Supporting SDN and OpenFlow within DIFFUSE

Dzuy Pham*, Jason But
Centre for Advanced Internet Architectures, Technical Report 160429A
Swinburne University of Technology
Melbourne, Australia
dhpham@swin.edu.au, jbut@swin.edu.au

***Abstract*—Software Defined Networking (SDN) has the potential to change the way in which networks are managed through the use of centralised control. DIFFUSE is an existing system that can dynamically classify traffic flows in real-time using Machine Learning based techniques, and subsequently (de-)prioritise those flows. We developed a RYU Action Node (RAN) as an OpenFlow compatible DIFFUSE Action Node to allow for dynamic traffic management in an SDN. Our tests have verified the correct functionality of the RAN in combination with DIFFUSE and live traffic.**

I. Introduction

Software Defined Networking (SDN) [1,2] is becoming more mainstream, with many hardware vendors releasing SDN compatible products. The concept behind SDN architecture is to abstract the network management into layers, moving the responsibility for management of infrastructure to centralised control infrastructure.

SDN has the potential to provide an agile network that is liberated from proprietary hardware. This also allows for dynamically customisable networking which can be managed more efficiently and optimised for better performance.

DIFFUSE (Distributed Firewall and Flow-shaper Using Statistical Evidence) [3], a network prioritisation scheme constructed around ML-based techniques was developed to segregate network traffic by 5-tuple inspection into classes and consequently deploy prioritisation. In its current state, DIFFUSE is currently constrained to FreeBSD [3] and OpenWRT [4] systems. [5]

This report documents our work to extend DIFFUSE to integrate with RYU [6] – an SDN framework controller. This will allow dynamic management of network bandwidth and improved responsiveness to network behaviour based on automated traffic classification by the DIFFUSE Classifier Nodes.

*This work was performed while the author was an undergraduate summer intern under the supervision of Dr Jason But.

We developed a RYU Action Node (RAN) to parse DIFFUSE messages and subsequently program an SDN switch. The RAN also manages the deployment of classification based prioritisation rules and was tested using the DIFFUSE Fake Classifier Node (FCN) [5].

This report is organised as follows. The background information on Software Defined Networking, RYU and OpenFlow can be found in section II. Section III outlines some more information on DIFFUSE and the problems with integrating DIFFUSE to SDN. Section IV contains an overview of the RYU Action Node and was verified in Section V. Section VI concludes with some notes and future plans for the complete DIFFUSE integration to SDN.

II. Background

The primary concept behind Software Defined Networking (SDN) is to separate the control and data planes within the network, allowing a central control system to manage multiple network switches using a well-defined API. SDN Applications can use centralised logic to determine outcomes and use the Northbound API to communicate network control decisions to the SDN Controller. The outcomes are then managed by the controller which uses the Southbound API to program the flow tables on the switches. (see Figure 1)

A. *Software Defined Networking*

Software Defined Networking provides a modern approach towards networking architecture, enabling flexibility and ease of use through software abstractions. Traditionally, the Control Plane and the Data Plane of a piece of network infrastructure resides within the same switch/router. The Control Plane handles the decision making and programs the Data Plane to forward those packets accordingly. This approach results in devices that are restricted by the manufacturer's firmware. SDN seeks to overcome these limitations by separating the Data

Plane from the Control Plane in order to have complete control over network behaviour.

The current methodology within SDN is to split the network into three distinct layers [1,2]:

Application Layer - Hosts all the systems which are used to program the SDN

Control Layer - Contains the controllers which manage the SDN and programs the switches

Infrastructure Layer - Contains the switches which forwards the data programmed by controller.

Advantages of this abstraction include:

- A centralised controller capable of configuring multiple connected switches
- A switch that is capable of being programmed by multiple controllers
- A dynamic network that is able to respond to different network conditions
- Highly customised network implementations, as the Control Layer is no longer bound to proprietary firmware

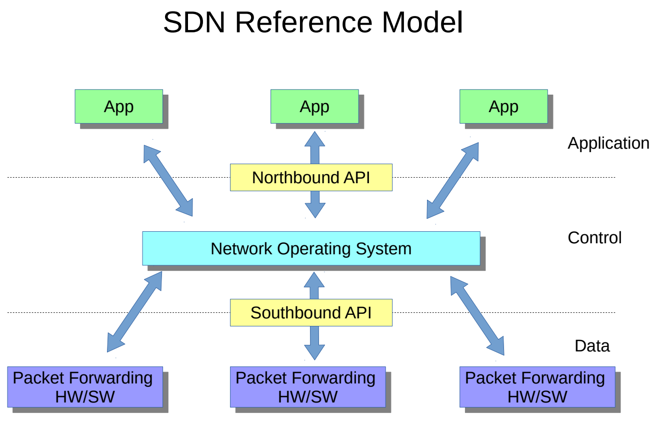

Figure 1. SDN Reference Model

B. *RYU and OpenFlow*

OpenFlow [7] is a well-defined Southbound API (see Figure 1) protocol commonly used in many SDN implementations to connect the Control and Infrastructure Layers. As OpenFlow standardises the communication between the Control and Infrastructure Layer, we can use any OpenFlow based controller.

RYU [6] is an open source Northbound API (See Figure 1) which allows the deployment of multiple applications within a common framework. The RYU framework contains a number of OpenFlow software components that are easily accessible by SDN applications to control the network. This allows a RYU controller to manage multiple applications as well as monitor any messages sent by the switches from the Control Layer.

III. DIFFUSE

DIFFUSE [3] is a system which uses statistical analysis and machine based learning techniques to first classify, and then prioritise selected flows in the network. Diffuse is split into two main components, the Classifier Node (CN) and the Action Node (AN).

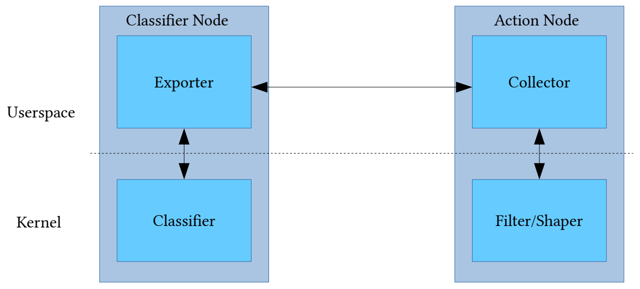

Figure 2. DIFFUSE Components

A. *Classifier Node*

The CN classifies flows using ML-based techniques based on statistical analysis of similar flows.

The CN runs within the kernel using IPFW [8] to collect packets for processing.

Once a classification is made, a Remote Action Protocol (RAP) [3] packet is created to be sent by the exporter to a collector module in an AN to be acted upon.

The DIFFUSE Fake Classifier Node (FCN) [5], is able to send customisable Remote Action Protocol (RAP) [3] messages to ANs. This allows more thorough testing of a DIFFUSE deployment.

Currently CN implementations only exist for FreeBSD [3] and OpenWRT [4], while the FCN operates on any platform supporting Python 3 [9].

B. *Action Node*

The AN is made up of two components, 1) the collector module which receives RAP messages from a CN, and 2) the filter/shaper which implements an action.

The action taken by the filter/shaper is based on the flow class as reported by the CN, and instructions based on the contents of an actions file that is loaded during the AN setup. The resultant action is then applied to Dummynet [10] in order to create the rule in the IPFW [8] table. All subsequent packets of this flow will now be (de)prioritised based on the classification.

Current AN implementations also only exist for FreeBSD [3] and OpenWRT [4].

C. An OpenFlow based DIFFUSE Action Node

The RYU Action Node (RAN) (see Section IV) was developed to extend DIFFUSE to support the SDN/OpenFlow framework. This will allow for full DIFFUSE ML-based classification using SDN to deploy prioritisation.

The current DIFFUSE program prioritises traffic using a software-based approach (IPFW) and is therefore performance limited. Extending DIFFUSE to support SDN will allow hardware-based prioritisation through direct configuration using the OpenFlow API. This also allows DIFFUSE to directly program multiple switches from a single AN.

By extending DIFFUSE to support OpenFlow, we can combine the benefits of automatic traffic classification with high performance hardware-based traffic prioritisation.

IV. Ryu Action Node

The RYU Action Node (RAN) is coded as an extension of the existing RYU simple switch code [11] provided by the RYU SDN Framework [6]. The simple switch program handles forwarding of packets and learning MAC addresses of connected hosts. Behaving as a regular switch, received packets are either forwarded directly to the destination or flooded on all ports.

The RAN runs alongside the simple switch listening for RAP flow information packets [3] and programs the hardware switch accordingly. Since the RAN is coded as an application using the RYU framework, other RYU applications can also be used in conjunction with the RAN.

A. Overview

The RAN is an SDN application that is able to decode RAP messages from DIFFUSE [3]. It utilises the RYU software components in order to both monitor and modify the SDN switch. In order to modify the switch, the RAN must first establish a connection with the switch. This is accomplished by running the RYU manager command, where all the RYU controller applications, including the RAN, will be instantiated. The RAN is able to accept RAP packets, decode the message and deploy an action.

The RAN currently only supports one flow information within the packet at a time, unlike the original AN which supports multiple flow messages within the same packet.

B. Design of Ryu Action Node

The RAN runs as an additional thread outside of the simple switch program, binding onto TCP port 5000 to listen for RAP flow information packets. When a RAP message is received, the contents of the message is decoded where a 5-tuple flow ID and flow class is extracted. The class is used to specify which action to take (see Section IV-D). This action, along with the 5-tuple information, is used to create an OpenFlow 1.3 message using RYU's API to deploy the action as a rule within the SDN hardware. All future traffic which matches the rule will be subject to the SDN flow table entry.

C. Supporting Multiple Prioritisation Techniques

The RAN supports a limited number of prioritisation techniques due to the hardware limitations on switches that provide OpenFlow. We explored the use of both queuing and metering within the RAN. The RAN is flexible in that it allows the utilisation of either queues, meters, or both. These options are specified in the RAN configuration file and can be specified on a per-class basis.

1) Queues: Each physical switch port can support up to eight Queues (also known as slices), where each queue is given a portion of the available bandwidth. This is achieved by either defining the maximum bandwidth of each queue or giving a minimum bandwidth to the queue which it will try to abide by. Using maximum bandwidth settings will throttle a queue, limiting the maximum achievable bandwidth. Using minimum bandwidth will attempt to limit the throughput of other queues such that queues with defined minimum bandwidth can be achieved.

2) Meters: Metering works by monitoring the aggregate bandwidth of each flow attached to it. When the threshold has been reached, the meter will perform the nominated action on the given flows. Actions supported by the Pica8 OVS Switch using Open vSwitch 2.3.0 are Drop and DSCP Remark.

Drop meter Limits the bandwidth of any aggregate flow that exceeds the threshold and discard any packets to maintain bandwidth limits.

DSCP Remark Changes the Differentiated Services Code Point (DSCP) [12] in the packet header.

DSCP Remark was not tested due to the lack of a configured DiffServ network [13]. In a DiffServ, each queue would be assigned DSCP numbers with different configurations. If a flow were to exceed the threshold set for their flow they would have their DSCP number increased by a certain amount and consequently this may place the flow into another queue with different properties.

D. Configuration File Format

The RAN actions are defined by entries in the RAN Configuration File. The file is formatted using the standard INI notation [14], where INI sections are defined per traffic class (as parameters specified by the Classifier Node) and action details for that class are specified within each corresponding section.

The RAN supports a "`default`" (case sensitive) class which is deployed when the CN sends flow information for an unspecified class. Alternatively, the RAN will allocate unknown classes to queue 0 with no other parameters.

For each class, traffic can be allocated to queues (0-7), multiple classes may be allocated to each queue. Classes not assigned to a queue will be allocated to queue 0. Queue configuration does not support bandwidth settings as this can only be specified manually either using a CLI (such as the RYU REST API [15]) or directly at the switch.

Meters are able to be assigned to each class and will activate when the aggregate bandwidth of each class assigned to a meter surpasses the set threshold.

It is also possible to combine a queue with an SDN meter by different settings in the configuration file. Parameters that can be set on a meter include

- Type - Either Drop or DSCP Remark
  - Drop will limit the maximum bandwidth through a meter when active
  - DSCP Remark will increase the DSCP number when active
- Rate - The bandwidth threshold
- Dscp - The DSCP amount that will be added to the existing DSCP number

Up to 100 meters can be assigned a unique numeric meterid. If a single meter is used across multiple classes, the actual configuration will be taken from the first instance in the configuration file.

An example configuration file is provided in Figure 3. In this example we have 3 classes a **Web**, **Voip** and a **default** class. The web class has the following parameters: any traffic in the Web class is placed in queue 0 with a meter also attached to it numbered 1. The threshold bandwidth at which the meter will activate is 100000 bits/s and the meter type will be a drop meter. Traffic classed as Voip will be placed into queue 1 with no meter attached. The default class will place traffic into queue 0, and meter 2 will be set as DSCP. The threshold bandwidth at which it activates is 50000 bits/s, it will increase the DSCP number by 12, if active, for any traffic classed as default.

The available parameters and their allowed values are listed in Table I.

Table I
CONFIGURATION PARAMETERS

Field	Description
Queue	Number between 0-7
MeterID	Number between 1-100
Rate	Maximum bandwidth in bits per second
Type	Drop or DSCP
DSCP	Any numeric value

```
[Web]
queue = 0
meterid = 1
rate = 100000
type = drop

[Voip]
queue = 1

[default]
queue = 0
meterid = 2
rate = 50000
type = dscp
dscp = 12
```

Figure 3. Example Configuration File

V. VERIFICATION

To be useful, the RAN needs to be capable of properly processing incoming RAP messages, and subsequently correctly program the SDN hardware.

A. Base Testing of the RAN

The DIFFUSE FCN [5] is used to test the RAN. There are two steps involved in confirming that the RAN is functioning correctly, the first requires verifying that the RAN can correctly process RAP packets and then implement appropriate rules on the SDN switch using OpenFlow. In this test there will be no live traffic flowing through the switch, and verification is via visual inspection of the switch output table following the receipt of the flow message from the FCN.

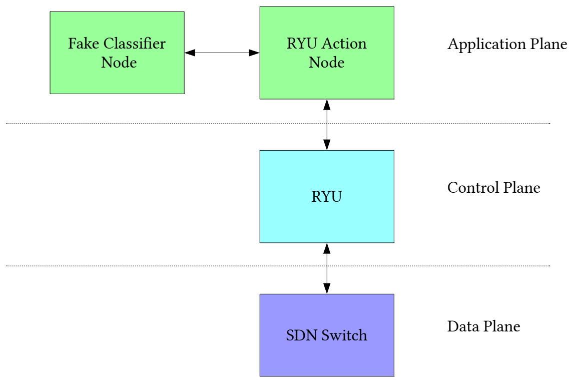

Figure 4. SDN Overview

For the purposes of this test we configure the RAN using the example configuration file in Figure 3. The configuration file defines three classes **Web**, **Voip** and **default**, each implementing the following prioritisation settings within the SDN switch:

Web Traffic is place into queue 0 using meter 1, configured as a drop meter.

Voip Traffic is not managed using meters but is placed into queue 1

Default Traffic is placed into queue 0 using meter 2, configured as a DSCP meter.

These tests will not confirm if traffic is being correctly processed by the switch but instead confirm that the appropriate changes are made to the output table based upon messages from the FCN.

To test the RAN, we built a testbed as per Figure 6. An FCN was instantiated on a PC loaded with FreeBSD 9.0 and the RAN & simple switch application loaded onto the RYU controller on a virtual machine running Ubuntu 13.34. The RYU controller was connected to a Pica8 OVS Switch running on Open vSwitch 2.3.0. The Switch was configured with the commands in Figure 5. As rules are added/removed, we can see the changes being made to the switch flow table.

```
ovs-vsctl set Bridge br0 protocols=OpenFlow13
ovs-vsctl set-manager ptcp:6632
ovs-vsctl set-controller br0
tcp:136.186.93.25:6633
```

Figure 5. Switch Configuration

At the start of the experiment, the RAN initialised communications with the SDN switch via the RYU controller, and the SDN Switch rule table was empty.

Three RAP messages were sent using the FCN to the RAN (see Figure 7). Each RAP message contains both

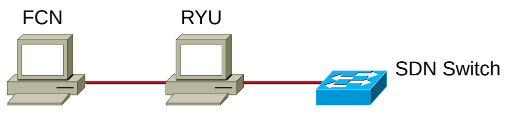

Figure 6. Testbed

a unique 5-tuple and traffic class to ease identification of flows in the Switch table. Furthermore, the last RAP message contains a traffic class not configured in the RAN configuration file.

As the three test RAP messages were sent to the RAN, we can see the switch rule table being populated. Upon completion of all three tests, the rule table contained the information shown in Table II. Further examination of the meters installed on the switch are shown in Figure 8. Through visual inspection, we can see that the meters are being created based on the information in the RAN configuration file and that the rules in the table correctly match the corresponding 5-tuple information and class configuration options.

We note that the FCN message containing the ”`unknown`” class was assigned the ”`default`” class (see Figure 3) in the switch rule table (see Table II). This shows that the default function works as intended and classes that are not specified in the action file are correctly assigned to the ”`default`” class.

```
OFPST_METER_CONFIG reply (OF1.3)
(xid=0x2):
meter=1 kbps bands=
type=drop rate=1000000


meter=2 kbps bands=
type=dscp_remark rate=50000 prec_level=12
```

Figure 8. Pica8 Switch Output Meters

Further testing was done by using the FCN to incrementally remove each flow from the RAN. Upon each FCN execution, the appropriate rule was removed from the SDN switch rule table and any required meters were removed from the meter output.

Finally, the RAN is required to auto-expire rules in the switch rule table if no regular updates are receipt from the FCN. It was confirmed by visual inspection of the SDN table that rules were appropriately removed 60 seconds after the rule was implemented.

At this stage we have verified that the RAN can properly parse RAP messages received from a DIFFUSE CN,

```
FCN CLI INPUT
python fake_cnode.py -i 10.0.0.1 -j 10.0.0.2 -k 8000 -l 8001 -t 60 -c Web -u 6 -x
136.186.93.25 -y 5000 -z tcp -n 20
python fake_cnode.py -i 10.0.0.1 -j 10.0.0.3 -k 5000 -l 80 -t 60 -c Voip -u 17 -x
136.186.93.25 -y 5000 -z tcp -n 21
python fake_cnode.py -i 10.0.0.1 -j 10.0.0.4 -k 66 -l 666 -t 60 -c unknown -u 17 -x
136.186.93.25 -y 5000 -z tcp -n 22
```

Figure 7. FCN Commands

Table II
PICA8 SWITCH OUTPUT TABLE 0 - ADD

Priority	Cookie	Match Fields	Actions	Duration	Packets	Bytes
22	0x0	ipv4_dst=10.0.0.4, hard_timeout=60, ipv4_src=10.0.0.1, idle_timeout=60, eth_type=0x0800, tp_dst=666, ip_proto=17, tp_src=66,	meter:2,set_queue:0,goto_table:1	6.921s	n/a	0
21	0x0	ipv4_dst=10.0.0.3, hard_timeout=60,ipv4_src=10.0.0.1, idle_timeout=60, eth_type=0x0800, tp_dst=80, ip_proto=17, tp_src=5000,	set_queue:1,goto_table:1	9.391s	n/a	0
20	0x0	ipv4_dst=10.0.0.2, hard_timeout=60,ipv4_src=10.0.0.1, idle_timeout=60, eth_type=0x0800, tp_dst=8001, ip_proto=6, tp_src=8000,	meter:1,set_queue:0,goto_table:1	9.970s	n/a	0
0	0x0		goto_table:1	1186.355s	n/a	0

and correctly create and remove rules on a connected OpenFlow switch via the RYU controller.

B. Testing the RAN in an SDN scenario

The second aspect of confirming RAN functionality is to confirm that it functions properly with other OpenFlow applications in the RYU architecture, and that live traffic flows are correctly processed by the switch following implementation of a rule.

The testbed used for this experiment is shown in Figure 8. The FCN is running on a dedicated FreeBSD 9.0 host which is communicating using the DIFFUSE RAP protocol to a RYU controller running on a Ubuntu 13.34 virtual host. The RYU controller is running both our RAN application and the simple switch application as provided within RYU. The SDN switch was a Pica8 OVS Switch running on Open vSwitch 2.3.0. Two virtual hosts running a standard Debian install are connected to the switch in order to generate and sink traffic flows.

Host1 is configured to send a UDP stream to host2 at 10 Mbps using **iperf [16]**. In order to ensure that there are no bottlenecks on our physical hardware or virtual hosts generating traffic, the link speed for each port on the Switch was limited to 10 Mbps. Throughput speed was measured using iperf at a 1 sec interval on host2.

It is expected that the simple switch application running on the RYU controller will correctly result in the switch being programmed to switch traffic between the two test hosts. Following this, the FCN host will be used to program queue and/or meter settings within the SDN switch to manage the UDP flow. This will verify both that the RAN can happily co-exist with other RYU applications, and that the flow rule programmed by the RAN has the desired effect.

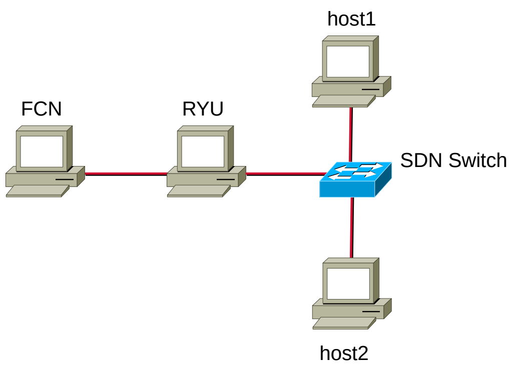

Figure 9. Testbed QoS

1) Queue Test :

In this experiment we are confirming basic functionality in that the queues are behaving as expected. The switch was configured with 6 queues as outlined in Table III. The RAN was then configured with 6 different traffic classes as per Figure 10, with classes 0 through 5 mapping to the corresponding queues.

Table III
BANDWIDTH LIMIT

Queue No.	Max BW(Mbps)	Section in Figure 11
0	Unlimited	1, 7
1	1	2
2	6	3
3	3	4
4	2	5
5	5	6

Initially, the live traffic was classified as **class0**, and was therefore being handled by queue 0. After brief periods, the FCN was used to change the classification of the flow through the sequence **class0, class1, class2, class3, class4, class5, class0**. Throughput measurements from iperf for the duration of this test are shown in Figure 11.

```
[default]
queue = 0
[class0]
queue = 0
[class1]
queue = 1
[class2]
queue = 2
[class3]
queue = 3
[class4]
queue = 4
[class5]
queue = 5
```

Figure 10. RAN Action File II

Initially, the traffic flow was observed to be using nearly all the available bandwidth. This is expected as queue 0 does not implement any bandwidth limitations. We can see during each change in classification, after a short transition period, the achieved throughput appears to be correctly limited by the corresponding queue bandwidth limits. Additionally, during the brief transition period, we can see a spike in throughput which appears to reach the overall bandwidth limit programmed on the port.

While the RAN was simply changing the queue allocated to the traffic via OpenFlow, it appears as though the process of changing the queue results in a two step process of first removing the queue rule from the flow table followed by immediately creating a replacement rule. This might explain the short period whereby the traffic does not appear to be managed. Further investigation is required to confirm this. This artefact is an outcome of the implementation of the SDN switch and not of the RAN controller. Furthermore, the brief period of default performance is equivalent to the outcome, should no prioritisation be in place.

It is also evident that when the queues were not limiting bandwidth, that there are occasional dips in measured throughput. We speculate that this is due to the bandwidth limitation programmed on the switch port being slightly mismatched to the bandwidth limitation employed by iperf, resulting in occasional packet loss due to overflow. In order to confirm this, further tests are required.

This test confirms that flows passing through the switch can be dynamically classified, and subsequently be moved into different queues subject to bandwidth limitations while traffic is flowing. The test also confirms that bandwidth limitation via queues is functional, despite a brief transition period when classification, and therefore queue allocation, changes dynamically.

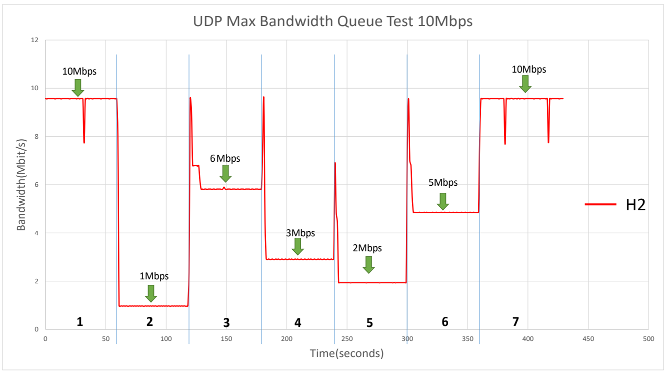

Figure 11. Queue Test Results

2) Meter Test: This experiment is to confirm that the SDN meters are performing as expected. The meters monitor live traffic and drop packets above the bandwidth threshold based on class. Two meters were configured as per Figure 12, both set with meter type drop and using queue 0. The two meters activate at bandwidth thresholds of 3 Mbps and 5 Mbps respectively. Throughput bandwidth was measured on host2 at 1 second intervals using iperf.

Initially traffic is allocated to the default class. Figure 13 shows that all the available bandwidth was being consumed, this behaviour is expected as the default class has no prioritisation schemes attached. As traffic is reclassified to class0 or class1 the output as measured by iperf drops to 3 Mbps and 5 Mbps respectively.

```
[class0]
queue = 0
meterid = 1
type = drop
rate = 3000000

[class1]
queue = 0
meterid = 2
type = drop
rate = 5000000
```

Figure 12. RAN Action File III

Our testing confirms that the RAN can correctly program an SDN switch to use meters and consequently subject live traffic to drop packets at a certain threshold. The measurements also confirm that the drop meter function is working as intended with stable transitions between meters and also output consistent throughput bandwidth.

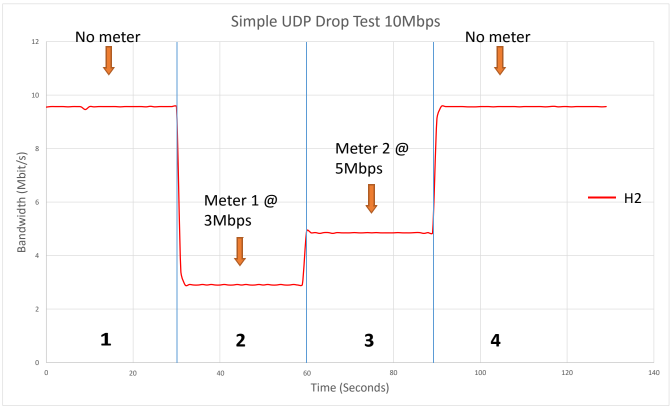

Figure 13. Meter Test Results

VI. Conclusions

This report explores the process of supporting DIFFUSE on SDN hardware, the testing methodology and the outcome of the results to verify the implementation.

To provide SDN-DIFFUSE compatibility, we developed the RYU Action Node to receive and parse DIFFUSE RAP messages. The RYU Action Node was successful at managing the DIFFUSE messages and was able to deploy the correct configured rules to an SDN switch.

Multiple Quality of Service parameters were tested to review the performance of the SDN switch and determine if QoS rules can be implemented dynamically and rapidly. The Queue test determined that flow throughputs were inconsistent when transitioning between queues, exhibiting spikes and drops during the test, whilst the Meter test worked as expected without any abnormal behaviour.

Future work can be done to better understand the unexpected behaviour from the Queue test. The RAN only provides the foundation for DIFFUSE to connect to the SDN network and will required further development on classification and machine learning in SDN to bring us closer to complete DIFFUSE integration.

References

[1] M.-K. Shin, K.-H. Nam, and H.-J. Kim, “Software-defined networking (sdn): A reference architecture and open apis,” in *ICT Convergence (ICTC), 2012 International Conference on*. IEEE, 2012, pp. 360–361.

[2] B. A. Nunes, M. Mendonca, X.-N. Nguyen, K. Obraczka, and T. Turletti, “A survey of software-defined networking: Past, present, and future of programmable networks,” *Communications Surveys & Tutorials, IEEE*, vol. 16, no. 3, pp. 1617–1634, 2014.

[3] S. Zander and G. Armitage, “Design of DIFFUSE v0.4 - Distributed Firewall and Flow-shaper Using Statistical Evidence,” Centre for Advanced Internet Architectures, Swinburne University of Technology, Melbourne, Australia, Tech. Rep. 110704A, 04 July 2011. [Online]. Available: http://caia.swin.edu.au/reports/110704A/CAIA-TR-110704A.pdf

[4] N. Williams and S. Zander, “Real Time Traffic Classification and Prioritisation on a Home Router using DIFFUSE,” Centre for Advanced Internet Architectures, Swinburne University of Technology, Melbourne, Australia, Tech. Rep. 120412A, 12 April 2012. [Online]. Available: http://caia.swin.edu.au/reports/120412A/CAIA-TR-120412A.pdf

[5] D. Pham and J. But, “Developing a Fake Classifier Node for DIFFUSE,” Centre for Advanced Internet Architectures, Swinburne University of Technology, Melbourne, Australia, Tech. Rep. 160422A, 22 April 2016. [Online]. Available: http://caia.swin.edu.au/reports/160422A/CAIA-TR-160422A.pdf

[6] Ryu Framework Community, “Ryu SDN Framework,” 2016. [Online]. Available: http://osrg.github.io/ryu/index.html

[7] B. Pfaff, B. Lantz, B. Heller *et al.*, “Openflow switch specification version 1.3.0,” *Open Networking Foundation*, 2012. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.0.pdf

[8] U. J. Antsilevich, P.-H. Kamp, A. Nash, A. Cobbs, and L. Rizzo, “ipfw – User interface for firewall, traffic shaper, packet scheduler, in-kernel NAT,” *FreeBSD System Manager’s Manual*, 2011. [Online]. Available: https://www.freebsd.org/cgi/man.cgi?query=ipfw&manpath=FreeBSD+9.0-RELEASE

[9] Python Software Foundation, “Python,” 2016. [Online]. Available: https://python.org/

[10] M. Carbone and L. Rizzo, “Dummynet revisited,” *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 2, pp. 12–20, 2010.

[11] Ryu Framework Community, “Simple switch 1.3 source,” 2015. [Online]. Available: https://github.com/osrg/ryu/tree/master/ryu/app

[12] K. Nichols, S. Blake, F. Baker, and D. Black, “Rfc 2474: Definition of the differentiated services field (ds field) in the ipv4 and ipv6 headers, december 1998,” *Obsoletes RFC1455 [47]*, 1998. [Online]. Available: https://www.ietf.org/rfc/rfc2474.txt

[13] D. Grossman, “New terminology and clarifications for diffserv,” 2002. [Online]. Available: https://www.ietf.org/rfc/rfc3260.txt

[14] Python Software Foundation, “Configparser,” 2016. [Online]. Available: https://docs.python.org/3/library/configparser.html

[15] Ryu Development Team, *Ryu Documentation Release 4.1*, 2016. [Online]. Available: https://media.readthedocs.org/pdf/ryu/latest/ryu.pdf

[16] J. Dugan, S. Elliott, B. A. Mah, J. Poskanzer, and K. Prabhum, “iperf3,” 2010. [Online]. Available: https://iperf.fr