOMM 2011 conference*, (Toronto, Ontario, Canada), 2011.

[10] S. Hassas Yeganeh and Y. Ganjali, "Kandoo: a framework for efficient and scalable offloading of control applications," in *Proceedings of the first workshop on Hot topics in software defined networks*, (Helsinki, Finland), pp. 19–24, August, 2012.

[11] R. Sherwood, G. Gibb, K. K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, "Flowvisor: A network virtualization layer," tech. rep., 2009.

[12] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood, "On controller performance in software-defined networks," in *USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE)*, (San Jose, CA, USA), April, 2012.

[13] E. Ng, "Maestro: A system for scalable openflow control," tech. rep., TSEN Maestro-Technical Report TR10-08, Rice University, 2010.

[14] A. Shalimov, D. Zuikov, D. Zimarina, V. Pashkov, and R. Smeliansky, "Advanced study of sdn/openflow controllers," in *Proceedings of the*

9th Central & Eastern European Software Engineering Conference in Russia, (Moscow, Russian Federation), October, 2013.

[15] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantzy, B. OConnory, P. Radoslavovy, W. Snowy, and G. Parulkar, "Onos: towards an open, distributed sdn os," in *Proceedings of the third workshop on Hot topics in software defined networking*, (Chicago, IL, USA), pp. 1–6, August, 2014.

[16] Y.-D. Lin, P.-C. Lin, C.-H. Yeh, Y.-C. Wang, and Y.-C. Lai, "An extended sdn architecture for network function virtualization with a case study on intrusion prevention," *Network, IEEE*, vol. 29, no. 3, pp. 48–53, 2015.

[17] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. Gude, N. McKeown, and S. Shenker, "Rethinking enterprise network control," *IEEE/ACM Transactions on Networking (TON)*, vol. 17, pp. 1270–1283, 2009.

[18] T. Benson, A. Akella, and D. Maltz, "Network traffic characteristics of data centers in the wild," in *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, (Melbourne, Australia), pp. 267–280, ACM, 2010.

[19] Z. A. Qazi, J. Lee, T. Jin, G. Bellala, M. Arndt, and G. Noubir, "Application-awareness in sdn," in *ACM SIGCOMM Computer Communication Review*, vol. 43, (Hong Kong, China), pp. 487–488, ACM, 2013.

[20] S. Li, E. Murray, and Y. Luo, *Programmable Network Traffic Classification with OpenFlow Extensions*, pp. 269–299. Boca Raton, FL, USA: CRC Press, 2014.

[21] W. de Donato, A. Pescape, and A. Dainotti, "Traffic identification engine: an open platform for traffic classification," *Network, IEEE*, vol. 28, no. 2, pp. 56–64, 2014.

[22] G. Bianchi, M. Bonola, A. Capone, and C. Cascone, "Openstate: programming platform-independent stateful openflow applications inside the switch," *ACM SIGCOMM Computer Communication Review*, vol. 44(2), pp. 44–51, 2014.

[23] S. Pontarelli, G. Bianchi, M. Bonola, A. Capone, and C. Cascone, "Stateful openflow: Hardware proof of concept," in *In High Performance Switching and Routing (HPSR), 2015 IEEE 16th International Conference on*, (Budapest, Hungary), July, 2015.

[24] Ryu SDN Framework, "Ryu sdn framework. [online]." `https://osrg.github.io/ryu/`, June 2016.

[25] Oracle Corporation, "Oracle VM VirtualBox. [online]." `https://www.virtualbox.org/`, October 2015.

[26] Ansible Inc., "Ansible is Simple IT Automation. [online]." `http://www.ansible.com/`, September 2015.

[27] R Core Team, "R: A Language and Environment for Statistical Computing. [online]." `https://www.R-project.org/`, September 2015.

[28] Hayes, M, "HTTP Object Retrieval Test (hort) GitHub Repository. [online]." `https://github.com/mattjhayes/hort`, October 2015.

[29] K. Reitz, "Requests: HTTP for Humans. [online]." `http://docs.python-requests.org/en/latest/`, October 2015.

[30] M. Kobayashi, S. Seetharaman, G. Parulkar, G. Appenzeller, J. Little, J. Van Reijendam, P. Weissmann, and N. McKeown, "Maturing of openflow and software-defined networking through deployments," *Computer Networks*, vol. 61, pp. 151–175, March 2014.

[31] Hayes, M, "Flow Incremental Load Test (filt) GitHub Repository. [online]." `https://github.com/mattjhayes/filt`, October 2015.

[32] P. Biondi, "Scapy [online]." `http://www.secdev.org/projects/scapy/`, October 2015.

[33] G. Rodola, "psutil documentation. [online]." `https://pythonhosted.org/psutil/`, October 2015.

[34] Hayes, M, "Measure Operating System Performance (mosp) GitHub Repository. [online]." `https://github.com/mattjhayes/mosp`, October 2015.

[35] Open Networking Foundation, "ONF overview. [online]." `https://www.opennetworking.org/about/onf-overview`, June 2016.

[36] Open Networking Foundation, "Sdn architecture 1.0 [online]." `https://www.opennetworking.org/images/stories/downloads/sdn-resources/technical-reports/TR_SDN_ARCH_1.0_06062014.pdf`, June 2014.

[37] Open Networking Foundation, "Sdn architecture 1.1 [online]." `https://www.opennetworking.org/images/stories/downloads/sdn-resources/technical-reports/TR-521_SDN_Architecture_issue_1.1.pdf`, February 2016.

[38] Open Networking Foundation, "Openflow table type patterns." https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/OpenFlowAugust 2014.

[39] M. Hayes, "Github - mattjhayes/nmeta2." https://github.com/mattjhayes/nmeta2, May 2016.

[40] M. Hayes, "Github - mattjhayes/nmeta2dpae." https://github.com/mattjhayes/nmeta2dpae, May 2016.

[41] T. Mori, M. Uchida, R. Kawahara, J. Pan, and S. Goto, "Identifying elephant flows through periodically sampled packets," in *In Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, (Taormina, Sicily, Italy), pp. 115–120, October. 2004.

[42] S. Russell, "Openflow 1.2 switch app with multiple tables. [online]." `http://pieknywidok.blogspot.co.nz/2013/07/openflow-12-switch-app-with-multiple.html`, July 2013.

[43] Open Networking Foundation, "The benefits of multiple flow tables and ttps. [online]." `https://www.opennetworking.org/images/stories/downloads/sdn-resources/technical-reports/TR_Multiple_Flow_Tables_and_TTPs.pdf`, February 2015.

[44] L. Yang, B. Ng, and W. K. G. Seah, "Heavy Hitter Detection and Identification in Software Defined Networking," in *Proceedings of the 25th International Conference on Computer Communication and Networks (ICCCN)*, (Waikoloa, HI, USA.), 1-4 August 2016.

[45] A. Botta, A. Dainotti, and A. Pescapé, "Do you trust your software-based traffic generator?," *Communications Magazine, IEEE*, vol. 48, no. 9, pp. 158–165, 2010.