Jonathan Vestin

Quality of Service Support for CloudMAC

Subtitle

Master's Thesis of 30 credit points

Date/Term: 00-00-00
Supervisor: Andreas Kassler
Examiner: Donald F. Ross
Serial Number: 0000000
Quality of Service Support for CloudMAC

Jonathan Vestin
This thesis is submitted in partial fulfillment of the requirements for the Bachelors degree in Computer Science. All material in this thesis which is not our own work has been identified and no material is included for which a degree has previously been conferred.

Jonathan Vestin

Advisor: Andreas Kassler

Examiner: Donald F. Ross
Abstract

Software Defined Networking (SDN) is a relatively recent technology, which has seen a rise in popularity. OpenFlow is a very popular SDN technology, but it does not have any support for Quality of Service options. Open vSwitch is a common software switch which supports OpenFlow. Open vSwitch is used in CloudMAC, another recent technology in wireless networking.

Testing of CloudMAC has shown that it has a low connection success rate. This thesis investigated the cause of the connection failures, and a solution using traffic control options in Open vSwitch was developed. Furthermore, additional traffic control options were added to Open vSwitch in order to evaluate the effectiveness of different traffic control algorithms.

While the connection success rate in a standard network without cross-traffic was 99%, with cross-traffic, the success rate dropped to 43%. With the introduction of traffic control, the success rate reached up to 99% success rate again.
Acknowledgments

I would like to express my sincere gratitude to my advisor Andreas Kassler for his extensive help throughout the whole duration of my thesis. His continuous support, enthusiasm, guidance, lightheartedness and humor has helped me immensely.
## Contents

1 Introduction .................................................. 1

2 Background & Related Work ................................. 3
   2.1 Introduction ............................................. 3
   2.2 Terminology ............................................ 3
   2.3 Quality of Service ...................................... 4
       2.3.1 Hierarchical Token Bucket (HTB) .............. 6
       2.3.2 Stochastic Fair Queueing (SFQ) .............. 6
       2.3.3 Random Early Detection (RED) ............... 7
       2.3.4 Controlled Delay (CoDel) and Fair Queueing CoDel (FQ_CoDel) .... 7
   2.4 Open vSwitch ........................................... 8
   2.5 OpenFlow ............................................... 9
   2.6 OpenWrt ................................................. 9
   2.7 CloudMAC ............................................... 9
   2.8 Related Work .......................................... 11
       2.8.1 QoSFlow ......................................... 11

3 Implementation .............................................. 12
   3.1 Overview ............................................... 12
### 3.2 Open vSwitch
- 3.2.1 The Open vSwitch Database ovSdb
- 3.2.2 The User-Space Tools
- 3.2.3 The User-Space Daemon vswitchd and Kernel Module

### 3.3 qdiscs
- 3.3.1 Hierarchical Token Bucket (HTB)
- 3.3.2 Stochastic Fair Queueing (SFQ)
- 3.3.3 Random Early Detection (RED)
- 3.3.4 Controlled Delay (CoDel) and Fair Queueing CoDel (FQ_CoDel)

### 3.4 Implementation of Datapath Changes
- 3.4.1 SFQ qdisc implementation

### 4 Evaluation & Results
- 4.1 Testbed
- 4.2 QoS Performance Evaluation
- 4.3 CPU Usage Evaluation
- 4.4 CloudMAC Delay & Performance Evaluation Setup
- 4.5 CloudMAC Connection Success Rate Results
- 4.6 CloudMAC Latency Test Results
- 4.7 CloudMAC Performance Test Results

### 5 Conclusion

### A Complete QoS implementations
- A.1 CoDel qdisc
- A.2 FQ_CoDel qdisc
B Test scripts

B.1 CloudMAC Connection Test Script .............................................. 61
B.2 QoS CPU Measurement Script .................................................. 62
List of Figures

2.1 pfifo_fast QoS algorithm ........................................... 5
2.2 SFQ QoS algorithm ................................................. 7
2.3 RED QoS algorithm ................................................. 7
2.4 The CloudMAC Architecture ........................................ 10

3.1 CloudMAC Network .................................................. 12
3.2 OpenFlow compared to Open vSwitch extension requirements ... 13
3.3 Implementation Overview .......................................... 14
3.4 Open vSwitch Database Schema .................................. 15

4.1 QoS series test example (2 in series with QoS HTB) .............. 29
4.2 QoS performance evaluation ....................................... 30
4.3 CPU usage test (4 separate queues at 50 Mbps) .................. 31
4.4 QoS performance evaluation ....................................... 31
4.5 Testbed setup ....................................................... 32
4.6 Cross-traffic graph .................................................. 34
4.7 CloudMAC connection success rate ................................ 36
4.8 CloudMAC latency test ............................................. 37
4.9 Latency without traffic control, without cross-traffic .......... 38
4.10 Latency without traffic control, with cross-traffic ............. 39
4.11 Latency with HTB traffic control .............................................. 39
4.12 Latency with FQ_CoDel traffic control ................................. 40
4.13 CloudMAC latency test ......................................................... 41
4.14 Bitrate without traffic control, without cross-traffic ............... 42
4.15 Bitrate without traffic control, with cross-traffic .................... 42
4.16 Bitrate with HTB traffic control ........................................... 43
4.17 Bitrate with FQ_CoDel traffic control ................................. 43
List of Tables

4.1 TP-Link TL-WR1043ND v1.x specifications .......................... 27
4.2 Cambria GW2358-4 specifications ................................. 28
4.3 VAP machine specifications[24] .................................... 28
4.4 Station machine specifications[25] ................................. 29
4.5 CloudMAC Tests List ................................................. 33
## List of Code Listings

<table>
<thead>
<tr>
<th>Code Listing</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>tc_ops struct</td>
<td>20</td>
</tr>
<tr>
<td>3.2</td>
<td>sfq struct</td>
<td>22</td>
</tr>
<tr>
<td>3.3</td>
<td>sfq_tc_install method</td>
<td>22</td>
</tr>
<tr>
<td>3.4</td>
<td>sfq_parse_qdisc_details__ method</td>
<td>22</td>
</tr>
<tr>
<td>3.5</td>
<td>sfq_setup_qdisc__ method</td>
<td>23</td>
</tr>
<tr>
<td>3.6</td>
<td>sfq_install__ method</td>
<td>24</td>
</tr>
<tr>
<td>3.7</td>
<td>sfq_tc_destroy method</td>
<td>24</td>
</tr>
<tr>
<td>3.8</td>
<td>sfq_qdisc_get method</td>
<td>25</td>
</tr>
<tr>
<td>3.9</td>
<td>sfq_qdisc_set method</td>
<td>25</td>
</tr>
<tr>
<td>3.10</td>
<td>sfq_class_get method</td>
<td>25</td>
</tr>
<tr>
<td>3.11</td>
<td>sfq_class_set method</td>
<td>25</td>
</tr>
<tr>
<td>3.12</td>
<td>sfq_class_delete method</td>
<td>26</td>
</tr>
<tr>
<td>3.13</td>
<td>sfq_tc struct</td>
<td>26</td>
</tr>
<tr>
<td>A.1</td>
<td>CoDel support implementation</td>
<td>48</td>
</tr>
<tr>
<td>A.2</td>
<td>FQ_CoDel support implementation</td>
<td>52</td>
</tr>
<tr>
<td>A.3</td>
<td>RED support implementation</td>
<td>56</td>
</tr>
<tr>
<td>B.1</td>
<td>Connection Test Script</td>
<td>61</td>
</tr>
<tr>
<td>B.2</td>
<td>CPU Measurement Script</td>
<td>62</td>
</tr>
</tbody>
</table>
Introduction

Software Defined Networking (SDN) have recently become very popular. A common way to perform SDN is through the use of OpenFlow. The latest version of OpenFlow does not support setting up traffic control. Traffic control is a requirement in many large networks used by business and organizations throughout the world. One attempt to add traffic control to OpenFlow switches was performed in the QoSFlow[1] paper.

CloudMAC is a recent development in wireless technologies. It allows the offloading of wireless mac layer processing from wireless access points to an external machine. This allows for heavier, centralized mac layer processing, seamless handovers between access points and a unified configuration interface.

One of the current issues with CloudMAC is that the connection success rate is relatively low. This is an issue because, when users connect to wireless networks they expect it to work without failure. As the intent of CloudMAC was to be, from the users standpoint, the same as any other wireless network, this is an important issue.

In order to solve the connection success rate issues with CloudMAC, this thesis investigates if the other (non-CloudMAC) traffic in the network could be the cause of the low connection success rate. Traffic control was thought out to be a suitable solution to the problem.

CloudMAC runs entirely in a OpenFlow enabled network, using Open vSwitch. This thesis will therefore attempt to add multiple traffic control algorithms to Open vSwitch (QoSFlow could not be used due to low performance), and make them available to controllers. As the OpenDaylight controller already has an interface to the Open vSwitch Database, we used that interface for configuration.

Various network configurations were set up to determine if the cause of the low connection success rate in CloudMAC was caused by cross-traffic. The results of the tests show that the cross-traffic was indeed related to the connection failure rate, and that with traffic control, the failure rate can be reduced from 57% to 1%.

The rest of the thesis is organized as follows: Chapter 2 contains the background of the project and related work. Chapter 3 is a description of the already existing implementation and the additions
that were performed. Chapter 4 is the evaluation setup and results. Chapter 5 presents the final conclusions.
Background & Related Work

This section will serve as an introduction to the various concepts, software and tools utilized in this project. First a list of terminology used in the rest of the thesis, then a list of various concepts, which the thesis will later refer to. There will also be a description of the software and tools used in the thesis.

2.1 Introduction

Wireless networking has very strict delay requirements during a wireless connection. Before this thesis, the fact that CloudMAC [2] had a low connection success rate was noticed. One possible cause to this low connection success rate could be the background traffic in networks.

This thesis seeks to investigate if the network load affects CloudMAC, and to what extent. Furthermore, if traffic control can improve the connection success rate.

2.2 Terminology

Here is a short description of the terminology and major utilities/systems used in this thesis. It can serve as a quick lookup while reading.

**QoS (Quality of Service)** Is the quality of a network service. Sometimes also synonymous with traffic control i.e. controlling traffic in order to provide quality of service.

**qdisc (Queueing Discipline)** A qdisc is attached to an Ethernet device and describes how that device should handle packets.

**class** A class (within traffic control) manages a certain type of data. Filters are used to determine which data ends up in which class. Classes allows for individual settings for different types of data/flows within a qdisc.
HTB (Hierarchical Token Bucket)  Classful algorithm which uses Token Buckets. Allows specifying priority, minimum and maximum rates for a certain class of data.

SFQ (Stochastic Fair Queueing) A classless algorithm that puts each flow in a separate FIFO queue, and dequeue the queues using a round robin scheduling algorithm.

RED (Random Early Detection)  Classless algorithm that drops packets based on a given probability. The probability is influenced by the average queue size.

CoDel (Controlled Delay)  Classless algorithm which is an attempt to deal with the bufferbloat problems in modern networking. Very dynamic and easy to configure.

FQ_CoDel (Fair Queue Controlled Delay)  Same as CoDel but also with fair queueing. Fair queueing, similarly to SFQ, divides flows into queues and then dequeues them using a round robin scheduling algorithm.

OpenFlow A protocol for configuring OpenFlow-enabled switches.

Open vSwitch A software switch with support for OpenFlow 1.3. Separates processing into a user space daemon and a kernel module for optimal performance.

CloudMAC System that offloads the wireless frame processing to a separate machine. Has very strict latency requirements.

FIFO (First In, First Out) A type of queue where the item first put into the list, is also the first to be processed and removed. Opposite of LIFO (Last In, First Out). Used in some QoS algorithms.

Round Robin A scheduling algorithm where time is divided into turns, and each process is allowed to perform processing for a certain time each turn. In queueing, each queue is allowed to dequeue a certain amount of bytes/packets every turn.

Flow A flow is a stream of data packets between a source and a destination. For example a TCP session or a UDP stream.

ToS Flags (Type of Service Flags) A part of the IPv4 header which designates the type of service which the packet originates from. Some QoS algorithms can use this field to determine how to prioritize the packet.

### 2.3 Quality of Service

Quality of Service (QoS) is defined as the quality of the network service. QoS paradigms and algorithms are used to help ensure a certain Quality of Service [3].

This background section will mainly focus on the Linux implementation of QoS. In Linux, QoS is performed by the use of *qdiscs*. A qdisc (Queueing Discipline) is attached to an Ethernet device
and describes how that device will handle packets (which QoS algorithm to use for that particular interface). In Linux, there are two types of qdiscs, ingress and egress qdiscs. Ingress qdiscs manage incoming packets, while egress qdiscs manage outgoing packets.

The default qdisc used in Linux is the egress qdisc pfifo_fast. It uses three 'bands' in which packets are enqueued. Each of these bands follow FIFO (first in first out) rules. All bands are the same length (txqueuelen) and the lower bands are dequeued first. pfifo_fast respects the ToS flags of a packet, putting higher priority traffic in band 0 (the band dequeued first). For more information, see `man tc-pfifo_fast(8)`.

![Figure 2.1: pfifo_fast QoS algorithm](image)

The pfifo_fast qdisc is a classless qdisc. Classless QoS algorithms can not configure individual subdivisions (classes) of traffic. The QoS algorithm might use internal 'queues' to schedule packets, but these are neither viewable nor configurable. For example pfifo_fast has internal bands, which it uses to subdivide traffic.

Classful qdiscs on the other hand, use configurable classes to subdivide traffic. When a packet is processed by a classful qdisc, it needs to be sent to one of its classes. This is called 'classifying' the packet. To decide which class a packet should be sent to, a list of filters are used. Filters will however not be a part of this project, as the OpenFlow enqueue action will be used instead.

There are several advantages in using traffic control in networks. It allows a more fair network allocation, allowing for greater control over how network resources are distributed. It can also make the network more reliable and robust. Malignant users can be dealt with and low-latency applications can get the low latency they require.

The disadvantages of using traffic control is that it adds additional complexity to the packet processing. This can lead into increased costs for the additional hardware that is required for enabling QoS. Another disadvantage is the number of algorithms that exists and their many options. This requires additional training for the network administrators. In addition, misconfigured traffic control can be difficult to detect. In many cases, it is cheaper to purchase more bandwidth than bother with traffic control at all. [4]

Here follows a description of the QoS algorithms that are already implemented or will be added to Open vSwitch.
2.3.1 Hierarchical Token Bucket (HTB)

HTB [man tc-htb(8)] is a classful traffic control algorithm, which allows for classifying different types of traffic, and give them different levels of throughput, latency and priority depending on this classification. HTB has three different types of classes: root, inner and leaf classes. HTB can be visualized as a tree, where the root is the top of the tree, the leaf classes are the end points and the inner classes are the rest. Traffic is classified into different classes using filters. The filters can filter on for example IP address or port number.

When the traffic has been classified it will be shaped according to the settings of the class it has been classified to. HTB uses tokens and buckets to allow for traffic shaping. Each class has an assured rate, ceil rate, actual rate, priority level and quantum. Excess bandwidth is shared with respect to the priority assigned to a class. This way, high priority classes get more bandwidth than low priority classes.

The advantage of HTB is that you can tailor the traffic control exactly as you like it, and HTB will adhere to those settings as much as possible. This allows for example to increase priority to certain applications, like web browsing or VoIP, while throttling others, like BitTorrent. [5]

2.3.2 Stochastic Fair Queueing (SFQ)

SFQ [man tc-sfq(8)] is a classless algorithm that provides very strong fairness between TCP transmissions. Each flow is given its own queue, to which packets from the flow are enqueued. SFQ then dequeues the first packet (FIFO) from each queue in a round robin fashion. The flows are divided by a hashing function.

SFQ allows you to change the perturbation period of the hashing function to provide more fairness, as the hashing function might group several distinct flows into the same queue. SFQ also provides the option of having a hard limit on the total queue size of all queues.

The advantage of SFQ is that it provides extremely fair queueing with a very simple algorithm, and very little configuration. But therein also lies its problem. It only works well if the users are well-behaving. Users that use up an abnormal amount of connections can still obtain an unfair amount of bandwidth.

Typically download managers, file sharing networks or BitTorrent clients, all very common applications of typical users, try to open as many flows as possible when downloading data. This will give them an unfair amount of bandwidth, as they obtain more queues in SFQ. This means the main usage is in a network where the users are well-behaving, and it can be quite efficient in such a network. [4]
2.3.3 Random Early Detection (RED)

RED [man tc-red(8)] is a classless algorithm which drops packets based on a given probability. It serves as an improvement over regular tail-drop (drop packets which overflow the queue at the tail). RED puts packets in a single queue, and drops packets according to a probability determined by the average queue size.

2.3.4 Controlled Delay (CoDel) and Fair Queueing CoDel (FQ_CoDel)

CoDel [man tc-codel(8)] is an algorithm to improve on the performance of RED. RED runs, due to the large number of parameters, a substantial risk of being configured inappropriately. Because of this CoDel was developed, to provide a QoS algorithm with fewer, and more easy to understand parameters. CoDel has also shown some improvements in performance over RED, due to additional improvements in the algorithm.
CoDel has been specifically developed to deal with the bufferbloat\(^1\) problems that is currently very prevalent in networks. Other features of CoDel is the separation of queues into "good" and "bad" queues, this allows CoDel to provide a low delay, while also dealing with bursts appropriately. It is also insensitive to round-trip delays, link rates and traffic load. Furthermore CoDel dynamically adapts to changes in rate and scales very well with the system it is used in. [6]

The advantage of CoDel is that it provides a variety of features, and scales very well with any type of networks it is used on. It also has practically zero configuration options, making configuring it wrong it a non-issue. However, it is a very new QoS algorithm, so its potential pitfalls are relatively unknown.

FQ_CoDel \([\text{man tc-fq_codel}(8)]\) is CoDel with the addition of fair queueing. Fair queueing works similarly to SFQ in that it divides traffic into flows, puts each flow in a separate queue and dequeues them according to a round robin scheduling algorithm.

### 2.4 Open vSwitch

Open vSwitch (OVS) is a software switch that has support for OpenFlow 1.3 (among other control protocols). It consists of a kernel module (openvswitch.ko) and a user space daemon (vswitchd). The user space daemon implements the switch, and uses the kernel module to perform some time-critical switch processing. [7]

The switch is configured through a simple database called ovsdb. There are a few tools provided for modifying the database: ovstool, ovsdb-tool and ovs-vsctl [7] [8]. The database uses the JSON protocol to communicate and store data [9]. ovsdb-client can be used to communicate directly with the database, sending JSON commands. Details on how these commands work are described in greater detail in [9].

ovs-vsctl provides a user-friendly interface to the database, and is also used to manage a running vswitchd instance. ovsdb-tool is used to manage ovsdb files, i.e. it can be used to perform transactions and queries, show logs etc. Open vSwitch also provides one more tool that will be used: ovs-ofctl which is a tool for sending OpenFlow commands to Open vSwitch. It is not exclusive to Open vSwitch, it can be used with any OpenFlow compatible switch. [8]

A software switch, such as Open vSwitch has many advantages. It provides the same functionality as a regular switch, but it is easily extensible and also highly configurable. In this thesis, it provides an excellent way to use OpenFlow without the need for expensive OpenFlow capable switches. Open vSwitch can also easily be extended with the additional functionality needed, such as the addition of quality of service options. The disadvantage of using a software switch is the amount of required processing power. On the other hand, a real switch can do the switch processing in

---

\(^{1}\)Excess packet queueing cause filled buffers, which leads to latencies. Particularly a problem with large buffers.
hardware, providing greater performance.

2.5 OpenFlow

OpenFlow was developed to be a system which would enable students at, for example a university to run experimental protocols at the university network. The idea is to open the typically closed technology behind a regular Ethernet switch, and allow users to modify the flow tables. It serves as an in-between protocol, a compromise between the user and the vendor. With OpenFlow, curious researcher can use the switches for running experiments, while the vendor does not have to expose the internal workings of the switch. [10]

A switch that uses the OpenFlow protocol, must implement at least a basic set of actions. OpenFlow is defined so that these core actions are always supported, but it is also extensible with additional features. OpenFlow can be divided into three components: The flow table, the secure channel and the OpenFlow protocol. [11]

2.6 OpenWrt

OpenWrt is a very small and extensible GNU/Linux distribution specifically made for embedded devices. OpenWrt is very dynamic because instead of providing a simple static firmware, it gives us a fully functional filesystem and a system for package management. In this project, OpenWrt will be used for the machines, as Open vSwitch can be run on it. [12]

OpenWrt uses BuildRoot [13] (with uclibc²), which is a way to easily build a complete Linux system for an external embedded system. The buildroot handles cross-compiling the toolchain, root filesystem generation, kernel compilation, bootloader compilation and image generation.

2.7 CloudMAC

CloudMAC is a system for offloading the processing of 802.11 WLAN MAC frames is moved from the Access Points to a Virtual Machine (VM), residing inside a data center. This is performed by forwarding all traffic received by the Access Point, through a tunnel, toward the Virtual Machine. Data is also forwarded in the other direction, from the Virtual Machine to the Access Point.

In CloudMAC, the name of the Access Point is WTP (Wireless Termination Point). The Virtual Machine is named the VAP (Virtual Access Point).

²C library optimized for embedded systems.
Separating the machines this way allows the VAP to run a Virtual WiFi Card, which is transparently connected to the Physical WiFi card in the WTP (both colored orange in Figure 2.4).

The Virtual WiFi Card is a modified version of the `mac802_hwsim` driver [14]. It has been modified to provide an interface, directly connected with a `gre tunnel` [15] tunnel. This tunnel will forward the information through the ethernet wire, to a WTP determined by OpenFlow rules (this tunnel is portrayed as the dashed arrow in Figure 2.4).

When the forwarded packet reaches the WTP, it will be sent onto the air, through a physical Wireless Network Card. This WiFi Card is set to monitor mode, using the `ath9k` driver. Any data received by this monitoring card will be forwarded back to the VAP (portrayed by the arrow in Figure 2.4).

This allows the VAP to be transparently connected to the Physical Wireless Card in the WTP. From the view of the VAP, the physical Wireless Card of the WTP is connected to the Virtual Machine. This allows for running Application Layer utilities, such as `hostapd` [16] on the VAP, instead of at the WTP. Because the VAPs are identical, regardless of the type of WTP, configuration becomes vendor independent.

The OpenFlow switch in-between the WTP and the VAP is running the software switch Open
vSwitch. A controller can be attached to this switch for performing various actions, such as seamless handovers, or changing VAP. In Figure 2.4, POX is used as controller, but any OpenFlow compatible controller can be used. [2] [17]

2.8 Related Work

In this section some related work to this thesis will be described, and how our approach is different from theirs.

2.8.1 QoSFlow

The QoSFlow [1] paper attempts to add QoS functionality to an OpenFlow switch. Their approach is to add the ability to alter QoS settings with the use of the OpenFlow protocol. The OpenFlow switch to which they implement these additions is the CPqD switch.

The CPqD switch performs the switching in user-mode. With CloudMAC, low latency is required, and using a switch that performs the processing in user-mode does not provide the low latency required. Instead a switch that performs the processing in kernel mode was chosen (Open vSwitch).

Another difference is that that the QoSFlow paper details changes to the OpenFlow protocol, while this thesis will on the other hand detail changes to the Open vSwitch database protocol.
Implementation

This section is a description of the changes that were made to Open vSwitch in order to enable the additional functionality we need, to provide low latency to the CloudMAC network. Background detail on the internals of Open vSwitch and Linux qdiscs. It will also present the changes made to improve CloudMAC latency.

3.1 Overview

This thesis aims to, through the use of traffic control, provide latency guarantees to a CloudMAC system, running alongside a standard OpenFlow network. An example of a CloudMAC network is shown in Figure 3.1.

Figure 3.1: CloudMAC Network
In Figure 3.1 the dashed lines represent standard network traffic flows, while the red bold lines represent CloudMAC traffic flows. Depending on the nature of the background traffic in the network, the CloudMAC traffic may experience transmission delays. As CloudMAC deals with very delay-sensitive packets (wireless association), immediate transmission is paramount. If a slight delay occurs, wireless association might fail.

As the entire network is using OpenFlow switches, and the controller is aware of the CloudMAC traffic flows, an extension to the OpenFlow protocol to enable traffic control options at the switches throughout the network would be beneficial. This would allow the OpenFlow Controller to set up the CloudMAC flows with higher priority, giving them better latency guarantees.

In this thesis, the extension will not be performed using OpenFlow. As the entire testbed is using Open vSwitch switches, it was instead decided to use the ovsdb JSON interface to configure the traffic control of the switches. This is due to that extending the OpenFlow protocol, would demand a substantially large workload, as compared to the ovsdb, which already has rudimentary support for traffic control (the HTB protocol).

Figure 3.2: OpenFlow compared to Open vSwitch extension requirements

The purpose of this thesis is to measure the viability of various traffic control schemes in conjunction with CloudMAC, and determine if any of them can improve the delay, especially in the situation of a very congested network. While extending OpenFlow to support traffic control, as in QoSFlow [1] is interesting, in this thesis, such an approach was not used. Instead, the implementation will not include an extension to the OpenFlow protocol, but rather an extension to the available traffic control protocols in Open vSwitch. The differences between these approaches are highlighted in Figure 3.2.

Currently Open vSwitch have one traffic control algorithm implemented (HTB). The implementation chapter will explain in detail the addition of four more algorithms (SFQ, RED, CoDel and FQ_CoDel). With these changes, the CloudMAC network should be able to prioritize traffic using five different traffic control algorithms. Figure 3.3 shows a summary of the changes made within this thesis.

13
The additional algorithms will later be used (along with the already implemented htb algorithm), in the evaluation chapter, to decide if these algorithms can make CloudMAC perform more reliably.

\subsection{Open vSwitch}

In order to fully understand the implementation, this section will provide a quick overview of Open vSwitch, and briefly explain the components which are used later. Open vSwitch is made up of many components, but the tools used in this thesis can be separated into four categories. There is the user-space daemon called \texttt{vswitchd}, the kernel module, the database \texttt{ovsdb} and finally the various user-space utilities used for configuring the switch.

\subsubsection{The Open vSwitch Database \texttt{ovsdb}}

The Open vSwitch database is where the configuration options for Open vSwitch is stored. The database is persistent, and is restored upon restart of the virtual switch. It is very similar to a normal relational database, see Figure 3.4 for a diagram of the Open vSwitch database schema. The tables marked red are the ones that are relevant for our changes.

The database supports most of the usual database operations, including:

- Listing databases and getting the schema for those databases
- Inserting new data (\texttt{insert})
- Updating data (\texttt{update \& mutate})
- Deleting data (\texttt{delete})
The Bridge table contains entries that describe the switches in the Open vSwitch instance. Each bridge contain a number of Ports. The Ports are represented in the Port table. They represent a virtual port on the virtual switch, which data can flow through. They are also used as ports in regards to OpenFlow rules. Each Port can be bound to an interface, which is an actual, physical network interface.

Furthermore, each port can have a single set of QoS options. The QoS options are, in case of Linux, represented as a qdisc. The changes described in this section mainly use this part of the database, thus it should be described in further detail. Each row in the QoS table corresponds to a single Port. Which means a Port cannot have multiple different QoS options. They can however have multiple queues.

Each QoS contain the following properties (only relevant properties are described):

- **_uuid** Unique UUID primary key.
- **other_config** A key-value pair list which stores the current settings of the qdisc.
- **queues** List of queues. In linux, queues are the same thing as classes.
**type**  The type of qdisc to be used (which QoS algorithm).

Here is an example of a simple HTB configuration (retrieved by `ovs-vsctl list qos`):

```
_uuid : 012a0e86-a9c6-421f-bc5d-24929ac116b3
external_ids : {}
other_config : {max-rate="1000000"}
queues : {0=237deba1-45b1-4bde-9dcb-652dbeb90944}
type : linux-htb
```

The Queue table contains entries for QoS queues. These queues act like classes for classful qdiscs. Open vSwitch currently only have classful qdiscs implemented. As classless qdiscs are going to be implemented, this have to be solved somehow. The solution chosen will be described later in the thesis.

Each Queue have the following properties (only relevant properties are described):

- **_uuid**  Unique UUID primary key.
- **other_config**  A key-value pair list which stores the current settings of the qdisc.
- **queues**  List of queues. In linux, queues are the same thing as classes.
- **type**  The type of qdisc to be used (which QoS algorithm).

Here is an example of the same simple HTB configuration (retrieved by `ovs-vsctl list queue`):

```
_uuid : 237deba1-45b1-4bde-9dcb-652dbeb90944
dscp : []
external_ids : {}
other_config : {max-rate="1000000", min-rate="1000000"}
```

### 3.2.2 The User-Space Tools

The user space tools constitute an important part of Open vSwitch. They provide an interface to the switch, allowing for configuration and monitoring. In this thesis, two different user-space utilities are used: `ovs-vsctl` and `ovs-ofctl`.

`ovs-vsctl` is a tool for controlling the virtual switch. It also provides a user-friendly interface to the database (as opposed to `ovsdb-client` which only allows you to send raw JSON requests). When the command is invoked, it will take any number of commands, which it will execute. If more than one command is specified, the commands must be separated with `--`. For example:
The command that will be used are:

**add-br** <name> Adds a bridge to the virtual switch. <name> specifies the desired name of the switch.

**del-br** <name> Deletes bridge with name <name>.

**add-port** <bridge> <interface> Adds a port to <bridge> connected to <interface>.

**set-controller** <bridge> <controller> Sets the controller of <bridge> to <controller>.

**set** <table> <record> <key=value>... Sets <key> to <value> at row where the record identifier is <record> (for example, port uses name as identifier) in <table>.

**create** <table> <key=value>... Inserts a row in <table> where the rows <key> is set to <value>.

**destroy** <table> <record> Deletes <record> from <table>.

**clear** <table> <record> <column> Sets <column> to an empty set or empty map in <record> in <table>.

Here is an example of configuring a HTB qdisc for interface eth0. The maximum rate is set to 1 Mbps (the --id=@uuid is used to store the created record's uuid in the variable @uuid, which can be referred to in subsequent commands):

```sh
ovs-vsctl add-br br0
ovs-vsctl add-port br0 eth0
ovs-vsctl -- set port eth0 qos=@newqos -- \
   --id=@newqos create qos type=linux-htb \
   other-config:max-rate=1000000 queues:0=@newqueue -- \
   --id=@newqueue create queue other-config:min-rate=1000000 \
   other-config:max-rate=1000000
```
3.2.3 The User-Space Daemon \texttt{vswitchd} and Kernel Module

As understanding of the user-space daemon and kernel module's structure is not required for this thesis, this section will be short. Basically, the user space daemon performs most of the not time sensitive tasks of the switch, such as QoS setup, while the kernel module performs all the time critical tasks, such as packet forwarding.

3.3 qdiscs

In addition to Open vSwitch, a basic understanding of the parameters of the Linux qdiscs are required. This section will be a description of the various qdiscs that are going to be added to Open vSwitch in order to perform the experiments in this thesis. For each qdisc, the parameters and how they are set up will be discussed, as those parameters will later be used in the Open vSwitch QoS implementation, through the use of tc and netlink.

3.3.1 Hierarchical Token Bucket (HTB)

HTB is a classful qdisc, and its parameters are configured for each class. The class parameters used in the thesis are \texttt{ceil} and \texttt{rate}:

\begin{itemize}
  \item \texttt{ceil} Maximum rate for a class, if its parents has bandwidth to spare.
  \item \texttt{rate} Maximum rate for this class and its children.
\end{itemize}

3.3.2 Stochastic Fair Queueing (SFQ)

SFQ is a classless qdisc with only a few parameters. It attempts to provide perfectly fair traffic control, but at the expensive of being easy to exploit. Here is a list of parameters that could be used to configure SFQ\footnote{There are actually more parameters, but we are only interested in the following}:

\begin{itemize}
  \item \texttt{perturb} The perturbation period for the queue algorithm i.e. the time between regeneration of the hash used to distinguish flows. This is used to ensure that a bad hash does not create unfair queueing.
  \item \texttt{quantum} The typical size of a packet.
\end{itemize}
3.3.3 Random Early Detection (RED)

RED is a classless qdisc with numerous parameters. As previously mentioned, one of the difficulties is the relative complexity of its parameters. The RED parameters used in the implementation are: min, max, limit, probability\(^2\), burst, avpkg and bandwidth. [18][19]

**min** When the average queue size is below this, no packet dropping will be performed.

**max** When the average queue size reaches max, the probability for dropping is probability (0.02).

**limit** Hard limit on the queue size in bytes. Packets that arrive when the actual queue size is above this are automatically dropped.

**burst** How large bursts should be allowed before dropping packets. The higher value, the slower the algorithm will react to changes in the flow.

**avpkg** Used along with the burst value to calculate average queue size.

**bandwidth** The bandwidth of the interface. Will be used to help calculate the average queue size when the queue has been idle for some time.

3.3.4 Controlled Delay (CoDel) and Fair Queueing CoDel (FQ_CoDel)

CoDel is a classless qdisc which attempts to alleviate bufferbloat problem. Compared to RED, it has fewer and more easily explained parameters: limit, target and interval. [20]

**limit** The hard limit on the queue i.e. maximum queue size. Specified in number of packets.

**target** The minimum acceptable delay.

**interval** Used to guarantee that the minimum delay does not stagnate. The minimum delay must have been experienced once within every interval.

FQ_CoDel is CoDel with the addition of fair queueing (very similar to SFQ). This addition comes with two additional parameters: flows and quantum. [21]

**flows** The number of flow slots into which incoming packets will be assigned.

**quantum** The typical size of a packet.

\(^2\)Not actually used in implementation, but included for clarity.
3.4 Implementation of Datapath Changes

The changes that were performed to Open vSwitch involve adding support for configuring additional quality of service options. Open vSwitch performs the configuration of QoS in the file `netdev-linux.c`. Each qdisc have a certain set of configuration options that need to be filled. These configuration options are stored in the `tc_ops` struct.

When QoS is configured, a search of each `tc_ops` struct in `tcs` is performed, where it looks if the field `ovs_name` matches the type configured in the QoS table in ovsdb. The struct has, beyond the `ovs_name` field various function pointers that point to functions for setting up that particular qdisc.

```c
static const struct tc_ops tc_ops_htb = {
    "htb", /* linux_name */
    "linux-htb", /* ovs_name */
    HTB_N_QUEUES, /* n_queues */
    htb_tc_install, /* tc_install */
    htb_tc_load, /* tc_load */
    htb_tc_destroy, /* tc_destroy */
    htb_qdisc_get, /* qdisc_get */
    htb_qdisc_set, /* qdisc_set */
    htb_class_get, /* class_get */
    htb_class_set, /* class_set */
    htb_class_delete, /* class_delete */
    htb_class_get_stats, /* class_get_stats */
    htb_class_dump_stats /* class_dump_stats */
};
```

Listing 3.1: `tc_ops` struct

Listing 3.1 contains an example of how the HTB qdisc was set up within Open vSwitch. The first parameter `linux_name` is the name of the qdisc in Linux. The `ovs_name` is, as previously stated simply the type in the ovsdb database. `n_queues` is the maximum number of supported OpenFlow queues. The remaining fields are function pointers to various qdisc specific functions:

- **tc_install** Takes a net device (`netdev`) and a key-value map (`smap`) as parameters. The key-value map contains the information about the QoS from the ovsdb database. This function should set up the net device with the parameters specified in the database (provided through `smap`). There is no need to delete the previous qdisc, as this is managed by Open vSwitch. If this option is null, the qdisc cannot be installed (such is the case for the default qdisc `tc_ops_other`).

- **tc_load** When Open vSwitch determines that there is a qdiscs installed on a managed net device, this method is called. Provides a net device and netlink (`ofpbuf`) message as parameters. This function should parse the netlink message and determine the parameters of the installed qdiscs.

---

3 Linux net device configuration. The openwrt devices we use will be using Linux.
4 This qdisc was already implemented, along with HFSC in Open vSwitch.
qdisc. It should also query for the queue configuration of the qdisc, if there is one. This method is not needed for the experiments in this thesis, and is therefore not implemented.

**tc_destroy** Destroy the data structures which is part of the tc struct.

**qdisc_get** Retrieve the configuration of the qdisc and return it to the caller. The provided parameters are the net device, and a key-value map which the function should fill with appropriate key-value pairs.

**qdisc_set** Reconfigure the qdisc bound to the provided net device parameter. The net device should be configured according to the key-value map provided as a parameter.

**class_get** Retrieve the configuration of the queues and return it to the called. The provided parameters are the net device, and a key-value map which the function should fill with appropriate key-value pairs.

**class_set** Reconfigure the queue which is provided as a parameters. Should be reconfigured according to the key-value map also provided as a parameter.

**class_delete** Delete the queue from the tc struct.

All these functions need to be implemented for each previously specified qdisc (SFQ, RED, CODEL, FQ_CODEL). This will allow for configuring these new qdiscs using ovsdb, or any of the user-space tools provided. If the qdisc is classful, this will also allow packets to be enqueued in the configured queues (classes) using OpenFlow flow tables.

Furthermore, as all the already implemented qdiscs are classful, a method to implement classless qdiscs must be devised. The ovsdb QoS tables requires at least one queue to be configured. The chosen solution involves simply configuring a imaginary queue, with the same parameters as the qdisc (ovs-vsctl doesn't allow queues or qdiscs to be parameterless). This would be the simplest solution, as it does not require changing the Open vSwitch internals. The solution is also good enough for the purposes of this thesis.

If these changes should actually be implemented into the mainline Open vSwitch, perhaps a different, more clean approach would be better, such as perhaps separating qdiscs in classless and classful. Or simply allowing qdiscs to be queueless. To help understand how the additional qdiscs were implemented, the implementation of one of them will be detailed in this report. The described qdisc will be SFQ, as it has the fewest parameters and has the simplest algorithm.

### 3.4.1 SFQ qdisc implementation

SFQ has, as previously stated, only two parameters that are going to be used in this thesis. These parameters are quantum and perturbation. quantum is the size of a packet, and perturbation
is the time between regeneration of the flow-separation hashes. The first step was to add a struct containing these values which can be seen in listing 3.2.

```c
struct sfq {
    struct tc tc;
    uint32_t quantum;
    uint32_t perturb;
};
```

Code Listing 3.2: sfq struct

The struct tc is a struct used by tc. It will be initialized in sfq_tc_install. The first method is the tc_install method. Listing 3.3 details the implementation of this method.

```c
static int sfq_tc_install(struct netdev *netdev, const struct smap *details) {
    int error;
    struct sfq sfq;
    sfq_parse_qdisc_details__(netdev, details, &sfq);
    error = sfq_setup_qdisc__(netdev, sfq.quantum, sfq.perturb);
    if (!error) {
        sfq_install__(netdev, sfq.quantum, sfq.perturb);
    }
    return error;
}
```

Code Listing 3.3: sfq_tc_install method

First a new sfq struct is defined, and later filled with data from the sfq_parse_qdisc_details__ function call (code listing 3.3 at line 7). This fills the sfq struct with data from the details parameter. Thereafter, an attempt to set up the qdisc is performed, and if there are no errors, the qdisc is installed.

The sfq_parse_qdisc_details__ (code listing 3.4) function parses the qdisc configuration provided in the details parameter. The parsed configuration is stored in the sfq parameter. If the details parameters contains an incomplete configuration, good default values\(^5\) are picked.

```c
static void sfq_parse_qdisc_details__(struct netdev *netdev,
    const struct smap *details, struct sfq *sfq)
{
    const char *perturb_s;
    const char *quantum_s;
    int mtu;
    int mtu_error;
    perturb_s = smap_get(details, "perturb");
    quantum_s = smap_get(details, "quantum");
    sfq->perturb = perturb_s ? strtoull(perturb_s, NULL, 10) : 0;
    sfq->quantum = quantum_s ? strtoull(quantum_s, NULL, 10) : 0;
```

\(^5\)Most of these values are from man files, or qdisc documentation
if (!sfq->perturb) sfq->perturb = 10;
if (!sfq->quantum) {
    mtu_error = netdev_linux_get_mtu__(netdev_linux_cast(netdev), &mtu);
    if (!mtu_error) {
        sfq->quantum = mtu;
    } else {
        VLOG_WARN_RL(&rl, "when using SFQ, you must specify quantum on a device without mtu");
        return;
    }
}

Code Listing 3.4: sfq_parse_qdisc_details__ method

Both parameters (perturbation and quantum) are retrieved from details and converted to integers. If the value cannot be converted, the default 0 is used (code listing 3.4 at line 10-13). Then, if the configuration cannot be retrieved, good defaults are used. The default for quantum is the mtu of the network interface. The default for perturbation is 10 seconds.

The next method to discuss is the sfq_setup_qdisc__ method (code listing 3.5). This method sets up the actual qdisc, using netlink. It is set up according to the options provided in the parameters (quantum and perturb).
First, the mtu of the interface is retrieved and the old qdisc is deleted (code listing 3.5 at line 9 and 11). Then a new tc message (tcmsg) is created and is set to the root qdisc (code listing 3.5 at line 15-19). Thereafter we use a tc_sfq_qopt struct to specify the options of the qdisc. This struct is defined in pkt_sched.h in the linux source. If there are no options specified, default value are picked similarly to in sfq_parse_qdisc_details__ (code listing 3.4 at line 14-23). Thereafter the message is sent to netlink (code listing 3.5 at line 34-37).

The last function call in sfq_tc_install was sfq_install__ (code listing 3.6). This method sets up a sfq struct and sets the netdev tc struct to the sfq struct's tc struct, saving the configuration.

The next required method that was implemented is tc_destroy (code listing 3.7). This method destroys the tc reference and frees the data used by the sfq struct.
Then the methods for gettings and settings qdiscs were implemented. These are presented in code listing 3.8, code listing 3.9, code listing 3.10 and code listing 3.11 respectively.

```c
static int
sfq_qdisc_get(const struct netdev *netdev, struct smap *details)
{
    const struct sfq *sfq = sfq_get__(netdev);
    smap_add_format(details, "quantum", "%u", sfq->quantum);
    smap_add_format(details, "perturb", "%u", sfq->perturb);
    return 0;
}
```

**Code Listing 3.8: sfq_qdisc_get method**

```c
static int
sfq_qdisc_set(struct netdev *netdev, const struct smap *details)
{
    struct sfq sfq;
    int error;
    sfq_parse_qdisc_details__(netdev, details, &sfq);
    sfq_install__(netdev, sfq.quantum, sfq.perturb);
    sfq_get__(netdev)->quantum = sfq.quantum;
    sfq_get__(netdev)->perturb = sfq.perturb;
    return error;
}
```

**Code Listing 3.9: sfq_qdisc_set method**

```c
static int
sfq_class_get(const struct netdev *netdev, const struct tc_queue *queue OVS UNUSED, struct smap *details)
{
    const struct sfq *sfq = sfq_get__(netdev);
    smap_add_format(details, "quantum", "%u", sfq->quantum);
    smap_add_format(details, "perturb", "%u", sfq->perturb);
    return 0;
}
```

**Code Listing 3.10: sfq_class_get method**

The `sfq_class_get` method returns the configuration for the qdisc, as every QoS needs a queue in the ovsdb schema, the `sfq_class_get` returns a "fake" queue to fulfill the schema requirements.

```c
static int
sfq_class_set(struct netdev *netdev OVS UNUSED, unsigned int queue_id OVS UNUSED, const struct smap *details OVS UNUSED)
{
    return 0;
}
```

**Code Listing 3.11: sfq_class_set method**
The `sfq_class_get` method only returns 0. This is because SFQ is classless and thus settings the options for a queue does nothing.

```c
1 static int
2 sfq_class_delete(struct netdev *netdev OVS_UNUSED, struct tc_queue *queue OVS_UNUSED)
3 {
4     return 0;
5 }
```

Code Listing 3.12: `sfq_class_delete` method

The `sfq_class_delete` method also only returns 0, as a queue is never created, there is no need to destroy it.

Finally, all these methods are set up into a struct (code listing 3.13):

```c
1 static const struct tc_ops tc_ops_sfq = {
2     "sfq", /* linux_name */
3     "linux-sfq", /* ovs_name */
4     SFQ_N_QUEUES, /* n_queues */
5     sfq_tc_install,
6     sfq_tc_load,
7     sfq_tc_destroy,
8     sfq_qdisc_get,
9     sfq_qdisc_set,
10    sfq_class_get,
11    sfq_class_set,
12    sfq_class_delete,
13    NULL,
14    NULL
15};
```

Code Listing 3.13: `sfq_tc` struct

The two last NULL pointers are the two previously unmentioned methods `class_get_stats` and `class_dump_stats`. These functions deal only with queues and are not needed within the scope of this thesis. This concludes the complete implementation of a single qdisc in Open vSwitch. The remaining three qdiscs are implemented in a similar manner, and the implementation details can be studied further in the appendix.
Evaluation & Results

This chapter will describe the system used for performing measurements of the additional features introduced into CloudMAC, how these measurements were set up and what devices were used so that the experiments may be replicated with ease.

4.1 Testbed

This chapter will detail the various machines and setups used to perform the evaluation. The testbed built for this thesis contains two different types of machines. The first type of machine is the TP-Link TL-WR1043ND. These are standard consumer routers, which are cheap and easily available. One requirement for the evaluation to work is that the device supports OpenWrt, which the TL-WR1043ND does. The devices used are revision v1.x. Detailed specifications are available in Table 4.1.

<table>
<thead>
<tr>
<th>Type</th>
<th>TL-WR1043ND v1.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Set</td>
<td>MIPS</td>
</tr>
<tr>
<td>Chip Vendor</td>
<td>Qualcomm Atheros</td>
</tr>
<tr>
<td>Boot Loader</td>
<td>U-Boot 1.1.4</td>
</tr>
<tr>
<td>SOC¹</td>
<td>AR9132 rev 2 (MIPS 24Kc V7.4)</td>
</tr>
<tr>
<td>Processor</td>
<td>24Kc V7.4 400 MHz</td>
</tr>
<tr>
<td>Flash Chip</td>
<td>ST 25P64V6P</td>
</tr>
<tr>
<td>Flash Size</td>
<td>8192 KiB</td>
</tr>
<tr>
<td>Memory</td>
<td>32MiB</td>
</tr>
<tr>
<td>Ethernet Speed</td>
<td>1000 Mbps (Tested: 800 Mbps)</td>
</tr>
<tr>
<td>WNIC</td>
<td>Atheros AR9103 2.4 GHz 802.11bgn (ath9k)</td>
</tr>
</tbody>
</table>

Table 4.1: TP-Link TL-WR1043ND v1.x specifications [22]

The TP-Link devices are going to be used as OpenFlow switches, by installing Open vSwitch on them. The other set of devices used are Cambria GW2358-4 machines, with four Wireless Network

¹System On a Chip
Interface Cards (WNICs) installed. These devices are going to be used as WTPs in the CloudMAC setup. Table 4.2 details the specifications of these machines.

<table>
<thead>
<tr>
<th>Type</th>
<th>Cambria GW2358-4</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Instruction Set</strong></td>
<td>ARM</td>
</tr>
<tr>
<td><strong>Boot Loader</strong></td>
<td>RedBoot</td>
</tr>
<tr>
<td><strong>Processor</strong></td>
<td>Intel® XScale® IPX435 667MHz</td>
</tr>
<tr>
<td><strong>Memory</strong></td>
<td>128 MiB DDRII-400 SDRAM</td>
</tr>
<tr>
<td><strong>Flash Size</strong></td>
<td>32 MiB</td>
</tr>
<tr>
<td><strong>Ethernet Speed</strong></td>
<td>100 Mbps</td>
</tr>
<tr>
<td><strong>WNIC</strong></td>
<td>3x Compex WLM54AG 2.4 GHz 802.11abg (ath5k)</td>
</tr>
<tr>
<td></td>
<td>1x Compex WLM200NX 2.4/5.0 GHz 802.11abgn (ath9k)</td>
</tr>
</tbody>
</table>

Table 4.2: Cambria GW2358-4 specifications [23]

For these two machines, two separate OpenWrt BuildRoots were prepared. In both buildroot, various diagnostic packages were installed: tc, tcpdump, wireless_tools and net-tools. In addition, Open vSwitch 2.0 was downloaded and compiled for both BuildRoots. Both BuildRoots were using the Attitude Adjustment 12.09 OpenWrt image, with the Linux 3.3.8 kernel.

In order to deploy the images, the TP-Link devices were flashed in recovery mode using the mtd tool provided by OpenWrt. The IPX435 devices on the other hand, used netboot and tftp to get its image from the development machine.

For the VAP, a separate desktop machines was used. Its specifications can be seen in Table 4.3. A separate, identical machine was used for cross-traffic generation in the CloudMAC delay evaluation test.

<table>
<thead>
<tr>
<th>Type</th>
<th>VAP / Dev Machine</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Instruction Set</strong></td>
<td>x86 (i686)</td>
</tr>
<tr>
<td><strong>Processor</strong></td>
<td>Intel® Core Duo 2.0Ghz</td>
</tr>
<tr>
<td><strong>Memory</strong></td>
<td>2 GiB DDRII SDRAM</td>
</tr>
<tr>
<td><strong>Ethernet Speed</strong></td>
<td>1000 Mbps (Tested: 1000 Mbps)</td>
</tr>
</tbody>
</table>

Table 4.3: VAP machine specifications[24]

Ubuntu 12.04 LTS was installed and used on the VAP machine, and the kernel was reverted to 2.6.32-22-generic. This because the modified wireless stack that CloudMAC uses requires kernel version 2.6, and 2.6.32.22-generic was recommended in the CloudMAC documentation. Furthermore Open vSwitch 1.4 was compiled and installed on the VAP. The same machine was also used for the two developer machines (later used for crosstraffic generation and reception).

Furthermore a final machines was used as a wireless station. It seemed fit to use a standard cheap consumer-level laptop for the actual performance measurements. The laptop used was a ASU Eee
PC 1215B, with an additional TP-Link TL-WN722N. This additional wireless adapter was used for its external antenna and better performance, while still being cheaply available. Both the internal WNIC and the TL-WN722N was used during CloudMAC testing.

<table>
<thead>
<tr>
<th>Type</th>
<th>Station Machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Set</td>
<td>x64</td>
</tr>
<tr>
<td>Processor</td>
<td>AMD® C60 1.0GHz (dual core) Processor</td>
</tr>
<tr>
<td>Memory</td>
<td>DDR3, 2 x SO-DIMM, 2GB</td>
</tr>
<tr>
<td>Ethernet Speed</td>
<td>100 Mbps</td>
</tr>
<tr>
<td>WNIC</td>
<td>Broadcom WLAN 2.4 GHz 802.11bgn (b43)</td>
</tr>
<tr>
<td></td>
<td>TP-Link TL-WN722N (ath9k_htc)</td>
</tr>
</tbody>
</table>

Table 4.4: Station machine specifications[25]

The later sections will explain how these devices are connected, and how the testing was performed.

4.2 QoS Performance Evaluation

The QoS performance evaluation is a test for measuring the performance of the QoS algorithms with different number of switches linked in series. The test was performed by linking several switches in series and testing the performance using iperf with both QoS turned off and with various QoS algorithms. See Figure 4.1 for a diagram depicting the setup.

![Diagram of QoS series test example (2 in series with QoS HTB)](image)

Figure 4.1: QoS series test example (2 in series with QoS HTB)

Traffic was generated and received by the two Dev Machines. The inspiration for this test came from the QoSFlow [1] paper. The results of this test can be seen in Figure 4.2.
In Figure 4.2, the y-axis represents the average bitrate in Mbps. The x-axis represents the number of switches in the network (see Figure 4.1 for details on the setup). The gray bars represent the network without any traffic control. The blue bars represent the network with HTB traffic control enabled, red with SFQ, green with CoDel and purple with FQ_CoDel. For HTB, the qdisc is set up so that all traffic is going through a single queue using HTB with a maximum rate of 1 Gbps. The other traffic control algorithms use the default settings.

This graph shows, in addition, that there is only a slight performance drop with SFQ, CoDel and FQ_CoDel. The performance drop is larger when using HTB. This could be due to the fact that HTB uses more resources than the other, classless qdiscs. We can also see that the number of switches in line does not severely reduce the performance. This is consistent to the findings in the QoSFlow[1] paper, however these tests show a 10 times increase in performance over the QoSFlow approach. The reason for this being that we use a kernel modules for the switching, as opposed to a user-space daemon.

### 4.3 CPU Usage Evaluation

In CPU Usage evaluation the cpu usage of one TP-Link switch was measured with different number of queues and flows. As the only qdisc we implemented that supports queues is htb (because it is classful). A TP-Link switch was set up with 1, 2, 3, 4, 5, 6 and 7 queues, while each queue had one
iperf generated flow going through it. Figure 4.3 shows a diagram of using 4 queues. Appendix B.2 contains the script used for CPU measurement.

![Diagram of using 4 queues](image)

**Figure 4.3:** CPU usage test (4 separate queues at 50 Mbps)

Each queue was set to with a max rate of 50 Mbps. Traffic was generated and received by the two Dev Machines. The inspiration for this test came from the QoSFlow [1] paper. The results of this test can be seen in Figure 4.4.

![CPU Usage Chart](image)

**Figure 4.4:** QoS performance evaluation

In Figure 4.4, the y-axis represents the CPU usage of the openflow switch in the experiment. The x-axis represents the number of queues with a max-rate of 50 Mbps. The darker the bars, the more queues are used. Each of these queues had an iperf session running through it. The measured

31
throughput at the other end was not exactly 50 Mbps, but rather, the rate sometimes ended up higher, up to 80 Mbps, which could be the reason for the high CPU usage.

The graph shows a linear increase in CPU usage, which means that the conclusion reached in the QoSFlow [1] paper (CPU usage is independent of queue count, but rather dependant on total throughput) can be confirmed.

4.4 CloudMAC Delay & Performance Evaluation Setup

The final and most important test was testing the change in performance when adding traffic control to a CloudMAC network. Specifically when dealing with cross-traffic. This because CloudMAC is very sensitive to delays, so that even without cross-traffic there can be connection failures. What this thesis is attempting to solve is that when previously the CloudMAC network was evaluated, a high amount of wireless connection failures were experienced.
It was determined that one of the factors to these connection failures were the additional time it takes to pass a single packet between the WTP and VAP, through a network of OpenFlow switches. It was theorized that by utilizing traffic control, these problems could be eliminated. This test is an attempt to evaluate various traffic control algorithms, and how they affect CloudMAC performance.

The test was built as a standard CloudMAC network, with a VAP running at a powerful machine (Dell Optiplex 755), and a single WTP running on a much weaker machine, the Cambria GW23584. Furthermore, two OpenFlow switches were added into the network. These switches manage both CloudMAC traffic between the VAP and WTP, and also traffic between a traffic generator/recipient pair (Dev Machines). Figure 4.3 is a diagram of how the testbed setup looks like.

The tests performed were delay measurements using ping, udp and tcp data rate using netperf and connection failure rate. These tests were performed without crosstraffic, with crosstraffic and with both traffic control and crosstraffic. It was deemed useless to perform these tests with traffic control, but without cross traffic, as there is only one flow. The traffic control algorithms tested were: HTB, SFQ, FQ_CoDel and CoDel. The implemented algorithm RED did not make it to the final tests due to both timing constraints, and problems getting the kernel module to work with the TP-Link devices.

Furthermore, the tests were performed with both 1000 Mbps connection and with 100 Mbps connections. The 100 Mbps connections was emulated using software rate limiting (HTB). A better alternative would have been to set the media independent interface status using mii-tool or ethtool, however the TP-Link devices did not support this, so HTB was used instead. Table 4.5 shows a summary of the tests performed.

<table>
<thead>
<tr>
<th>Rate</th>
<th>qdisc</th>
<th>Cross Traffic?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000 Mbps</td>
<td>None</td>
<td>No</td>
</tr>
<tr>
<td>1000 Mbps</td>
<td>None</td>
<td>Yes</td>
</tr>
<tr>
<td>1000 Mbps</td>
<td>SFQ</td>
<td>Yes</td>
</tr>
<tr>
<td>1000 Mbps</td>
<td>HTB</td>
<td>Yes</td>
</tr>
<tr>
<td>1000 Mbps</td>
<td>CoDel</td>
<td>Yes</td>
</tr>
<tr>
<td>1000 Mbps</td>
<td>FQ_CoDel</td>
<td>Yes</td>
</tr>
<tr>
<td>100 Mbps</td>
<td>None</td>
<td>No</td>
</tr>
<tr>
<td>100 Mbps</td>
<td>None</td>
<td>Yes</td>
</tr>
<tr>
<td>100 Mbps</td>
<td>SFQ</td>
<td>Yes</td>
</tr>
<tr>
<td>100 Mbps</td>
<td>HTB</td>
<td>Yes</td>
</tr>
<tr>
<td>100 Mbps</td>
<td>CoDel</td>
<td>Yes</td>
</tr>
<tr>
<td>100 Mbps</td>
<td>FQ_CoDel</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 4.5: CloudMAC Tests List

The cross-traffic was generated using netperf, with the help of the netperf-wrapper [26] utility.

---

2This test failed presumably due to the intense load, which the TP-Link device could not handle, and thus crashed (reboot, not kernel panic).
This tool was also used to measure the TCP, UDP and latency of the network, and for the creation of statistics and graphs. Figure 4.6 shows the cross-traffic throughput in a network without any traffic control.

Figure 4.6: Cross-traffic graph

In Figure 4.6, the BE, BK, CS5 and EF represent four different TCP flows. As can be seen in the figure, there are both TCP upload and TCP download traffic. These two traffic types are generated with the TPC_STREAM and TCP_MAERTS netperf tests respectively. Furthermore, the figure shows the latency calculated using different methods. UDP EF, UDP BK, and UDP BE are all UDP ping tests using the UDP_RR netperf test. ICMP is ICMP ping generated using the ping utility.

The QoS settings for the tests are as follows:

**HTB Root**

max-rate: 450 Mbps

**HTB CloudMAC Queue**

priority: 1
HTB Cross-Traffic Queue

priority: 99

SFQ

limit: 127p
quantum: 1500p
depth: 127
divisor: 1024
perturb: 10sec

CoDel

limit: 1000p
target: 4us
interval: 99us

FQ_CoDel

limit: 1000p
flows: 1024
quantum: 1514
target: 4us
interval: 99us

4.5 CloudMAC Connection Success Rate Results

The CloudMAC connection success rates test involves attempting to connect to the CloudMAC access many times and try to determine the success rate for connecting. Tests with multiple different configurations were performed to determine if traffic control increase the success rate of CloudMAC connections. The results are available in Figure 4.7.

Figure 4.7 contains two subfigures. One is using a 100 Mbps (rate limited) connection and the other is the same test using a 1000 Mbps connection. The y-axis represents the connection success rate in percent. The x-axis represent the setup used (notc = no traffic control, nc = no cross-traffic). Each figure has five different bars. The gray bar (notc-nc) is without traffic control or cross-traffic. The other gray bar is without traffic control, but with cross-traffic. The blue ones are using HTB (with cross-traffic), red SFQ (with cross-traffic), green CoDel (with cross-traffic), purple FQ_CoDel (with cross-traffic).

3From here on, 100 Mbps graphs will be on the left, while 1000 Mbps graphs on the right.
From these measurements various observations can be made. It should be noted that the connection rate of CloudMAC without any crosstraffic is 99%. This means that the best possible connection rate, even with traffic control enabled is 99%. However, when cross-traffic was added to the network, the connection rate dropped to 43% in the 100 Mbps network and 83% in the 1000 Mbps network. The loss in success rate is greater in the 100 Mbps network due to higher latency caused by bufferbloat.

HTB seems to perform well in both the 100 Mbps network and the 1000 Mbps network. In both cases, a 99% connection success rate can be seen. This is the same as the connection rate without cross-traffic, and thus HTB solves the latency problem perfectly. In 100 Mbps we also see a large improvement in the connection success rate using the other three algorithms, SFQ and FQ_CoDel showing the best performance (CloudMAC flow is isolated from the other flows). The CoDel improvement in 100 Mbps is not as good as the other two algorithms, but some improvement can be seen. This is probably due to that CoDel does not perform fair queueing.

In the 1000 Mbps network, the improvements that SFQ, CoDel and FQ_CoDel provide are not at all as impressive. It can be theorized that this is due to CPU usage. The TP-Link switches CPU is completely used up during these tests and not even the control channel (port 4 in Figure 4.5) is responsive at all. It is possible that the packets are dropped due to this.

Figure 4.7: CloudMAC connection success rate
4.6 CloudMAC Latency Test Results

As latency is important in CloudMAC, additional latency tests were performed using the ping utility. Latency is measured between the Station connected to the CloudMAC network and the VAP.

![CloudMAC Latency Test (100Mbps)](image1)

![CloudMAC Latency Test (1000Mbps)](image2)

Figure 4.8: CloudMAC latency test

Figure 4.8 shows a summary of all the ping tests performed. The y-axis represent the round-trip time between the VAP and the Station. The x-axis represent the different network configurations. Like in the connection test, notcnc (Gray) means no traffic control and no cross-traffic. notc (Gray) means no traffic control, but with cross-traffic. The others (htb, sfq, codel, fq_codel) just bear the names of the traffic control algorithm used. Those four tests all have cross-traffic going through the network.

From the leftmost graph, the delay in the 100 Mbps network without any cross-traffic can be seen, the delay without any cross-traffic is 1.858 ms. When cross-traffic is added to the network, the increases to 166.32 ms. The increased delay, is just like in the connection rate test, caused by bufferbloat. At this delay, the connection rate was, as seen in Figure 4.7 as low as 43%. When traffic control was introduced into the network, the latency lowered significantly. The results are overall consistent with the connection success rate test. HTB is at the top, with about the same latency as a network without cross-traffic (1.88 ms). CoDel gives some improvement at 28.659 ms, while SFQ (5.63 ms) and FQ_CoDel (4.317 ms) are in-between.

Looking at the rightmost graph, the delay in the 1000 Mbps network, the latency is consistent
with the connection success rate. It can be concluded that network latency is at least part in what cause the high connection failure rate. In the next section, the results for the throughput will be presented, and with that it can be determined if both latency and throughput affect connection success probability, or if only latency does.

The 1000 Mbps graph show that the delay without any cross-traffic is similar to the 100 Mbps network (1.726 ms). With cross-traffic the delay is slightly increased (4.181 ms), but not at all at the same extent as in the 100 Mbps test. The delay is not lowered significantly by the QoS algorithm, however this is probably due to the Settings (FQ_CoDel and CoDel have target delay set to 5 ms).

What is interesting is that in the 100 Mbps network, a 5 ms delay gives a connection success rate of 98%, while in the 1000 Mbps network a 5 ms delay gives a connection success rate of only 83%. In other words, when the network speed changes to from 1000 Mbps to 100 Mbps, the connection success rate is increased. One cause could be that the HTB qdisc used for rate limiting shapes the flows in such a way that the connection success rate is increased. Redoing this test with a Ethernet card that supports mii-tool or ethtool could provide more accurate results.

To give a better insight into the variance of the latency, additional figures will be presented below which show a more detailed analysis of the latency.

![Latency without traffic control, without cross-traffic](image)

Figure 4.9: Latency without traffic control, without cross-traffic

Figure 4.9 shows the latency of the network, without any cross-traffic. The y-axis represents the round trip time for a single ICMP packet generated by the ping utility. The x-axis show the time passed in seconds. Pings are sent one per second. The 100 Mbps network show a mean latency of 1.866 ms ($\sigma$ 0.13), while the 1000 Mbps network show a mean latency of 1.7 ms ($\sigma$ 0.11).
Figure 4.10: Latency without traffic control, with cross-traffic

Figure 4.10 shows the same ping test, but with cross-traffic generated between the dev machines. The cross-traffic has in both graphs increased the latency between the Station and the VAP. The mean latencies recorded were 207.012 ms (σ 28.956) for the 100Mbps network and 4.201203 ms (σ 0.333) for the 1000Mbps network. The reason for the difference could be explained by that the 100Mbps network experiences bufferbloat, while the 1000Mbps one does not. This because there is little queue buildup at the switch in the 1000Mbps network.

Figure 4.11: Latency with HTB traffic control
Figures 4.11 and 4.12 show the latency when using HTB and FQ_CoDel respectively for traffic control.
4.7 CloudMAC Performance Test Results

Here are the results for the TCP and UDP tests performed on the CloudMAC network. Data was sent using netperf\text{-}wrapper between the traffic generator and traffic receiver.

![CloudMAC Latency Test (100Mbps)](image)

(a) 100Mbps

![CloudMAC Throughput Test (1000Mbps)](image)

(b) 1000Mbps

Figure 4.13: CloudMAC latency test

Figure 4.13 shows two subfigures, the right one is using a 100 Mbps (rate limited) connection and the left one is using a 1000 Mbps connection. Both perform the same TCP test. The y-axis the bitrate of the connection as reported from netperf\text{-}wrapper. The x-axis represents the setup used (notc = no traffic control, nc = no cross-traffic). Each figure has five different bars. The gray bar (notc-nc) is without traffic control or cross-traffic. The other gray bar is without traffic control, but with cross-traffic. The blue ones are using HTB (with cross-traffic), red SFQ (with cross-traffic), green CoDel (with cross-traffic), purple FQ_CoDel (with cross-traffic).

This test, as opposed to the latency test are not similar to the connection success rate test. So it could be assumed that the there is little to no correlation between the bitrate and the connection success rate. In the 100 Mbps test, HTB appears to be the best traffic control mechanism when it comes to providing bitrate for CloudMAC networks. CoDel, SFQ and FQ_CoDel do not seem as viable. The reason for this could be that HTB always prioritizes CloudMAC data, while the other algorithms attempt to perform indiscriminating traffic control. However, the results with (non-HTB) traffic control does not provide a significant increase in bitrate over the test without any traffic control.

In the 1000 Mbps test, similar results were achieved. However, here HTB is only marginally better than the indiscriminate protocols. Figures 4.14, 4.15, 4.16 and 4.17 shows more in-detail graphs
of the tests.

Figure 4.14: Bitrate without traffic control, without cross-traffic

Figure 4.15: Bitrate without traffic control, with cross-traffic
Figure 4.16: Bitrate with HTB traffic control

Figure 4.17: Bitrate with FQ_CoDel traffic control
Conclusion

Software Defined Networking has recently become more and more popular. Because of this, protocols like OpenFlow has seen a lot of recent developments. The latest version of OpenFlow does not have any Quality of Service extensions. Traffic control is a requirement in many large networks.

Within the scope of this thesis, a solution to the lack of traffic control in OpenFlow-based networks was devised. A solution using the software switch Open vSwitch was developed. This solution uses the Open vSwitch database to configure the traffic control options in the network.

CloudMAC is also a recent development in wireless technology. It allows offloading the wireless mac layer processing to an external machine, which allows for increased processing power, seamless handovers and a unified, centralized configuration interface for wireless access points.

However, one of the problems with CloudMAC was that the users of the network noticed, from time to time, that connecting to the network failed. It was theorized that the cause behind this problem was that the cross-traffic on the network CloudMAC ran upon caused the connection failures. This due to the strict latency requirements in wireless networks, combined with the additional latency introduced by CloudMAC.

Tests were made to observe the changes in CloudMAC's connection success rate was significantly lowered by the cross-traffic in the network. The tests did show that the rate dropped from 99% to 43% during a time of heavy network load. As connection success rate is important to the usability of the network, it had to be ensured.

As CloudMAC runs entirely in an OpenFlow based network, the traffic control extensions that were made to Open vSwitch were used to test if traffic control could affect the connection success rate of the CloudMAC network. It turned out that traffic control did help ensure a more stable connection success rate.

Furthermore, a close relation between the network latency was noticed. When the latency was lowered, the connection success rate seemed to increase. This helps to confirm the hypothesis that the delay caused by (among other things) cross-traffic was part of the reason why CloudMAC users experienced connection failures.

Using HTB as traffic control brought the success rate up to 99% again, SFQ and FQ_CoDel allowed
for a success rate of 97% and 98% respectively. With that, it can be concluded that Quality of Service level (particularly the latency) is very important for a CloudMAC network.
Bibliography


Complete QoS implementations

The SFQ qdisc is already detailed in the thesis.

A.1 CoDel qdisc

```c
/* CoDel traffic control class. */

#define CODEL_N_QUEUES 0xf000

struct codel {
    struct tc tc;
    uint32_t target;
    uint32_t limit;
    uint32_t interval;
};

static struct codel *
codel_get__(const struct netdev *netdev_)
{
    struct netdev_linux *netdev = netdev_linux_cast(netdev_);
    return CONTAINER_OF(netdev->tc, struct codel, tc);
}

static void
codel_install__(struct netdev *netdev_, uint32_t target, uint32_t limit, uint32_t interval)
{
    struct netdev_linux *netdev = netdev_linux_cast(netdev_);
    struct codel *codel;
    codel = xmalloc(sizeof *codel);
    tc_init(&codel->tc, &tc_ops_codel);
    codel->target = target;
    codel->limit = limit;
    codel->interval = interval;
    netdev->tc = &codel->tc;
}
```
static int
codel_setup_qdisc__(struct netdev *netdev, uint32_t target, uint32_t limit, uint32_t interval)
{
    size_t opt_offset;
    struct ofpbuf request;
    struct tcmsg *tcmsg;
    uint32_t otarget, olimit, ointerval;
    int error;

tc_del_qdisc(netdev);

tcmsg = tc_make_request(netdev, RTM_NEWQDISC,
    NLM_F_EXCL | NLM_F_CREATE, &request);
if (!tcmsg) {
    return ENODEV;
}
tcmsg->tcm_handle = tc_make_handle(1, 0);
tcmsg->tcm_parent = TC_H_ROOT;

otarget = target ? target : 5;
olimit = limit ? limit : 10240;
ointerval = interval ? interval : 100;

nl_msg_put_string(&request, TCA_KIND, "codel");
opt_offset = nl_msg_start_nested(&request, TCA_OPTIONS);
nl_msg_put_unspec(&request, TCA_CODEL_TARGET, &otarget, sizeof otarget);
nl_msg_put_unspec(&request, TCA_CODEL_LIMIT, &olimit, sizeof olimit);
nl_msg_put_unspec(&request, TCA_CODEL_INTERVAL, &ointerval, sizeof ointerval);

error = tc_transact(&request, NULL);
if (error)
    VLOG_WARN_RL(&rl, "failed to replace %s qdisc, 
    "target %u, limit %u, interval %u error %d(%s)",
    netdev_get_name(netdev),
    otarget, olimit, ointerval,
    error, strerror(error));
return error;
}

static void
codel_parse_qdisc_details__(struct netdev *netdev OVS_UNUSED,
        const struct smap *details, struct codel *codel)
{
    const char *target_s;
    const char *limit_s;
    const char *interval_s;

target_s = smap_get(details, "target");
limit_s = smap_get(details, "limit");
interval_s = smap_get(details, "interval");
codel->target = target_s ? strtoull(target_s, NULL, 10) : 0;
codel->limit = limit_s ? strtoull(limit_s, NULL, 10) : 0;
codel->interval = interval_s ? strtoull(interval_s, NULL, 10) : 0;
if (!codel->target) codel->target = 5;
if (!codel->limit) codel->limit = 10240;
if (!codel->interval) codel->interval = 100;
}

static int
codel_tc_install(struct netdev *netdev, const struct smap *details)
{
    int error;
    struct codel codel;
    codel_parse_qdisc_details__(netdev, details, &codel);
    error = codel_setup_qdisc__(netdev, codel.target, codel.limit, codel.interval);
    if (!error) {
        codel_install__(netdev, codel.target, codel.limit, codel.interval);
    }
    return error;
}

static int
codel_tc_load(struct netdev *netdev, struct ofpbuf *nlmsg OVS_UNUSED)
{
    struct codel codel;
    /* Get qdisc options. */
    codel.target = 5;
    codel.limit = 10240;
    codel.interval = 100;
    /* TODO: find real values of qdisc instead of class
    codel_query_class__(netdev, tc_make_handle(1, 0xffff), 0, &hc, NULL);
    */
    codel_install__(netdev, codel.target, codel.limit, codel.interval);
    return 0;
}

static void
codel_tc_destroy(struct tc *tc)
{
    struct codel *codel = CONTAINER_OF(tc, struct codel, tc);
    tc_destroy(tc);
    free(codel);
}

static int
codel_qdisc_get(const struct netdev *netdev, struct smap *details)
{
    const struct codel *codel = codel_get__(netdev);
    smap_add_format(details, "target", "%u", codel->target);
    smap_add_format(details, "limit", "%u", codel->limit);
    smap_add_format(details, "interval", "%u", codel->interval);
    return 0;
}
static int
codel_qdisc_set(struct netdev *netdev, const struct smap *details)
{
    struct codel codel;
    int error;

codel_parse_qdisc_details__(netdev, details, &codel);
codel_install__(netdev, codel.target, codel.limit, codel.interval);
codel_get__(netdev)->target = codel.target;
codel_get__(netdev)->limit = codel.limit;
codel_get__(netdev)->interval = codel.interval;
    return error;
}

static int
codel_class_get(const struct netdev *netdev, const struct tc_queue *queue OVS_UNUSED, struct smap *details)
{
    const struct codel *codel = codel_get__(netdev);
    smap_add_format(details, "target", "%u", codel->target);
    smap_add_format(details, "limit", "%u", codel->limit);
    smap_add_format(details, "interval", "%u", codel->interval);
return 0;
}

static int
codel_class_set(struct netdev *netdev OVS_UNUSED, unsigned int queue_id OVS_UNUSED, const struct smap *details OVS_UNUSED)
{
    return 0;
}

static int
codel_class_delete(struct netdev *netdev OVS_UNUSED, struct tc_queue *queue OVS_UNUSED)
{
    return 0;
}

static const struct tc_ops tc_ops_codel = {
    "codel",    /* linux_name */
    "linux-codel",    /* ovs_name */
    CODEL_N_QUEUES,    /* n_queues */
    codel_tc_install,
    codel_tc_load,
    codel_tc_destroy,
    codel_qdisc_get,
    codel_qdisc_set,
    codel_class_get,
    codel_class_set,
    codel_class_delete,
    NULL,
    NULL
}
A.2 FQ_CoDel qdisc

/* FQ-CoDel traffic control class. */

#define FQCODEL_N_QUEUES 0xf000

struct fqcodel {
    struct tc tc;
    uint32_t target;
    uint32_t limit;
    uint32_t interval;
    uint32_t flows;
    uint32_t quantum;
};

static struct fqcodel *
fqcodel_get__(const struct netdev *netdev_)
{
    struct netdev_linux *netdev = netdev_linux_cast(netdev_);
    return CONTAINER_OF(netdev->tc, struct fqcodel, tc);
}

static void
fqcodel_install__(struct netdev *netdev_, uint32_t target, uint32_t limit, uint32_t interval, uint32_t flows, uint32_t quantum)
{
    struct netdev_linux *netdev = netdev_linux_cast(netdev_);
    struct fqcodel *fqcodel;
    fqcodel = xmalloc(sizeof *fqcodel);
    tc_init(&fqcodel->tc, &tc_ops_fqcodel);
    fqcodel->target = target;
    fqcodel->limit = limit;
    fqcodel->interval = interval;
    fqcodel->flows = flows;
    fqcodel->quantum = quantum;
    netdev->tc = &fqcodel->tc;
}

static int
fqcodel_setup_qdisc__(struct netdev *netdev, uint32_t target, uint32_t limit, uint32_t interval, uint32_t flows, uint32_t quantum)
{
    size_t opt_offset;
    struct ofpbuf request;
    struct tcmsg *tcmsg;
    uint32_t otarget, olimit, ointerval, oflows, oquantum;
    int error;
tc_del_qdisc(netdev);

tcmsg = tc_make_request(netdev, RTM_NEWQDISC,
    NLM_F_EXCL | NLM_F_CREATE, &request);
if (!tcmsg) {
    return ENODEV;
}
tcmsg->tcm_handle = tc_make_handle(1, 0);
tcmsg->tcm_parent = TC_H_ROOT;

otarget = target ? target : 5;
olimit = limit ? limit : 10240;
ointerval = interval ? interval : 100;
oflows = flows ? flows : 1924;
oquantum = quantum ? quantum : 1514; /* fq_codel default quantum is 1514
not mtu */
nl_msg_put_string(&request, TCA_KIND, "fq_codel");
opt_offset = nl_msg_start_nested(&request, TCA_OPTIONS);
nl_msg_put_unspec(&request, TCA_FQ_CODEL_TARGET, &otarget, sizeof otarget);
nl_msg_put_unspec(&request, TCA_FQ_CODEL_LIMIT, &olimit, sizeof olimit);
nl_msg_put_unspec(&request, TCA_FQ_CODEL_INTERVAL, &ointerval, sizeof ointerval);
nl_msg_put_unspec(&request, TCA_FQ_CODEL_FLOWS, &oflows, sizeof oflows);
nl_msg_put_unspec(&request, TCA_FQ_CODEL_QUANTUM, &oquantum, sizeof oquantum);
nl_msg_end_nested(&request, opt_offset);
error = tc_transact(&request, NULL);
if (error)
    VLOG_WARN_RL(&rl, "failed to replace %s qdisc, 
" "target %u, limit %u, interval %u, flows %u, quantum %u error %d(%s)",
netdev_get_name(netdev),
    otarget, olimit, ointerval, oflows, oquantum,
    error, strerror(error));
return error;

static void
fq_codel_parse_qdisc_details__(struct netdev *netdev OVS_UNUSED,
    const struct smap *details, struct fqcodel *fqcodel)
{
    const char *target_s;
    const char *limit_s;
    const char *interval_s;
    const char *flows_s;
    const char *quantum_s;

target_s = smap_get(details, "target");
limit_s = smap_get(details, "limit");
interval_s = smap_get(details, "interval");
flows_s = smap_get(details, "flows");
quantum_s = smap_get(details, "quantum");

    fqcodel->target = target_s ? strtoull(target_s, NULL, 10) : 0;
    fqcodel->limit = limit_s ? strtoull(limit_s, NULL, 10) : 0;
    fqcodel->interval = interval_s ? strtoull(interval_s, NULL, 10) : 0;
    fqcodel->flows = flows_s ? strtoull(flows_s, NULL, 10) : 0;
static int fqcodel_qdisc_get(const struct netdev *netdev, struct smap *details)
{
    struct fqcodel fqcodel;
    /* Get qdisc options. */
    fqcodel.target = 5;
    fqcodel.limit = 10240;
    fqcodel.interval = 100;
    fqcodel.flows = 1024;
    fqcodel.quantum = 1514;
    /* TODO: find real values of qdisc instead of class
    fqcodel_query_class__(netdev, tc_make_handle(1, 0xffff), 0, &hc, NULL); */
    fqcodel_install__(netdev, fqcodel.target, fqcodel.limit, fqcodel.interval, fqcodel.flows, fqcodel.quantum);
    return 0;
}

static void fqcodel_tc_destroy(struct tc *tc)
{
    struct fqcodel *fqcodel = CONTAINER_OF(tc, struct fqcodel, tc);
    tc_destroy(tc);
    free(fqcodel);
}

static int fqcodel_tc_load(struct netdev *netdev, struct ofpbuf *nlmsg OVS_UNUSED)
{
    struct fqcodel fqcodel;
    /* Get qdisc options. */
    fqcodel.target = 5;
    fqcodel.limit = 10240;
    fqcodel.interval = 100;
    fqcodel.flows = 1024;
    fqcodel.quantum = 1514;
    /* TODO: find real values of qdisc instead of class
    fqcodel_query_class__(netdev, tc_make_handle(1, 0xffff), 0, &hc, NULL); */
    fqcodel_install__(netdev, fqcodel.target, fqcodel.limit, fqcodel.interval, fqcodel.flows, fqcodel.quantum);
    return 0;
}

static int fqcodel_tc_install(struct netdev *netdev, const struct smap *details)
{
    int error;
    struct fqcodel fqcodel;
    /* Get qdisc details. */
    fqcodel_parse_qdisc_details__(netdev, details, &fqcodel);
    if (!error) {
        fqcodel_setup_qdisc__(netdev, fqcodel.target, fqcodel.limit, fqcodel.interval, fqcodel.flows, fqcodel.quantum);
        fqcodel_install__(netdev, fqcodel.target, fqcodel.limit, fqcodel.interval, fqcodel.flows, fqcodel.quantum);
    }
    return error;
}

static int fqcodel_tc_destroy(struct tc *tc)
{
    struct fqcodel *fqcodel = CONTAINER_OF(tc, struct fqcodel, tc);
    tc_destroy(tc);
    free(fqcodel);
}

static int fqcodel_qdisc_get(const struct netdev *netdev, struct smap *details)
const struct fqcodel *fqcodel = fqcodel_get__(netdev);
smap_add_format(details, "target", "%u", fqcodel->target);
smap_add_format(details, "limit", "%u", fqcodel->limit);
smap_add_format(details, "interval", "%u", fqcodel->interval);
smap_add_format(details, "flows", "%u", fqcodel->flows);
smap_add_format(details, "quantum", "%u", fqcodel->quantum);
return 0;
}

static int
fqcodel_qdisc_set(struct netdev *netdev, const struct smap *details)
{
    struct fqcodel fqcodel;
    int error;

    fqcodel_parse_qdisc_details__(netdev, details, &fqcodel);
    fqcodel_install__(netdev, fqcodel.target, fqcodel.limit, fqcodel.interval, fqcodel.flows, fqcodel.quantum);
    fqcodel_get__(netdev)->target = fqcodel.target;
    fqcodel_get__(netdev)->limit = fqcodel.limit;
    fqcodel_get__(netdev)->interval = fqcodel.interval;
    fqcodel_get__(netdev)->flows = fqcodel.flows;
    fqcodel_get__(netdev)->quantum = fqcodel.quantum;
    return error;
}

static int
fqcodel_class_get(const struct netdev *netdev,
                   const struct tc_queue *queue OVS_UNUSED, struct smap *details)
{
    const struct fqcodel *fqcodel = fqcodel_get__(netdev);
    smap_add_format(details, "target", "%u", fqcodel->target);
    smap_add_format(details, "limit", "%u", fqcodel->limit);
    smap_add_format(details, "interval", "%u", fqcodel->interval);
    smap_add_format(details, "flows", "%u", fqcodel->flows);
    smap_add_format(details, "quantum", "%u", fqcodel->quantum);
    return 0;
}

static int
fqcodel_class_set(struct netdev *netdev OVS_UNUSED, unsigned int queue_id OVS_UNUSED,
                   const struct smap *details OVS_UNUSED)
{
    return 0;
}

static int
fqcodel_class_delete(struct netdev *netdev OVS_UNUSED, struct tc_queue *queue OVS_UNUSED)
{
    return 0;
}
static const struct tc_ops tc_ops_fqcodel = {
    "fq_codel",
    "linux-fq-codel",
    FQCODEL_N_QUEUES,
    fqcode_tc_install,
    fqcode_tc_load,
    fqcode_tc_destroy,
    fqcode_qdisc_get,
    fqcode_qdisc_set,
    fqcode_class_get,
    fqcode_class_set,
    fqcode_class_delete,
    NULL,
    NULL
};

A.3 RED qdisc

#define RED_N_QUEUES 0xf000

struct red {
    struct tc tc;
    uint32_t limit;    /* HARD maximal queue length (bytes) */
    uint32_t min;      /* Min average length threshold (bytes) */
    uint32_t max;      /* Max average length threshold (bytes) */
    uint32_t burst;    /* How fast average queue is influenced */
    uint32_t avpkt;    /* Average packets */
    uint32_t rate;     /* Bandwidth of interface (bps) */
}; /* TODO: flags:ecn, adaptive and harddrop */

static struct red *
red_get__(const struct netdev *netdev_)
{
    struct netdev_linux *netdev = netdev_linux_cast(netdev_);
    return CONTAINER_OF(netdev->tc, struct red, tc);
}

static void
red_install__(struct netdev *netdev_, uint32_t limit, uint32_t min, uint32_t max,
              uint32_t burst, uint32_t avpkt, uint32_t rate)
{
    struct netdev_linux *netdev = netdev_linux_cast(netdev_);
    struct red *red;

    red = xmalloc(sizeof *red);
    tc_init(&red->tc, &tc_ops_red);
    red->limit = limit;
    red->min = min;
    red->max = max;
    red->avpkt = avpkt;
red->burst = burst;
red->rate = rate;
netdev->tc = &red->tc;
}

static int
red_setup_qdisc__(struct netdev *netdev, uint32_t limit, uint32_t min, uint32_t max,
uint32_t avpkt, uint32_t burst, uint32_t rate)
{
    size_t opt_offset;
    struct ofpbuf request;
    struct tcmsg *tcmsg;
    struct tc_red_qopt opt;
    int error, wlog, max_P;
    __u8 sbuf[256];
    double probability = 0.02;

tc_del_qdisc(netdev);

tcmsg = tc_make_request(netdev, RTM_NEWQDISC,
            NLM_F_EXCL | NLM_F_CREATE, &request);
    if (!tcmsg) {
        return ENODEV;
    }
    tcmsg->tcm_handle = tc_make_handle(1, 0);
    tcmsg->tcm_parent = TC_H_ROOT;

    if (avpkt == 0 || limit == 0)
    {
        VLOG_WARN_RL(&rl, "RED: required parameter avpkt missing.");
        return EINVAL;
    }
    rate = rate ? rate : 10000000;
    opt.limit = limit;
    opt.qth_max = max ? max : opt.limit / 4;
    opt.qth_min = min ? min : opt.qth_max / 4;
    if (!burst)
        burst = (2 * opt.qth_min + opt.qth_max) / (3 * avpkt);
    if ((wlog = tc_red_eval_ewma(opt.qth_min, burst, avpkt)) < 0) {
        VLOG_WARN_RL(&rl, "RED: failed to calculate EWMA constant.");
        return 1;
    }
    if (wlog >= 10)
    {
        VLOG_WARN_RL(&rl, "RED: WARNING. Burst %d seems to be too large.", burst);
        opt.Wlog = wlog;
        if ((wlog = tc_red_eval_P(opt.qth_min, opt.qth_max, probability)) < 0) {
            VLOG_WARN_RL(&rl, "RED: failed to calculate probability.");
            return 1;
        }
        opt.Plog = wlog;
        if ((wlog = tc_red_eval_idle_damping(opt.Wlog, avpkt, rate, sbuf)) < 0) {
            VLOG_WARN_RL(&rl, "RED: failed to calculate idle damping table.");
            return 1;
        }
    }

57
opt.Scell_log = wlog;

opt.flags = 0;

nl_msg_put_string(&request, TCA_KIND, "red");

nl_msg_put_unspec(&request, TCA_OPTIONS, NULL, 0);

nl_msg_put_unspec(&request, TCA_RED_PARMS, &opt, sizeof opt);

max_P = probability * pow(2, 32);

nl_msg_put_unspec(&request, TCA_RED_MAX_P, &max_P, sizeof(max_P));

error = tc_transact(&request, NULL);

if (error)
    VLOG_WARN_RL(&rl, "failed to replace %s qdisc, 
    "limit %u, qth_max %u, qth_min %u, 
    "wlog %d, plog %d, scell_log %d error %d(%)",
    netdev_get_name(netdev),
    opt.limit, opt.qth_max, opt.qth_min,
    opt.Wlog, opt.Plog, opt.Scell_log,
    error, strerror(error));

return error;

static void
red_parse_qdisc_details__(struct netdev *netdev OVS_UNUSED,
const struct smap *details,
struct red *red)
{
    const char *limit_s;
    const char *min_s;
    const char *max_s;
    const char *avpkt_s;
    const char *burst_s;
    const char *rate_s;

    limit_s = smap_get(details, "limit");
    min_s = smap_get(details, "min");
    max_s = smap_get(details, "max");
    avpkt_s = smap_get(details, "avpkt");
    burst_s = smap_get(details, "burst");
    rate_s = smap_get(details, "bandwidth");
    red->limit = limit_s ? strtoull(limit_s, NULL, 10) : 0;
    red->min = min_s ? strtoull(min_s, NULL, 10) : 0;
    red->max = max_s ? strtoull(max_s, NULL, 10) : 0;
    red->avpkt = avpkt_s ? strtoull(avpkt_s, NULL, 10) : 0;
    red->burst = burst_s ? strtoull(burst_s, NULL, 10) : 0;
    red->rate = rate_s ? strtoull(rate_s, NULL, 10) : 0;

    if (!red->min) red->min = red->limit / 12;
    if (!red->max) red->max = red->limit / 4;
    if (!red->burst) red->burst = 1;
    if (!red->rate) red->rate = 1;
}

static int
red_tc_install(struct netdev *netdev, const struct smap *details)

{  
    int error;
    struct red red;
    
    red_parse_qdisc_details__(netdev, details, &red);
    error = red_setup_qdisc__(netdev, red.limit, red.min, red.max, red.burst, red.avpkt, red.rate);
    if (!error) {
        red_install__(netdev, red.limit, red.min, red.max, red.burst, red.avpkt, red.rate);
    }
    return error;
}

static int red_tc_load(struct netdev *netdev, struct ofpbuf *nlmsg OVS_UNUSED)
{
    struct red red;
    
    /* Get qdisc options. */
    red.limit = 100000;
    red.avpkt = 1000;
    red.burst = red.rate = 0;
    /* TODO: find real values of qdisc instead of class
    red_query_class__(netdev, tc_make_handle(1, 0xfffe), 0, &hc, NULL);
    */
    red_install__(netdev, red.limit, red.min, red.max, red.burst, red.avpkt, red.rate);
    
    return 0;
}

static void red_tc_destroy(struct tc *tc)
{
    struct red *red = CONTAINER_OF(tc, struct red, tc);
    tc_destroy(tc);
    free(red);
}

static int red_qdisc_get(const struct netdev *netdev, struct smap *details)
{
    const struct red *red = red_get__(netdev);
    
    smap_add_format(details, "limit", "%u", red->limit);
    smap_add_format(details, "min", "%u", red->min);
    smap_add_format(details, "max", "%u", red->max);
    smap_add_format(details, "burst", "%u", red->burst);
    smap_add_format(details, "avpkt", "%u", red->avpkt);
    smap_add_format(details, "bandwidth", "%u", red->rate);
    return 0;
}
static int
red_qdisc_set(struct netdev *netdev, const struct smap *details)
{
    struct red red;
    int error;
    red_parse_qdisc_details__(netdev, details, &red);
    red_install__(netdev, red.limit, red.min, red.max, red.burst, red.avpkt, red.rate);
    red_get__(netdev)->limit = red.limit;
    red_get__(netdev)->min = red.min;
    red_get__(netdev)->max = red.max;
    red_get__(netdev)->burst = red.burst;
    red_get__(netdev)->avpkt = red.avpkt;
    red_get__(netdev)->rate = red.rate;
    return error;
}

static int
red_class_get(const struct netdev *netdev, const struct tc_queue *queue OVS_UNUSED, struct smap *details)
{
    return red_qdisc_get(netdev, details);
}

static int
red_class_set(struct netdev *netdev OVS_UNUSED, unsigned int queue_id OVS_UNUSED, const struct smap *details OVS_UNUSED)
{
    return 0;
}

static int
red_class_delete(struct netdev *netdev OVS_UNUSED, struct tc_queue *queue OVS UNUSED)
{
    return 0;
}

static const struct tc_ops tc_ops_red = {
    "red",            /* linux_name */
    "linux-red",      /* ovs_name */
    RED_N_QUEUES,     /* n_queues */
    red_tc_install,   
    red_tc_load,      
    red_tc_destroy,   
    red_qdisc_get,    
    red_qdisc_set,    
    red_class_get,    
    red_class_set,    
    red_class_delete, 
    NULL,             
    NULL
};

Code Listing A.3: RED support implementation
Test scripts

B.1 CloudMAC Connection Test Script

```python
import os, sys, subprocess, time

if len(sys.argv) < 3:
    print "./condcon.py <SSID> <output file>"
    sys.exit(0)

def run(cmd):
    proc = subprocess.Popen("" + cmd, stdout=subprocess.PIPE, shell=True)
    (out, err) = proc.communicate()
    return out

ssid = sys.argv[1]
outfile = sys.argv[2]
print "Disconnecting from %s" % ssid
f = open(outfile, 'w')
run("sudo iw wifi0 disconnect")

print "WiFi Test Actual"
for i in range(1, 100):
    print "cycle", i
    print "removing drivers"
    run("sudo rmmod brcmsmac")
    run("sudo rmmod b43")
    time.sleep(1)
    print "inserting drivers"
    run("sudo modprobe b43")
    run("sudo modprobe brcmsmac")
    time.sleep(1)
    res = run("sudo ifconfig wlan0 up")
    res = run("sudo iw wlan0 connect -w %s" % ssid)
    f.write(str(time.time()) + " " + res)
    print res,
    res = run("sudo iw wlan0 disconnect")
```

Code Listing B.1: Connection Test Script
#!/usr/bin/python

import sys, os
import pexpect
import time

def processfile(infname, outfname):
    f = open(infname)
    lines = 0
    items = []
    head = False
    for l in f.read().split('
'):
        if l.strip() == '':
            continue
        lines += 1
        p = int(l.split(' ')[1][:-1])
        if p == 0 and not head:
            continue
        if p != 0:
            head = True
            items.append(p)
        else:
            break
    f.close()
    f = open(outfname, 'w')
    f.write("cpu\n")
    for i in items:
        f.write("%s\n" % i)
    f.close()
    else:
        len(items) == 0:
        print "No CPU Usage"
        else:
            print "Result: Lines = %d, Items = %d, Average = %d%% CPU" % \
                (lines, len(items), reduce(lambda x, y: x + y, items) / len(items))

def cmd(c, what):
    c.expect("# ")
    c.sendline(what)

if len(sys.argv) < 5:
    print "./queue.py <qdiscrate> <queuerate> <queuecount> <runtime>"
    exit(0)

discrate = int(sys.argv[1])
queuerate = int(sys.argv[2])
queuecount = int(sys.argv[3])
runtime = int(sys.argv[4])

print "Setting up Switch"
c = pexpect.spawn('ssh root@192.168.1.12')
#Destroy all previous QoS
for i in range(0, 5):
    cmd(c, "ovs-vsct1 clear port eth0.%d qos" % (i))
```
cmd(c, "ovs-vsctl --all destroy qos")

#Setup switch with two ports
cmd(c, "ovs-vsctl del-br br0")

#Delete default flows
cmd(c, "ovs-ofctl del-flows br0")

#Add flows for port 1 -> 2
for i in range(0, queuecount+1):
    cmd(c, 'ovs-ofctl add-flow br0 "in_port=1,dl_type=0x0800,nw_proto=6,tp_dst=560%d,actions=enqueue:2:%d" % (i+1, i))

#Add flow for 2 -> 1
cmd(c, 'ovs-ofctl add-flow br0 "in_port=2,actions=output:1"')

#Build QoS Command
#Add QoS at egress port 2 type HTB rate qdiscrate
q = "ovs-vsctl -- set port eth0.2 qos=@newqos -- id=@newqos create qos type=linux-HTB other-config:max-rate=%d queues=" % (qdiscrate)

#Build QoS queue list
for i in range(0, queuecount):
    q += "%d=@q%d," % (i, i+1)
q = q[:-1]

#Add queues
for i in range(0, queuecount):
    q += "--id=@q%d create queue other-config:max-rate=%d other-config:min-rate=%d " % (i+1, queuerate, queueerate)

#Run queue setup command
cmd(c, q)

#Setup sirq monitor
print "Setting up cpu monitor"

# Kill processes
```
for i in range(0, queuecount):
    os.system('iperf -c 10.0.0.2 -p 560%d -t %d & ' % (i+1, runtime))
    print "Running iperf port", i+1

time.sleep(runtime)
print "Waiting 5 seconds before terminating flows..."
time.sleep(5)
os.system('pkill iperf')

print "Shutting down cpu monitor"
c = pexpect.spawn('ssh root@192.168.1.12')
cmd(c, "kill `ps | grep sirq_monitor | head -n 1 | awk '{ print $1 }'")
c.expect("# ")
c.kill(1)
print "Copying result files.."
os.system('scp root@192.168.1.12:sirq.txt .')
processfile("sirq.txt", "sirq_queue_%d.txt" % (queuecount))