# DYNAMIC QUALITY OF SERVICE IN SOFTWARE-DEFINED NETWORKS

An Undergraduate Research Scholars Thesis

by

THOMAS STEP

Submitted to the Undergraduate Research Scholars program at
Texas A&M University
in partial fulfillment of the requirements for the designation as an

UNDERGRADUATE RESEARCH SCHOLAR

Approved by Research Advisor: Dr. Guofei Gu

May 2018

Major: Computer Engineering

# TABLE OF CONTENTS

# ABSTRACT

Dynamic Quality of Service in Software-defined Networks

Thomas Step
Department of Computer Science and Engineering
Texas A&M University

Research Advisor: Dr. Guofei Gu
Department of Computer Science and Engineering
Texas A&M University

Quality of service is a necessary function of today's networks. A proper quality of service ensures that packets are delivered effectively and fast. In traditional networks, quality of service has to be manually configured on each piece of hardware in the network. This manual procedure makes the process of implementing a quality of service in a network costly. Not to mention, if part of the configuration is incorrect, or a mistake is made during the configuration, everything must be corrected on each piece of affected hardware. In this paper, I will be exploring the effect of using a software-defined network controller and a quality of service to handle certain flows of traffic in a network. The main tool used is the OpenFlow defined queue. Queues and flow rules will allow a switch to control individual flows and the network resources that each flow consumes. Factors that will be explored are the bandwidth usage of a flow and the time taken by the network to implement new flow rules. While bandwidth usage is taken into account in traditional networks, changing a quality of service is a new dynamic.

# DEDICATION

This is dedicated to my mother and father who encouraged me throughout the process as well as throughout my years studying at Texas A&M University, and to Bernard Natho who introduced me to Texas A&M University and has been a supporter of my work and studies.

# ACKNOWLEDGEMENTS

I would like to thank my honors advisor, Dr. Welch, for helping and supporting me during my years here at Texas A&M University.

Another thank you to my father who helped me through the process of research and writing a thesis.

# NOMENCLATURE

| | |
|---|---|
| QoS | Quality of Service |
| VoIP | Voice Over IP |
| SDN | Software-defined Network |
| OVS | Open vSwitch |
| Mbps | Megabit per second |
| Gbps | Gigabit per second |

# CHAPTER I

# INTRODUCTION

Networks deal with traffic from different connections made from one computer to another. Traffic is made up of a continuous stream of pieces of data. Those pieces of data are called packets and can consist of pieces of a website being downloaded, frames of a video being streamed, or portions of a voice call happening over the Internet. Another name for traffic is flow. Flows are the basis of how the software-defined networking protocol OpenFlow is able to control a network. A network needs a quality of service in order to utilize its resources to the fullest extent, and a key component in a quality of service is identifying the flows that need to be slowed down or allowed to move through the network as fast as they can.

Quality of service is a necessary functionality of today's networks. It provides certain types of traffic with improved bandwidth, less latency, or less jitter depending on the type of traffic. With Voice over IP and video streaming becoming more and more popular, quality of service is more relied upon to keep our networks running smoothly. For example, VoIP traffic needs minimal latency in order to make a phone call run smoothly and not sound jumpy. A good quality of service will note which packets belong to a VoIP call and make sure that those packets are forwarded first to minimize latency.

In traditional networks, quality of service is manually configured on each router and switch. If the quality of service needs to change or is incorrectly configured on a piece of hardware, then someone needs to manually reconfigure it. This additional work is time consuming and prone to human error. One comprehensive way of solving a problem such as this is to use a software-defined network. A software-defined network gives an engineer more control

over an entire network from a central location because the control plane is abstracted away from the data plane.

The attractiveness of software-defined networks is the decoupling of the control plane and the data plane. Instead of having decisions made on each individual piece of networking hardware, the decision making is done by a controller. The controller connects to each switch in the network and communicates with those switches using the OpenFlow protocol. The protocol allows the controller to relay any updates in the decision making, thus making configuration of an entire network more simplistic for an engineer. The switches in the network solely forward packets based on the rules given to them by the controller.

Figure 1 shows a simple linear topology with three switches and three hosts from a traditional network. Something to note here is the lack of a central control plane. Instead, each switch possesses its own control plane for decision making. It is not explicitly highlighted, but the control plane and the data plane are contained within each switch.

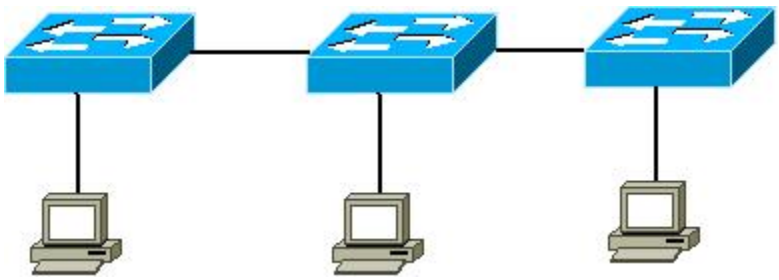

Figure 1. Linear Topology in a Traditional Network Environment.

As previously discussed, each switch's control plane must be configured before it can become part of the network. As a network grows the amount of overall time spent configuring switches increases with that growth.

Figure 2 shows the same linear topology but in a software-defined network environment. The biggest difference here is the centralized control plane, the network controller. The controller is represented as a server connected to each switch in the network.

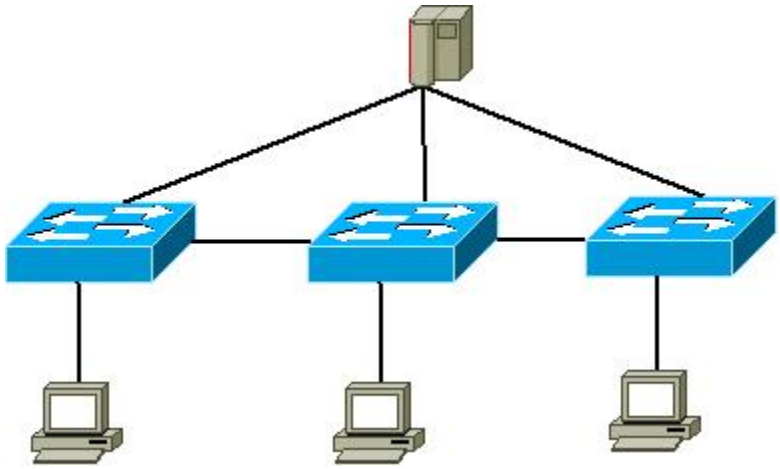

Figure 2. Linear Topology in a SDN Environment.

Since the controller is responsible for all of the decision making, the switches are left only with moving information around the network. Data would move just as it would in the traditional network, but the controller's connection to each switch enables the decision making changes to be communicated. The traditional idea of quality of service positively affecting how traffic moves through the network still applies because of this. The difference is the way in which a network engineer can tell the switches in the network of a change in their configuration. The SDN environment allows the network to be changed from the singular control point, while the traditional environment requires reconfiguration at every switch.

In order to setup quality of service on switches that speak using the OpenFlow protocol, the switch must first have queues added to it manually. This does not take much time, and other necessary setup can be completed at the same time that the queues are added. After the switch is

setup, it can be deployed into the network. At this point in time, the controller would start to communicate with the switch to send and receive messages using the OpenFlow protocol. The controller can be programed to tell the switch of a certain quality of service policy based on different events that have transpired or a given situation within the network.

During the configuration of a quality of service on a traditional network the priorities of certain traffic must be statically defined and cannot change throughout the course of a certain deployment unless every single switch and router is reconfigured. With a software-defined network, multiple policies with different priorities for different types of traffic can be interchanged by simply writing a program for the controller. Some events that may warrant a need for change in the quality of service would be companywide emails, streaming events, or periodic updates. In statically defined quality of services, the priorities of those certain types of traffic might not be high which would cause the traffic to fall behind in queues if the network was already congested. If a network were told to let those certain types of traffic through before they were flooded into the network, the overall traffic flow would be improved. This research will explore this new concept. During certain times of the day or during certain events, a network may benefit by changing the priorities of different traffic flows in order to run more efficiently.

This idea stems from the fascination with software-defined networks as a whole. Topics surrounding this technology include transforming traditional networking concepts into a more dynamic and programmable setting. Quality of service is one of these traditional networking concepts that can be applied to a software-defined network environment to possibly benefit from the inherent programmability. Traditionally, a quality of service must be carefully planned out by network engineers to meet the needs of a network because it should only be configured on networking hardware once. Part of the planning process is weighing advantages and

disadvantages for putting different priorities on packets. However, now that software-defined networks offer a way to alter an entire network from a central location, quality of service can be thought of in a different light. If a network wide quality of service can be manipulated quickly, the advantages given to certain flows will not detract from other types of flows. The possibility of using a network to its fullest potential at all times is not possible with a static quality of service in a traditional network. There will always be times when the quality of service policies could be better. In a software-defined network with a dynamic quality of service, this is possibly achievable. By changing the policies with respect to a given situation, a network can possibly work to its fullest potential for a greater amount of time than a traditional network could.

Quality of service in software-defined networks has already been a topic discussed within the networking community. Ryan Wallner has worked to create an experimental version of quality of service for the Floodlight controller [1] [3]. The beta version contains type of service as well as other common ways to distinguish traffic in quality of service policies. The module keeps track of policies and pushes any new updates to switches in the network. This module simply applies the traditional concept of quality of service to software-defined networks.

Other researchers have taken the traditional concept of quality of service and completely redesigned it such as OpenQoS [5]. Instead of using the more common methods of either IntServ or DiffServ, OpenQoS reroutes traffic based on a specific flow's needs. This type of quality of service is definitely dynamic in the sense of routing, but it does not involve changing quality of service policies. However, this type of dynamic rerouting in tandem with policy changing could possibly be more beneficial for a network overall.

This research does not redefine quality of service but explores new ways of making the most of an existing QoS with some enhancements. I will be observing how changing a given set

of quality of service policies affects the network as a whole in terms of speed and reliability. As I have previously mentioned, a network may need to change traffic priorities if presented with a given situation. The situation is variable but can include a certain time of day or network wide event. The certain event may benefit from a new network wide quality of service or even just policies in a certain section of that network. Using a software-defined network together with a changing quality of service can possibly increase the efficiency of a network over a given period of time, and I will investigate the feasibility of changing policies to possibly benefit a network in situations like these.

# CHAPTER II

# METHODS

The main testing revolved around how efficiently and reliably we can change the network's quality of service. In an actual network, a network engineer would want to also test how different types of traffic utilized the newfound benefit of a policy. For example, if a server needed to send out updates to every computer on a network, the traffic could be given a higher priority. However, if the traffic never fully utilized the bandwidth freedom that it is given, then the change in policies would be a waste. We would also need to know how or if other flows in the network are negatively affected. Measuring the usefulness of changing certain quality of service policies is hard to quantify. The amount of time saved with a new policy in place versus the old policy is a good indicator of the usefulness of a swap. However, the actual time saved depends heavily on other factors that cannot be controlled by the network controller. Those factors include average bandwidth usage and duration that a policy change is in place. In these tests, I will assume that the policy change is beneficial overall to all flows. This will allow me to simply test the efficiency of those changes.

Another observable measurement is the amount of time taken by the controller to inform the OpenFlow enabled switches regarding the policy exchange. This measurement is strictly overhead. Taking a start time at the controller and measuring the end time when the switch receives and implements the new flow rule can help us measure this overhead. It would be up to the network engineer to determine whether or not the network would benefit from changing the quality of service policies or to just remain in the state that it is in.

In order to test my hypotheses on feasibility, I used a single computer to run the Floodlight controller and a virtualized network using Mininet. The computer used for this research is an Apple MacBook Pro with a 2.4 GHz Intel Core i5 processor with 8 GB of 1600 MHz DDR3 RAM. The laptop was using macOS High Sierra Version 10.13.2 as its operating system. The controller was a forked version of the original Floodlight controller with its own beta version of a quality of service implemented. The exact code can be found at the appropriate GitHub repository [1]. Mininet was used to virtualize a test network. I used a virtual machine that was provided by Mininet at their website [2]. The virtual machine was run using VirtualBox version 5.1.26. I first started the controller using a command from the directory where the controller code was located. After the controller was started I remotely started the Mininet virtualized network by using an SSH session to connect to the virtual machine. One topology used for the network was a linear topology of three OVS switches; each switch had one connected host. Once the network was running and the connectivity was tested using built in Mininet functions, I added queues to the switches using a script that I created. The script adds queues to each switch in the network. This is the one step that has to be configured manually by a network engineer before the switch can be deployed into the network. Most switches now need more configuration than this before they are deployed, so adding in this simple step would most likely not demand more time from an engineer than is already necessary.

I changed the quality of service policies through the REST API provided by Floodlight. I found this to be the most time efficient way of manipulating a quality of service in my case. I wrote a program in Python that allowed me to add and delete rules from the switches' flow tables, while also allowing the network to continue running. This program requires the appropriate queues to already be installed on the switches.

After the switches in the network had the queues implemented on them, I tested the bandwidth of a connection using the iperf tool on the Mininet virtual machine. Iperf is commonly used to test maximum and minimum bandwidth between two connected hosts on a network. Using iperf, I could verify that the queues were correctly configured on the switches that I deployed into the network. Iperf can also verify that the quality of service policy is properly working on the affected flows. By measuring the bandwidth of traffic from one host to another before and after a policy is pushed to a switch, we can determine if the policy was enacted and exactly how long it took to be pushed from the controller to the switch.

Using the three switches in the network, I was able to apply a certain policy to two switches and observe how it affected the network as a whole. The switch that was not directly affected by the policy change experienced some delay due to how the network had to react.

Figure 3 shows the setup for my tests with the three switch linear topology. This topology is similar to Figure 2.

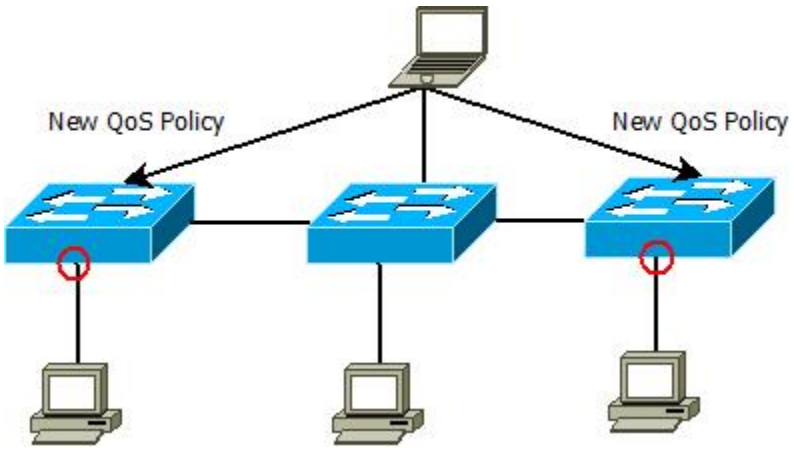


Figure 3. Three Switch Linear Topology Used for Testing.

Symbolizes affected interface.

I have included arrows to show how and where the QoS policy changes were pushed. I also exchanged the server from Figure 2 for a laptop to show that the controller is indeed running from my laptop. The circles at the connection ports for the two outside switches connecting to hosts show where the QoS policy changes were enacted. Furthermore, the policy changes that I enacted were only supposed to affect flows between the two outermost hosts and not the middle host. After the policy changes are communicated from the controller on my laptop to the switches, the switches will enact a rule on the circled interfaces.

I was also able to perform similar functions on a network with more than just three switches. Most enterprise networks have more than three switches, so it was important to test my hypothesis on a larger network as well as the smaller one. For this purpose, I used a network with eight switches. Similar operations and policy changes were performed on this network to observe how the overall network reacted. Of course this is not as large as an enterprise network would be, but I was able to observe how the time of changing policies scaled with a larger network.

Figure 4 shows the eight switch linear topology used in more complicated tests. This is still a linear topology like the one shown in Figure 3, but there are eight switches instead of just three.

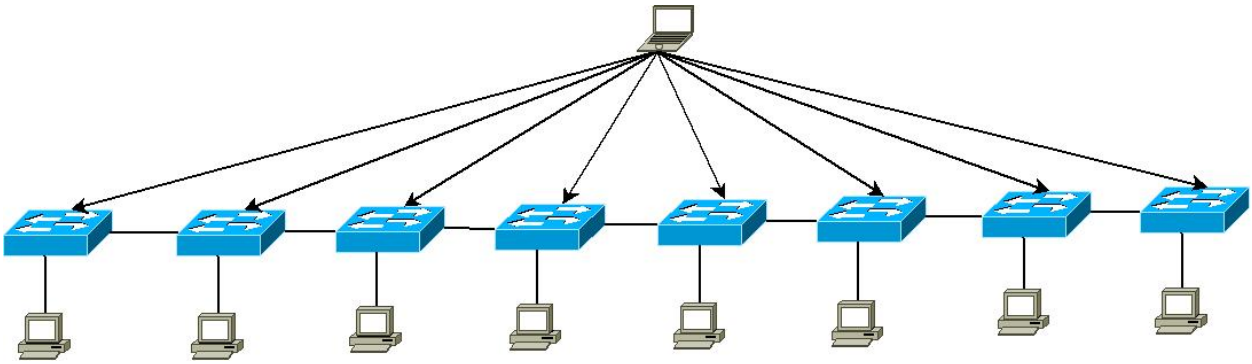

Figure 4. Eight Switch Topology Used for Testing.

Using a larger topology allowed me to test the scalability of changing QoS policies. This topology was also used to test the effects having some flows belonging to a queue and other flows not belonging to a queue. I did not explicitly circle any interfaces in this diagram because I performed different types of changes to this topology.

# CHAPTER III

# RESULTS

Expected and unexpected benefits and drawbacks made themselves clear during testing. I expected that the controller could quickly and easily push a new flow rule to a switch, but I expected the switch to take longer to actually implement the new flow rule. Throughout multiple tests I observed a single policy change taking between 6.1779 and 8.8689 milliseconds as can be seen in Table 1. If a network needs multiple rules changed for multiple switches, this will take longer of course; however, observing a time of less than one second was an unexpected positive result. As I increased the amount of new rules to be pushed to switches, I observed a linear increase in time. Pushing ten rules took me between 63.0262 and 81.9299 milliseconds, and pushing one hundred rules took me between 604.7258 and 763.2890 milliseconds. The linear increase in time makes sense for this problem. The controller must make a connection with a different switch for each new rule and push that rule. There is definitely overhead associated with all of that, but the addition of a new flow rule to a switch's table should be a constant time cost. The results in Table 1 show the time needed to tell the controller of the policy change and actually show a change on the switch for different amounts of policy changes. I ran tests on different amounts of policy changes because I wanted to see how this concept scaled into larger networks.

Table 1. Timing Results (milliseconds) for Changing Different Amounts of Policies.

| | Number of Flow Rules: | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 25 | 50 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Trial: | 1 | 6.1779 | 16.4609 | 20.4630 | 27.0650 | 30.9429 | 40.0970 | 40.3981 | 51.5149 | 75.8531 | 58.3720 | 117.4119 | 313.3199 | 604.7258 |
| | 2 | 7.3559 | 16.2342 | 31.3570 | 43.4651 | 35.1911 | 36.4690 | 47.0691 | 52.7351 | 72.0949 | 73.7610 | 129.1370 | 318.8579 | 763.2890 |
| | 3 | 6.4249 | 18.6360 | 19.6750 | 29.4330 | 34.0202 | 63.8359 | 48.9330 | 52.7351 | 69.1109 | 69.6762 | 122.0679 | 327.8389 | 533.6571 |
| | 4 | 8.8689 | 19.3620 | 18.6629 | 32.7549 | 41.9669 | 34.7979 | 40.9260 | 60.6539 | 75.0360 | 81.9299 | 117.2159 | 255.7161 | 678.2701 |
| | 5 | 8.4698 | 12.8891 | 18.1570 | 25.2349 | 31.9009 | 37.8611 | 37.3862 | 99.7930 | 55.5470 | 63.0262 | 133.7030 | 248.6730 | 643.8742 |
| Average | | 7.4595 | 16.7164 | 21.6630 | 31.5906 | 34.8044 | 42.6122 | 42.9425 | 63.4864 | 69.5284 | 69.3531 | 123.9071 | 292.8812 | 644.7632 |

Figure 5 is the graph of Table 1. It is obvious from Figure 5 that the results demonstrate a strong linear trend. I have also included a trend line. The R-squared value for this line is 0.9938, so I can be confident in saying that time needed to push flow rules is a linear function. My results show an increase of 6.2922 milliseconds for each additional flow rule change, which even matches up well with the timings I observed for pushing one rule. This result may change depending on the hardware used in the network, current traffic in the network, and link bandwidth between the controller and switches. All of my results were measured without any extra traffic in my network. I did this to measure time needed to change flow rules only because I was not interested in how traffic in a network affected the bandwidth between a controller and switches.

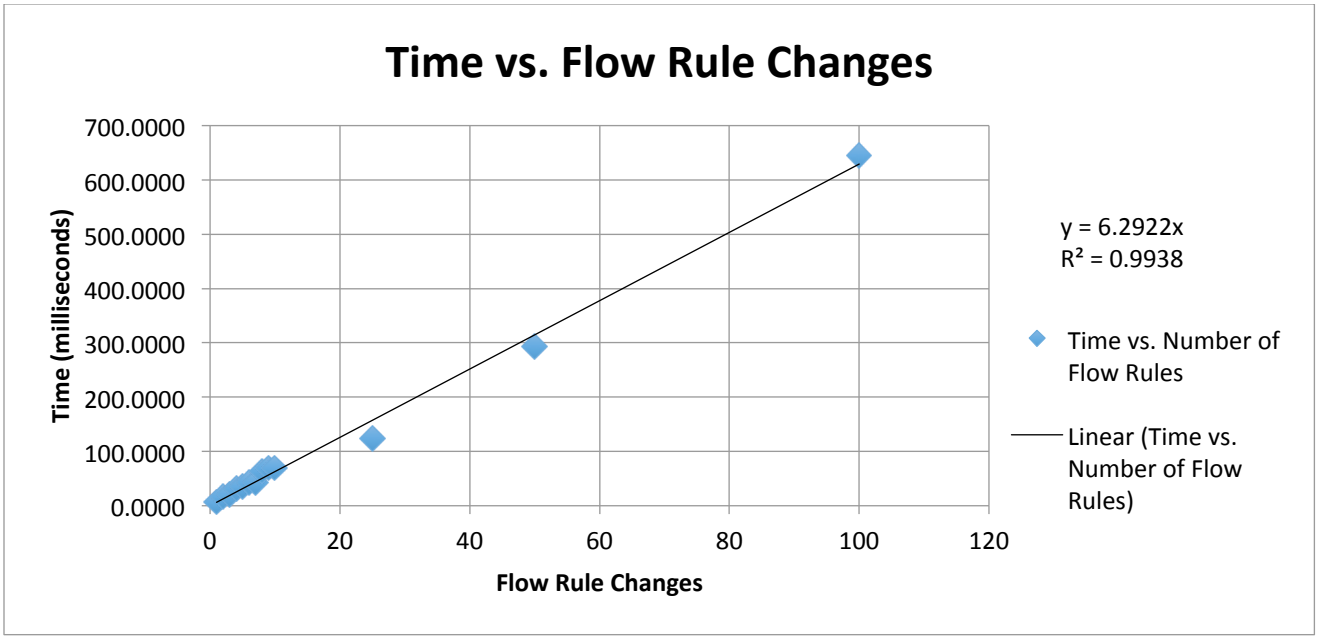


Figure 5. Timing Results for Pushing Different Amounts of Flow Rules

This result alone could save an abundant amount of time for a network engineer if a network has been misconfigured or needs to be initially configured. If a certain quality of service has been implemented network wide on all of a network's switches, there would be a lot of necessary reconfiguration if any part of that quality of service were misconfigured. If the network was a software-defined network, any part of the quality of service could quickly be changed if the process for a single switch takes less than one second. Observing this result can also give insight into how simple implementing a quality of service network wide would be. Instead of logging onto every switch and configuring the quality of service, the policies can be pushed remotely, thus using less time overall.

This result also shows that it is possible to entirely change a network wide quality of service relatively quickly. Of course the amount of flow rules that need to be changed is the largest factor in determining how quickly this can be done. Nevertheless, this result will be a

major piece to help determine whether or not a quality of service change will ultimately benefit a network or not.

The queues that have been designed on the switches are what keep the traffic at acceptable rates. While designing the quality of service, the maximum and minimum rates for the queues should be taken into consideration before switch configuration. It is possible to set the maximum and minimum rates of a specific queue to be the same number. In this case, the switch will try to maintain a rate of that certain traffic to fit the given rate no matter the load on the network. I have observed rates with amounts of variation above and below the given rate of a queue, which can be seen in Table 2. Of course, my testing was conducted on a specific switch, so rates and the rate tolerance may change depending on the types and brands of switches used within a network.

Table 2. Actual Rates of Flows Compared to Queue Rates

| | Upper Bound (Mbps) | 2 | 8 | 20 | 80 | 100 | 500 | 1000 |
|---|---|---|---|---|---|---|---|---|
| Result (all in Mbps) | | 1.91 | 7.64 | 19.10 | 74.20 | 91.40 | 413.00 | 744.00 |
| | | 1.92 | 7.65 | 25.10 | 89.20 | 108.00 | 431.00 | 762.00 |
| | | 3.13 | 10.10 | 19.10 | 75.20 | 91.50 | 414.00 | 751.00 |
| | | 2.88 | 9.86 | 25.10 | 90.50 | 109.00 | 432.00 | 770.00 |
| Average | | 2.46 | 8.81 | 22.10 | 82.28 | 99.98 | 422.50 | 756.75 |
| Range | | 1.22 | 2.46 | 6.00 | 16.30 | 17.60 | 19.00 | 26.00 |

The results seen in Table 2 are not what I expected. I expected the flow rates to remain fairly true to the set queue rate limits. However, I observed quite a bit of variation in the actual rates as a queue's set rate limit increased. This result can be observed in Figure 6 as a graph. The amount of variation increases greatly in queue rate limits up to 100 Mbps, but after that threshold the increase of variation declines. I expected to see a constant rate of variation for this result. I believe that it is still possible to work with this type of result, but this observation needs to be

taken into account. At a 1 Gbps rate limit on the queue, I observed rates fluctuating 26 Mbps. A queue rate limit of 26 Mbps provides for a 3.44% tolerance, while a queue rate limit of 2 Mbps results in 50% tolerance. I calculated tolerance as the ratio of range to average rate, and I expect tolerance to remain lower for higher queue rate limits.

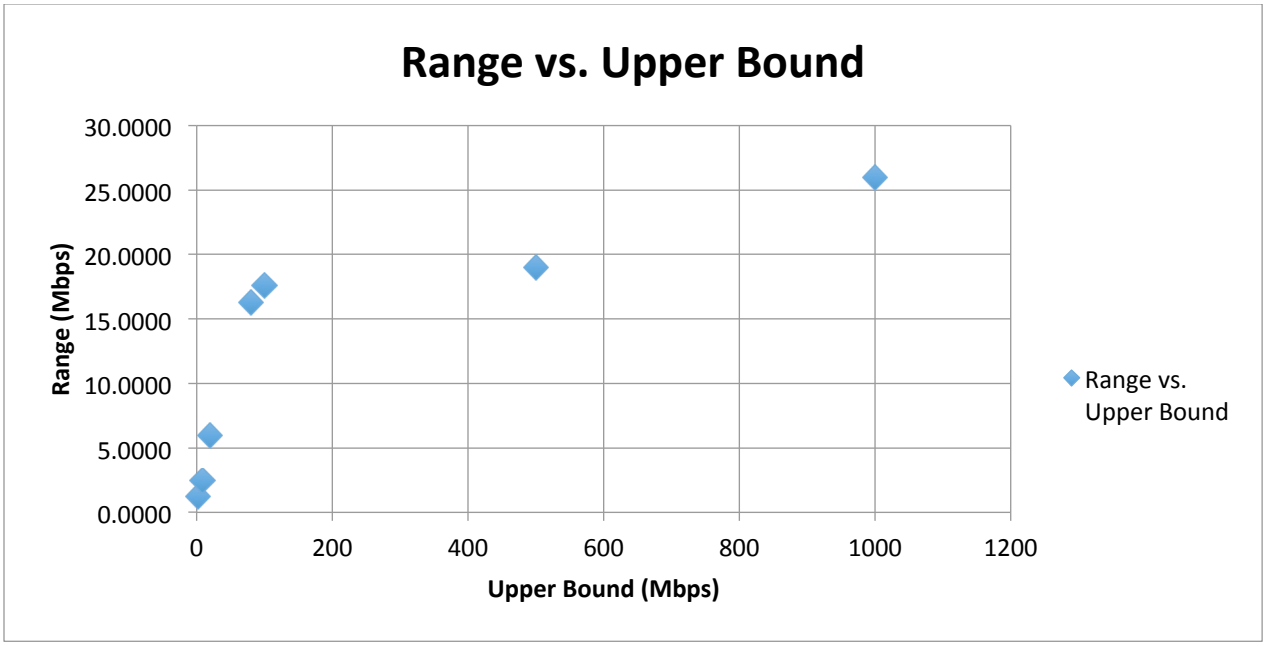


Figure 6. Flow Rate Range Compared to Rate Limit

Another interesting result, which can be seen in Table 2, is the average rate of the flows. I expected the average rate to remain closer to the set rate limit, but this was not the case. Figure 7 shows the average rate of a flow versus the queue's upper rate limit. This graph shows a strong linear trend. The cause of this may possibly be internal and require more research into why this is happening. Nevertheless, the linear trend may possibly help network engineers determine what value of the queue should be configured in order to achieve the desired rate.

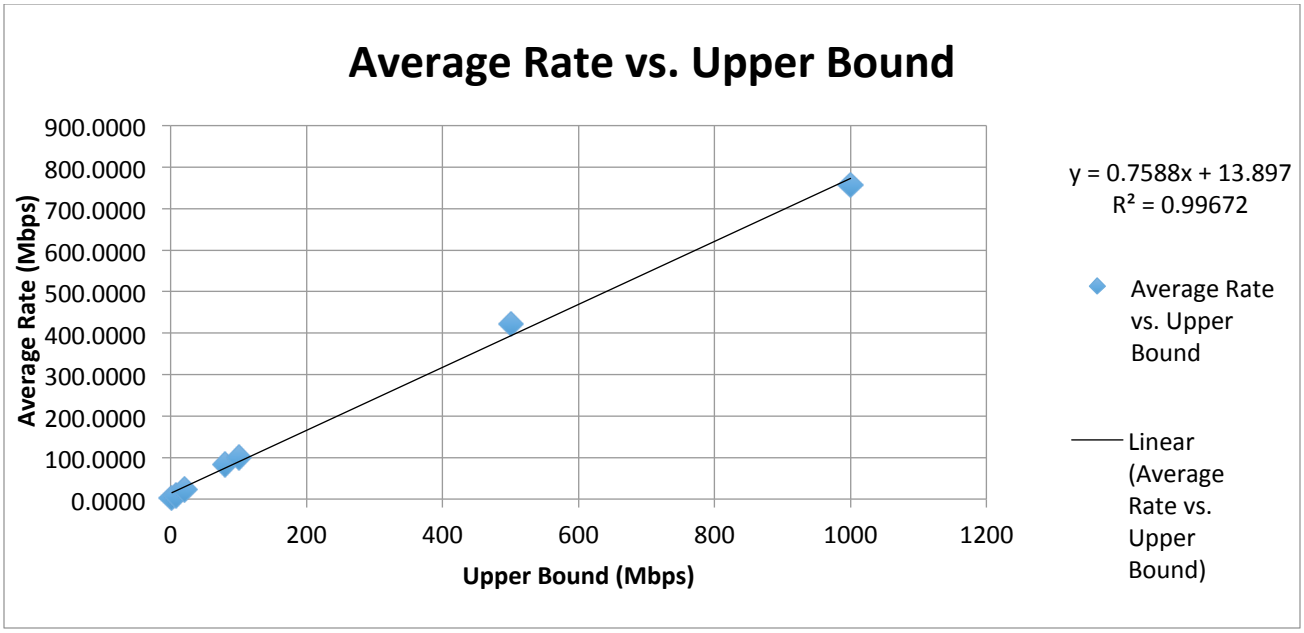


Figure 7. Average Flow Rate Compared to Queue Rate Limit.

Assigning two traffic flows to the same queue on a switch will ensure that both flows receive similar minimum and maximum bandwidth. Flooding a network with traffic without assigning flows to specific queues does not ensure this; the traffic is not even given equal priority in the network but, instead, is given preference depending on when the flow was established. Ensuring that traffic is given the correct priority requires a queue with the correct maximum and minimum rates allowed and a flow rule that directs traffic to the appropriate queues.

Another way that traffic is affected in and out of queues can be seen through the interaction of some flows being pushed to queues and other flows without queue assignments. The flows without queue assignments are negatively affected by all of the OpenFlow matches occurring. Since each packet has to be checked through the flow tables in an OpenFlow switch, the packets that are not placed into queues have access to significantly less bandwidth. Before enabling any sort of queues or quality of service on my virtualized network, I observed

bandwidth as high as 25.6 Gigabits per second. After the queues had flows pushed to them and any sort of quality of service was on the switch, the flows without a queue assignment dropped down to between 2.10 and 3.50 Megabits per second. The first number is not as important because that is the default bandwidth given to a switch in Mininet on my computer. The second number does bear some significance though. I believe that this number is tied to the queues that are being utilized. I did further testing with different networks, and I have reason to believe that the flows that are not specifically given queue assignments are still rate limited by a queue that the switch is using.

After noticing that the flows that were not supposed to be rate limited were experiencing lower bandwidth, I tested the network by installing the normal queues on the switches. However, instead of applying all of the quality of service rules to the network, I only applied one rule. During performance testing, I observed that pathways would exhibit bandwidth similar to what they would if they have quality of service rules on them. It seems as if the switches are queuing the packets even though they should not. If the appropriate rules were later installed on the network, the switches behaved how I expected them to by limited the bandwidth the appropriate amount.

# CHAPTER IV

# CONCLUSION

The most exciting result that I observed was the speed at which a quality of service rule could be exchanged. The longest time that I observed for a single rule was 8.8689 milliseconds, which was under my expectation of this taking around one second. With a time this low, a switch that is deployed into a network can populate its flow table in minimal time with the required rules for a quality of service. In traditional networks, a switch is normally configured outside of the network environment with the correct QoS policies before it is deployed into a network. With this information, a switch can first be deployed into a network and connected to a controller, which then informs the switch of what it should do with the incoming flows. The proper queues would still need to be configured on a switch before it is deployed; however, I believe that this can be done in less time than it takes a network engineer to initially configure quality of service on a switch in a traditional setting. The proper commands can be written in a script that configures the queues on a switch requiring minimal work from a network engineer.

My suggestion for queue configuration is to determine a few default rates and to think ahead to different possible QoS policies that might be changed or implemented in the network. The default rates could correspond to certain proportions of the available bandwidth like a traditional quality of service. Those queues could have multiple types of flows pushed to them. As I have shown in my testing, the queues will keep the rates of all flows pushed to it as close to the maximum allowed rate. The minimum rates put on the queues should work together as well. Whenever the network's bandwidth is fully used, the minimum rates should show the true form of how the network handles certain types of flows.

The other possible queues that might be useful to configure on a switch depends on how a network engineer thinks they will change the quality of service. There might be two completely different sets of queues used in two different qualities of service that can be interchanged. Another possibility is to predict how the policies might be changed to benefit from the preexisting quality of service with some minor tweaks. This would mean minimal extra queues on top of the default queues. However, I believe that there could be enough combinations with the said default queues to be able to effectively change the QoS policies to benefit the network. Increasing or decreasing the priorities on different types of traffic flows could benefit the network.

The way in which the flow rules are pushed to the switches can be done remotely, and I have shown that little time needed to perform this. All of the flow rules will also be accessible from a central location, namely the network controller. This is just another benefit of this type of environment. In a traditional network, a network engineer would log into each individual switch and configure the quality of service from there. However, in an SDN environment, the controller can be thought of as containing all of the configurations for the switches that it communicates with. The configuration can be given to the controller as the switch is deployed, and the switch's quality of service can come online almost as soon as it starts receiving packets. The time needed before a switch is ready includes first time flow rule pushing. The overhead that I mentioned in the results section plays a role here. As the rules required increases, the amount of time needed to push the rules from the controller to the switch increases linearly. However, I do not believe that this initial setup time for the switch will take any longer than it takes a network engineer to manually configure a quality of service on a switch for a traditional network. Being able to automate this process would also bring down the time needed on the network engineer's behalf.

The interaction of queues amongst each other should also be noted while configuring default queues and extra queues on the switches. My testing showed that multiple flows can be assigned to a single queue and maintain the appropriate bandwidth. This means that a network engineer should not create unnecessary queues. However, once the network is at maximum capacity, the flows that share a queue will fight for bandwidth. Two different flows assigned to two different queues with the same maximum and minimum rates will do their best to maintain the minimum bandwidth for each individual flow. These results should be taken into account while determining how many queues to initially configure on the switch.

Another interesting result that I found is the interaction between flows that are assigned to a queue with flows that are not assigned to a queue. As mentioned in my results, the flows that are not assigned to queues are not given much or any priority in the network. The results that I found were quite confusing and not always consistent. For this reason, my suggestion is to assign every flow to a queue. If the miscellaneous flows only need best effort service, then I suggest placing them in a queue with a low minimum bandwidth. Sorting every possible flow into a respective queue will keep the network running more predictably.

I believe that my hypothesis can be proven with some stipulations. The original hypothesis was to prove the feasibility of changing policies given a certain situation. I believe that this approach is feasible depending on the situation and how long the new policies will be in place. The variation in the situation can be great. It is possible for a network to be working under its maximum potential, which could lead to an overall time loss between queuing and dropping packets while there is still available bandwidth. In this type of situation, a policy change would be of great benefit. The alternative to this would be a situation in which the network is working under its full potential but is not dropping enough packets to warrant a policy change. In this

case, a policy change could benefit the flows with dropped packets, but it could also harm other flows more. Meaning that a policy change would be an overall time loss.

The duration that the new policies will be in place is also related to network benefits over that time period. If a new quality of service is pushed to every switch in the network, the overhead can add up quickly. Pushing possibly hundreds of new flow rules to different switches could take seconds to complete. By that time the network could be in a completely new situation that would require yet another quality of service. A better scenario is changing a quality of service or just a few rules then leaving them in place for long enough to experience the benefits. The amount of time that they would have to be left in place is a wide variable. The benefits will never be seen though unless the quality of service is a net benefit, meaning it benefits more flows than it hinders.

One use case for this information is to have a quality of service to be interchanged depending on predetermined situations. Some of those situations could be companywide updates or video conferences. In these cases, a network could benefit from changing priorities of those more important flows. A network engineer would also know the situation that the network will be in as well as how long the new quality of service will be in place. This known information can be the reason that policy changes are beneficial to the network.

One technology that I originally wanted to use in this research was the OpenFlow meter. Meters are a way of programmatically changing the rates of a queue as well as gathering the packet rate. This technology would have kept most of my testing in OpenFlow itself instead of relying on other technologies to measure bandwidth and timing. However, meters were not implemented into the switches that I used in my virtualized network at the time of this research. I believe that one future area of research for this topic would include incorporating OpenFlow

meters into a dynamic quality of service. This would allow the controller to contain all of the relevant information for the QoS and the network engineer would be able to retrieve that information from the controller.

Another future area of research on this topic would include automating the interchanging of policies. Either through simple scripting or even machine learning. As I have previously mentioned, there are a few considerations to take into account before changing policies. If the network engineer knows of certain recurring situations, a simple script might be the best option. However, changing out policies without supervision might require machine learning so the network can analyze what the best outcome would be depending on how the policies are changed.

# REFERENCES


[1] Ryan Wallner. 2012. floodlight-qos-beta. (December 2012). Retrieved September 16, 2017 from https://github.com/wallnerryan/floodlight-qos-beta.

[2] Mininet Team. 2009. Getting Started with Mininet. (December 2009). Retrieved September 16, 2017 from http://mininet.org/download/#option-1-mininet-vm-installation-easy-recommended.

[3] Ryan Wallner and Robert Cannistra. 2013. An SDN Approach: Quality of Service using Big Switch's Floodlight Open-source Controller. *Proceedings of the Asia-Pacific Advanced Network* 35, 14-19. DOI: http://dx.doi.org/10.7125/APAN.35.2

[4] Murat Karakus and Arjan Durresi. 2016. Quality of Service (QoS) in Software-defined Networking (SDN): A survey. *Journal of Network and Computer Applications* 80, 15 (February 2017), 200-218. DOI: https://doi.org/10.1016/j.jnca.2016.12.019

[5] Hilmi E. Egilmez, S. Tahsin Dane, K. Tolga Bagci, and A. Murat Tekalp. 2012. OpenQoS: An OpenFlow Controller Design for Multimedia Delivery with End-to-End Quality of Service over Software-Defined Networks. *Signal Information Processing Association Annual Summit and Conference*, 17 (January 2012), 1-8.

[6] Open Networking Foundation. 2012. OpenFlow Switch Specification. Retrieved from https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.1.pdf.

[7] William Stallings. 2013. Software-Defined Networks and OpenFlow. *The Internet Protocol Journal* 16, 1 (Mar. 2013), 2-14.