ABSTRACT

AN ADAPTIVE APPROACH TO OPTIMIZING QOS THROUGHPUTS IN SOFTWARE DEFINED NETWORKS

J. M. Boley, MS
Department of Computer Science
Northern Illinois University, 2017
Nicholas T. Karonis, Director

Maximizing bandwidth availability to critical science data flows is a key strategy in guaranteeing that current infrastructure can keep pace with the increasing demands of distributed science workflows across multiple facilities. Much of the research in this area has gone into developing parallel file transfer protocols such as GridFTP and leveraging circuit-based reservation systems like ESnets OSCARS. Software Defined Networks (SDN) is a recent development that presents a programmable approach to networking that moves routing logic out of the network fabric and abstracts the physical network behind a unified API. While SDN has gained significant traction in the data center, its potential is just beginning to garner attention in the scientific computing space and much investigation has yet to be done. In this study we examine an adaptive, priority-driven algorithm that leverages SDN capabilities to monitor flows, enforce per-application bandwidth guarantees and reallocate unutilized bandwidth between virtual circuits in real time.

Trials were conducted on a simplified, six-switch topology with a single bottleneck link to demonstrate the correct behavior of the algorithm and compare it to competing traffic management techniques. We show that by intelligently adjusting throughput limits–effectively

loaning out bandwidth from flows that are under-utilizing reserved bandwidth to flows that can make better use of the extra bandwidth–more optimal throughputs can be achieved for priority applications at a minimal cost to total bandwidth usage compared to such classic policing schemes as Differentiated Services queues.

NORTHERN ILLINOIS UNIVERSITY
DE KALB, ILLINOIS

MAY 2017

AN ADAPTIVE APPROACH TO OPTIMIZING QOS THROUGHPUTS IN SOFTWARE DEFINED NETWORKS

BY

J. M. BOLEY

A THESIS SUBMITTED TO THE GRADUATE SCHOOL

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE

MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

Thesis Director:
Nicholas T. Karonis

ACKNOWLEDGEMENTS

Special thanks goes to Drs. Rajkumar Kettimuthu of Argonne National Laboratory and Eun Sung Jung of Hongik University, South Korea, without whose patronage, guidance and contributions this work would never have left the ground. The author would also like to thank his Northern Illinois University faculty committee members, Drs. Nicholas Karonis, Michael Papka and Kirk Duffin, who graciously dedicated the time and energy to critiquing and guiding the author through both the writing process and the defense of this work. The author gratefully recognizes the resources and funding provided by Argonne National Laboratory, without which this work would not have been possible.

TABLE OF CONTENTS

Chapter Page

1 INTRODUCTION 1

2 BACKGROUND 5

2.1 Software-Defined Networking 5

2.2 Quality of Service (QoS) and OpenFlow 7

2.3 Related Work 10

3 DESIGN AND IMPLEMENTATION 12

3.1 Problem Definition 12

3.2 Algorithm Overview 14

3.3 Algorithm Design 17

3.4 Implementation 24

3.4.1 Reservation Subsystem 24

3.4.2 Network Monitor Subsystem 25

4 RESEARCH METHODS 27

4.1 Topology 27

4.2 Organization of Experiments 28

4.3 Traffic Generation Parameters 30

4.4 Analysis Strategy 33

5 EVALUATION 35

5.1 Testbed Architecture 35

5.1.1 Virtual Network Configuration 36

Chapter Page

5.1.2 Controller Configuration 39

5.1.3 Traffic Generation 40

5.1.4 Data Collection 42

5.2 Analysis 43

6 DISCUSSION AND CONCLUSIONS 50

6.1 Conclusions 53

REFERENCES 55

LIST OF TABLES

Table Page

4.1 Scenario 1 QoS, BE specifications for randomized and controlled waveforms . 30

4.2 Scenario 2 QoS, BE specifications for randomized and controlled waveforms . 31

4.3 Scenario 3 QoS, BE specifications for randomized and controlled waveforms . 31

4.4 Scenario 4 QoS, BE specifications for randomized and controlled waveforms . 32

5.1 DiffServ queue configurations. Scenarios 1, 2, 3 and 4 refer to 20%, 30%, 40% and 50% unreserved bandwidth scenarios, respectively. 39

LIST OF FIGURES

Figure Page

2.1 The canonical software-defined network archetype 6

2.2 The Differentiated Services Queuing approach 7

2.3 OpenFlow meter specification. 9

3.1 Comparison of hypothetical flow rates by traffic shaping method at four intervals 15

4.1 Simple six-switch topology with bottleneck link and flows directed from left to right 27

5.1 Topology configuration specifications 37

5.2 Topology configuration specifications (repeated). 42

5.3 Measured ingress against egress rates, overlaid with tracking/reservation for QoS1, QoS2 and BE Flows, from 40% BE scenario. 43

5.4 Cumulative ingress and egress rates against bottleneck capacity (rates in Kbps) 45

5.5 Performance gains for 20% Unreserved (BE) Scenario. 47

5.6 Performance gains for 50% Unreserved (BE) Scenario. 47

5.7 QoS and total average throughputs by traffic type and category 48

5.8 AQoS QoS gains and overall throughput losses against DiffServ queues by scenario 49

6.1 Behaviors of the OpenFlow meter implementation 51

CHAPTER 1
INTRODUCTION

Driven by the increasing sophistication and fidelity of simulation and science experiments, processing and storage of the massive volumes of data acquired and analyzed in science workflows has become highly distributed, typically spanning multiple facilities. Moreover, this deluge of science data continues to grow. As noted in [1], the sheer volume of data produced in scientific computing results in the need to support transfers of terabytes (TB) and even petabytes (PB) of data between facilities across the Wide Area Network (WAN). The obstacles to ensuring efficient and timely delivery of these massive volumes of data are many, but perhaps the single greatest roadblock is the fact that the data transfer capabilities of the intervening networks, specifically at the routing hubs, is not growing in proportion to the volumes of the data being routed [2].

Maximizing the overall efficiency of multiple end-to-end data transfers across the Wide Area Network poses significant challenges when faced with uncertain network conditions towards the endpoints. A considerable amount of work has been done in the scientific computing space, including the GridFTP protocol [1] and Globus' implementation of a GridFTP parallel file transfer system [3]. However, despite significant progress the nature of the infrastructure makes it extremely difficult to consistently maximize performance of individual data transfers over long-haul connections. Data can be buffered at the Data Transfer Nodes (DTN) and large files split and sent into the network as parallel streams, for instance, and paths and bandwidth slices can be provisioned with ESnet's On-Demand Secure Circuits and Advance Reservation System (OSCARS) [4], but once the packets leave the DTN they must travel through campus and regional networks before even reaching an OSCARS circuit in the

WAN. Even before packets leave the DTN, changing loads on the subnets behind the DTN and within the data storage facility may hamper availability of the data on the DTN. While it may be possible to optimize load balancing on local networks there is still no guarantee of optimal performance. Minimization of wasted wide area network resources therefore falls to the provisioning and management systems.

Distributed and parallel storage systems introduce a number of variables that impact their ability to service large data requests, including the number of simultaneous or overlapping requests (possibly background, or noise), the diameter of the smallest link in a Storage Area Network (SAN), queuing delays at the switches and the sizes of the files being transferred [5]. The work done on GridFTP's pipelining feature [6] underscores the detrimental impact on transfer rates introduced by lots of little files. These issues become even more glaring in the presence of 100 Gb-capable networks [5]. The observed throughputs over time of large data transfers never consistently utilize the total reserved link capacity. It is most often the case that flows do not use all of the available bandwidth most of the time. As a result significant amounts of bandwidth go wasted.

The majority of effort invested in solving these and similar problems has gone into working around the problems or dealing with them in software that runs on the endpoints; far less has been applied directly to the networking infrastructure and traffic management frameworks in this context. The recent advent of Software Defined Networking (SDN) [7] as a viable, accessible and industry-endorsed alternative to classic networking has the potential to induce a major shift in the scientific computing landscape. For the first time, network designers and developers have been given the tools to build multi-tiered network applications that begin in the user application space and drill down to the network fabric, where each layer operates on an abstraction of the one below it. Before SDN, engineers had been forced to design around the lack of flexibility in classic network infrastructure, and were further forced to patch around the inevitable collection of heterogeneous protocols brought by a heterogeneous network fabric comprising a patchwork quilt of closed, proprietary network devices [8]. In contrast, SDN has the potential to allow scientific computing frameworks to tap into the potential of the infrastructure on which they are built; instead of being passive players in how data is prioritized and transferred, applications could have a direct line to the network application framework.

The question of how Software Defined Networks can be leveraged to fulfill this promise is still very much open; the technology and specifications remain under ferocious development. The predominant SDN protocol to emerge in recent years is the OpenFlow protocol [9], but like everything else in the SDN space it too has undergone significant evolution. Looking back to classic traffic engineering approaches such as classic Differentiated Services (or DiffServ) Quality of Service (QoS) [10] offers us hints as to how network resources might be effectively managed, but these approaches are limited by their sheer complexity, the decentralized and heterogeneous nature of the technology on which they are built and the monumental task of coordinating the implementation of network-wide resource allocation changes in anything approaching real-time [8]. Software Defined Networks have the proven potential to make the unmanageable manageable by offering an abstract, unified view of the physical network to management applications. The challenge remains for the SDN and OpenFlow communities to find new and innovative ways to tackle traffic-engineering problems, and develop a new toolset that leverages the capabilities of a programmable network infrastructure. In answer to this challenge, our research investigates how Software Defined Networks can be leveraged to enable real-time optimization of network resource allocation. We focus on bandwidth allocation amongst multiple high-priority flows and how the problem of (per-flow) bandwidth under-utilization may be addressed by the OpenFlow protocol's QoS-like mechanisms. Our work is specifically targeted towards reservation-based, access-controlled network applications such as OSCARS [4], though the results of our work will certainly have much broader implications. Four key considerations drive our efforts:

1. Exploration of Quality of Service (QoS) in the SDN space, and its application to the scientific computing domain. Our research falls amongst other first steps efforts in the scientific community aimed towards investigating traffic engineering innovations made possible by SDN architectures.

2. Support for the growing impetus for expansion of SDN's application space, which until recently has largely coalesced around the data center. Our initial survey of available SDN technologies reveals a plethora of specialized SDN network components that (loosely) fall under the general SDN umbrella but are tightly integrated into existing, proprietary solutions that could best be described as hybrids. While research and development of SDN is thriving, its common application remains largely confined to a very narrow niche.

3. The increasing need for real-time analysis in several science domains that correspondingly increase the demands for both bandwidth and latency guarantees from traffic management frameworks.

4. The increasingly orchestrated nature of data movement between multiple facilities across the Wide Area Network in our science workflows, which also require greater flexibility and stricter QoS guarantees on a per-workflow basis.

We develop and evaluate a traffic control algorithm implemented on the open source Floodlight SDN controller platform. This study focuses specifically on demonstrating the benefits of using an OpenFlow traffic engineering solution in comparison with other notable approaches, both traditional and rooted in the SDN movement.

CHAPTER 2
BACKGROUND

2.1 Software-Defined Networking

Software-defined networking is an approach to network application design that separates the control logic from the implementation of the data plane, or the physical devices and operating systems running on those devices. The SDN paradigm separates network architecture into three layers: a control layer, network virtualization layer, and a network operating system (NOS) layer [9]. At the highest level is the control program, which operates on a high-level abstraction of the global network and is responsible for driving network policy. Below that is the virtualization layer, where the physical characteristics of the global network are reduced to a simplified representation against which the control program can issue commands. At the lowest level, the Network OS controls the physical devices that comprise the network fabric. In broad terms, the virtualization layer translates the control program's high-level directives into low-level dialogue understood by the NOS.

The foundational principles and concepts underlying the SDN concept are not new, and solutions incorporating the core principles have existed in one form or another since the 1990's. Some of the earliest examples can be found in work done with ATM switches [11]. Others include Devolved Control of ATM Networks (DCAN), Common Open Policy Service (COPS), Simple Network Management Protocol (SNMP), and Forwarding and Control Element Separation (ForCES) [8]; the drive for SDN-like networks has continued unabated and numerous examples and alternative approaches have appeared over the years [7]. The most

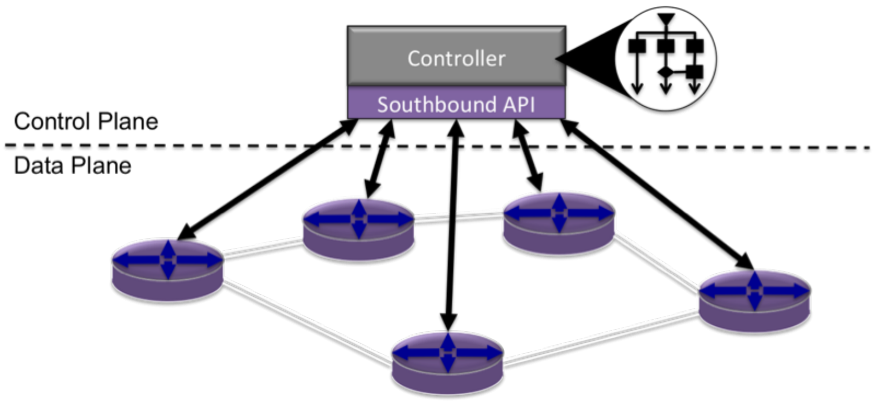


Figure 2.1: The canonical software-defined network archetype

recent SDN movement began with the inception of the OpenFlow protocol [12], which was introduced in the 2008 newsletter OpenFlow: enabling innovation in campus networks [9]. The OpenFlow protocol bridges the controller software and the network operating system devices with a completely open specification under a GPL-type license. Because of its flexibility and transparency, the OpenFlow protocol has become widely recognized as a de facto standard and driving force in the networking community's adoption of SDN technology [4], eventually gaining traction with major companies such as Cisco and Juniper, and creating a new market for start-ups like Corsa Technologies and BigSwitch Networks.

Perhaps the most recognizable incarnation of the SDN paradigm, driven by the notion of a homogenized network fabric where all devices speak to a controller through a unified Southbound API, is illustrated in Figure 2.1. In nearly all non-proprietary SDN deployments known to the research team, the OpenFlow protocol provides the common language through which the controller communicates with network devices and vice versa. Removing the routing logic from the switch has the profound effect of making the network fabric programmable, in real time. More importantly, traffic management strategies and policies are no longer hampered by a distributed networking intelligence that may or may not be in full possession of the information needed to make smart traffic management decisions.

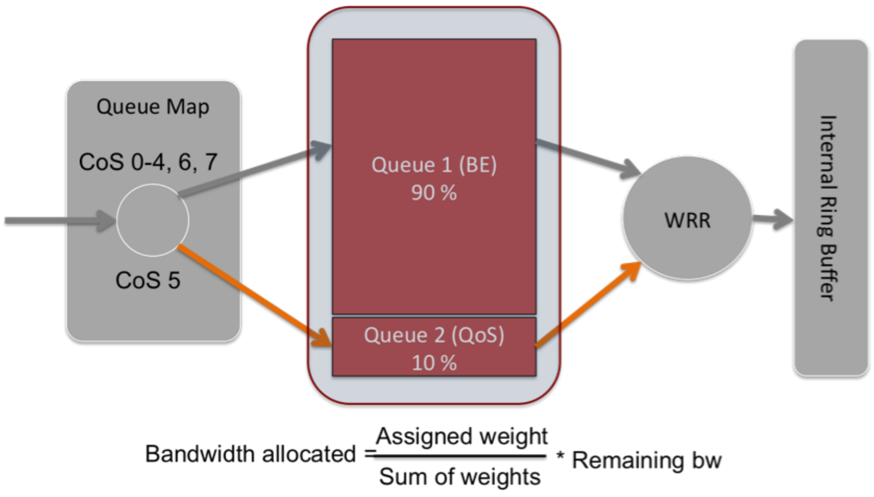


Figure 2.2: The Differentiated Services Queuing approach

2.2 Quality of Service (QoS) and OpenFlow

Classic traffic engineering approaches include Differentiated Services (or DiffServ), originally proposed in [13]. As detailed in [10], traffic flows are assigned a priority (each packet carries a DSCP code that maps to a Class of Service) and sent to a queuing mechanism on the edge router at which it enters the network. The DiffServ queuing strategy typically employs a token-bucket algorithm in which overflow packets are dropped and a weighted round-robin scheduler that controls the subsequent transmission of packets. Both are implemented on the switch and so the queuing mechanism enjoys near-instantaneous reaction times to changes in flow rates crossing the switch.

The ratio of traffic belonging to different classes of service is specified by a system of weights, as illustrated by Figure 2.2. A reduction in the total bandwidth utilized by flows belonging to a class of service leads to redistribution of the unused bandwidth following the indicated weighted ratio scheme. The weights in the figure are expressed as percentages. This has the serendipitous effect of creating a minimum throughput guarantee; the aggregate traffic sent to any queue will get at least the proportion of bandwidth that the round-robin ratio guarantees, and flows in any queue are allowed to expand into any bandwidth not being used by the aggregate of any other flows directed to any other queue. In practice, most hardware switches do not implement more than four queues per port, effectively precluding fine-grained, per-flow rate policies as switches in real networks can have hundreds or thousands of flows exiting their ports at any given time.

An unfortunate limitation of DiffServ implementations is their relative inflexibility and the difficulty associated with managing a network of DiffServ switches. The Simple Network Management Protocol (SNMP) [14] was adopted to address these and similar issues [8]. In the SDN space, the OpenFlow standard does not include any protocol for updating queue weights, and so they too must be adjusted on a switch-by-switch basis, meaning that individually touching all involved ingress switches is an absolute necessity when traffic rates need to be adjusted. Additionally, most switches (whether virtual or hardware) only allow a maximum of four queues per port. In the presence of a busy network, which could be host to hundreds or thousands of flows, this severely reduces the viability of using a strictly DiffServ approach to fine-grained management of flow behavior; the DiffServ model was never designed to allow traffic management on a per-flow basis, but rather takes the position that traffic engineering can be accomplished most effectively operating on class-based aggregates of flows.

From the outset, the OpenFlow specification [12] has included support for legacy QoS queues, and most production switches, both virtual and physical, still implement them. Any OpenFlow enabled controller can issue an instruction that sends matched incoming packets to a specific queue on the relevant output port. However, setting or changing queue ratios is not supported by the OpenFlow protocol; from the controller's perspective ratios are fixed, requiring jury-rigged fixes outside of the controller and OpenFlow specification to allow controller-driven adjustments. This was the only traffic management tool available to the OpenFlow community until the OpenFlow 1.3 [15] specification was released.

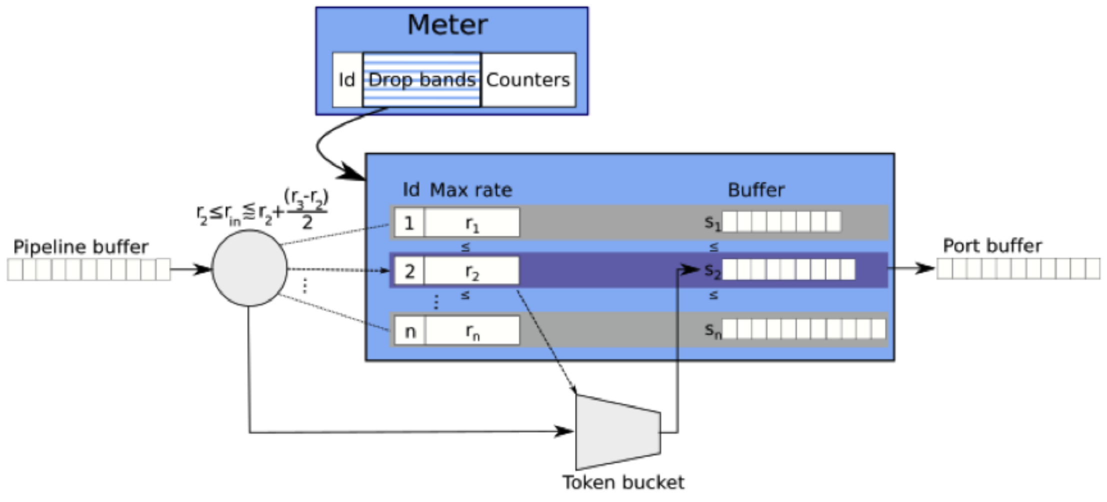


Figure 2.3: OpenFlow meter specification

The v1.3 draft of the OpenFlow specification [15] introduced a new traffic management tool, dubbed a meter by the protocol designers. Despite their somewhat misleading label, OpenFlow meters do more than track byte counts; they police flows by enforcing a rate cap, or absolute upper rate limit. A schematic representation of the protocol's specification of the meter is shown in Figure 2.3. Multiple drop bands provide several rate caps, and the cap closest to the collective rates of the flows sent through is selected as the active rate limit. All meter implementations encountered by the research team have done away with the notion of multiple bands, effectively enforcing only a single rate cape irrespective of flow rates.

Thanks to the flexibility of the OpenFlow protocol, both individual and aggregates of flows can be directed to any meter. It is therefore possible to carve out channels of link bandwidth for any number of flows in OSCARS-like fashion; if flows attempt to grow outside of their so-called channel the meter mechanism will selectively drop packets until flow rates more closely match set rate limits. Most notably, unlike classic DiffServ queues and other static schemes, meter rates can be programmed by the controller in real-time, making them ideal for much more dynamic traffic management policies.

An approach to traffic engineering that arises naturally from the OpenFlow 1.3 specification is the use of so-called meters (with drop packet bands) that impose an upper limit on a flow's rate [15]. This approach is appealing because the OpenFlow specification specifically prescribes the use of meters as per-flow traffic control mechanisms, which is a natural fit for any fine-grained traffic management strategy. However, as this study will show, using a purely static approach (i.e., one not driven by some sort of application logic) to metering flows imposes penalties to bandwidth utilization that can be as harmful to overall bandwidth utilization as simply doing nothing.

2.3 Related Work

The drive to find answers to the bandwidth-tuning problem is not new. Other works, many sponsored by the US Department of Energy, have sought to develop traffic management frameworks that accomplish one kind of traffic optimization or another. Examples include work done for the UltraScience Net project [16] described in [17], which relies on intelligent bandwidth scheduling and signaling daemons that coordinate activities of network components, and the GARA framework [18][19] which also targets adaptation of the network in real-time. These frameworks differ from the present work in that they are built on classical (i.e., non-SDN) network approaches and infrastructure and, in the case of the UltraScience Net project, a different approach to making bandwidth guarantees.

Utilizing a traditional DiffServ approach for SDN networks was explored in [20] and implemented as a plugin to the Floodlight controller [21]. Most OpenFlow-enabled switches–in fact, all of the hardware and software switches known to the research team–offer standard DiffServ queues. The work detailed in the article implemented a QoS application at the control layer, and while it offered no innovations that demonstrated any clear advantage to

SDN-based traffic engineering over legacy networks, it more than adequately demonstrated how 1) QoS concepts could be integrated into the SDN space; and 2) the controller's northbound API could be extended to allow management applications to coordinate QoS services.

The OpenFlow 1.3 (and above) specifications recommend combining DiffServ queues and meters in a sort of hybrid approach for fine-grained rate policy implementation. This approach is briefly touched on in [22], where the authors advance the idea of leveraging SDN and OpenFlow capabilities in the virtualization of satellite network provisioning. This study also examines the performance of a simple strategy combining static metering and queueing.

The DANCES framework [23] also identifies the OpenFlow 1.3 meter as an ideal traffic control mechanism. Like the UltraScience Net project, DANCES also relies on the notion of bandwidth scheduling. OpenFlow meters are used to enforce static classes of flows, where flows are policed in aggregate as in classic DiffServ queuing. The focus of research efforts with the DANCES framework is to maximize the aggregate throughputs of each class of traffic, not the individual bandwidth needs of the applications. In a similar vein, the Google B4 [24] and Microsoft SWAN [25] projects focus on maximizing total throughputs across the network but ignore the needs of individual, mission-critical applications.

CHAPTER 3
DESIGN AND IMPLEMENTATION

3.1 Problem Definition

Science workflows that require coordinated activities at multiple endpoints (sites) are the target application of this work. These workflows require significant transmission of science data between sites and, significantly, the storage facilities behind the Data Transfer Nodes (DTN). Real-time simulation workflows may also require streaming between HPC or supercomputer clusters. There is no way to predict how many such workflows may be moving data in parallel, so it must be assumed that at any given time it is possible that more traffic is being sent than the networks can handle given current practices. Whether this is strictly true all the time at present is less important than the fact that, as the demands on infrastructure increase as the generation or collection of science data becomes ever more prodigious, the total infrastructure supporting long-haul connections will be in an increasingly poor position to shoulder the burden.

It is further assumed that these science workflows require solid Quality of Service (QoS) guarantees, particularly in regards to bandwidth availability, and will accordingly suffer if demands are not met. Real-time analyses that consume data collected remotely are obvious candidates for QoS guarantees. However, even science applications that simply want to move data from Site A to Site B may entail the movement of data of sufficient volume that QoS guarantees are desirable. Parallel workflows requiring different degrees of guarantees is inevitable. This provides an opportunity to examine more nuanced schemes of QoS guarantees than a bipolar classification system that treats all traffic as completely critical or completely trivial. Several QoS classes are proposed, ranging from QoS1 through QoS4 in decreasing order of precedence. Application traffic that is not critical to a science workflow, such as email, streaming video and video conference feeds, is considered secondary to the mission of the science network. This kind of traffic—along with general Internet traffic—is considered background but classified as Best Effort in recognition of the fact that some minimum guarantee is still necessary.

Science application (or QoS) flows are considered on a per-application basis, while Best Effort traffic is managed in aggregate. However, for the sake of simplicity each science workflow in these experiments is assumed to have a single flow and each flow is given a separate QoS guarantee. A single flow also stands in for Best Effort traffic. In a real-world deployment, QoS guarantees would be made on the basis of the aggregates of the flows belonging to individual applications. The OpenFlow protocol renders the implementation differences fairly trivial.

The OSCARS model is followed in this work, where each application submits a request for a specific slice of bandwidth and the framework creates a dedicated virtual circuit and enforces the bandwidth agreement. These agreements are referred to as reservations, and generally follow the notion of a Service Level Agreement (SLA). In this sense, the framework evaluated in these experiments makes per-application SLAs.

$$F = (s, sp, d, dp, proto, AT, ST, D, class, minBW) \tag{3.1}$$

Equation 3.1 formally defines a flow, where s is the source IP, sp is the source port, d is the destination IP, dp is the destination port and $proto$ is the network protocol, which uniquely identifies the flow. AT is the expected arrival time of a flow, ST is the actual start time, and D is the duration or expected lifetime of the reservation. Finally, $class$ is the QoS class requested for the flow and $minBW$ is the minimum bandwidth required by the flow.

User applications may not have a clear notion of how much bandwidth is required for a data transfer, which leads to two possible scenarios: 1) The application overestimates its requirements, leading to bandwidth that would go unutilized in a static circuit allocation system like OSCARS; or 2) The application underestimates its requirements, leading to strangulation of the data transfer. One is assumed as likely as the other, and the AQoS framework takes this into account. The specific goals of the AQoS algorithm are to lend bandwidth to needy applications while not allowing other applications to tie up unused bandwidth in gratuitous reservations. At the same time, the AQoS algorithm must also honor minimum bandwidth guarantees to applications that can utilize them.

3.2 Algorithm Overview

In simple terms, the AQoS algorithm tackles the bandwidth-sharing problem by leveraging the following techniques:

1. **Redistribution of unused bandwidth amongst priority flows.** Similar to traditional QoS schemes, we treat flows as having given priorities on a scale from highest (QoS1) to lowest (Best Effort, or background). Priority flows that are consistently underperforming have the unused portion of their bandwidth reservations shared out amongst those other priority flows that can make use of it.

2. **Priority-driven bandwidth assignment.** Unlike traditional traffic shaping with DiffServ Queues, which proportionally allow all flows (regardless of priority) to expand into unused bandwidth based upon a fixed ratio, we selectively allow flows to expand into available bandwidth. The highest priority flows are allowed to expand into unused

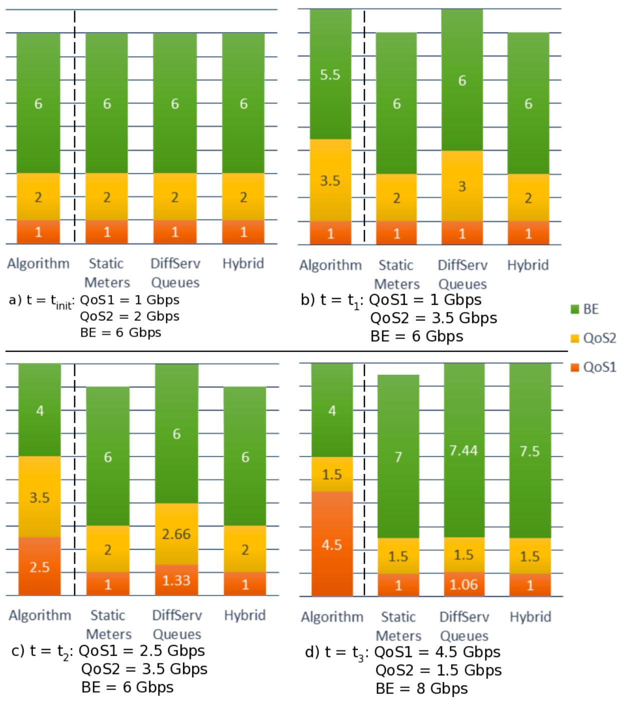


Figure 3.1: Comparison of hypothetical flow rates by traffic shaping method at four intervals

bandwidth first, after which any remaining unused bandwidth is passed down to lower priority flows. Any leftover bandwidth is then handed over to background traffic after flows at all priority levels have been serviced.

QoS flows are treated as first-class citizens. If a QoS flow is able to expand it is generally allowed to do so, unless any link on its path is so heavily loaded that there is no bandwidth left to give it. This comes at the expense of the Best Effort flows, which in our implementation are used as a catchall category for unimportant background traffic; Best Effort flows have no priority and are considered secondary to QoS flows. In cases of extreme overload where available Best Effort bandwidth is exhausted, the Adaptive Quality of Service (AQoS) algorithm is designed to take bandwidth from lower-priority flows that are over their reservations and give it back to an under-performing higher-priority QoS flow if it is growing back into its reserved bandwidth.

To get a sense of how the algorithm should work, consider a scenario with three flows, two QoS and one Best Effort, on a simple two-node topology. Flows QoS1 and QoS2 have initially reserved 1 and 2 Gbps on a 10 Gbps link, respectively, while BE traffic is free to utilize the remaining 7 Gbps, of which it is able to make use of 6 Gbps for a total utilization of 9 Gbps (Figure 3.1.a). Alternately, assume 1) a static metering scheme; 2) a traditional DiffServ Queues scheme utilizing 3 queues with weights 1:2:7; and 3) a hybrid scheme with static meters for QoS1 and QoS2 and two DiffServ Queues with weights 3:7, with QoS1 and QoS2 going to the first and Best Effort sent to the second. If QoS2 was then able to expand to 3.5 Gbps, in our scheme it would be allowed to do so, while QoS1 remained at 1 Gbps and, because it has no priority, Best Effort traffic would be throttled back to 5.5 Gbps, as in Figure 3.1.b. This is a marked contrast with the static meters-only approach, where QoS2 would not be allowed to expand at all, QoS1 and Best Effort remain at their rates and 1 Gbps of bandwidth goes to waste. As a result of the minimum guarantees made by the Queues-only approach, Best Effort would remain at 6 Gbps at the expense of QoS2, which is only allowed to expand to 3 Gbps. The hybrid approach fares no better than simple metering in this case.

Now suppose that QoS1 could similarly expand to 2.5 Gbps. In our scheme Best Effort traffic would be scaled back by an additional 1.5 Gbps while QoS2 remained at 3.5 Gbps (Figure 3.1.c). If a static metering scheme were used instead, nothing would change; QoS2 and BE would remain steady while QoS1 would not be able to expand into the unused 1 Gbps of bandwidth. The queues-only scheme would penalize both QoS1 and QoS2 (at approximately 1.33 Gbps and 2.66 Gbps, respectively) while Best Effort remained steady at 6 Gbps. The hybrid approach would severely penalize the QoS flows; again, nothing would change as the static meters would not allow the flows to expand beyond their reservations and 1 Gbps of bandwidth would go wasted.

Finally, if QoS2's potential utilization drops to 1.5 Gbps while both QoS1 and Best Effort flows were capable of utilizing the extra 2 Gbps, in our scheme the extra bandwidth would be assigned to the priority flow, QoS1 (Figure 3.1.d). In a meters-only approach, QoS2's throughput would drop 0.5 Gbps below its reservation, QoS1 would remain at 1 Gbps, and Best Effort would pick up an extra 1 Gbps with an associated 0.5 Gbps of bandwidth wasted. A queues-only approach would redistribute QoS2's unused bandwidth between both QoS1 and Best Effort, yielding rates at approximately 1.06 Gbps for QoS1 and 7.44 Gbps for Best Effort. A hybrid approach would penalize QoS1 with its rate cap while allowing Best Effort to fully expand into the unused 0.5 Gbps.

Based on the above discussion, we expect to be able to show that a traffic engineering approach based on real-time monitoring and adaptation of bandwidth allocation will increase overall network utilization for priority traffic and improve the performance of client applications that are able to exploit an increase in bandwidth share relative to alternative strategies.

3.3 Algorithm Design

The AQoS algorithm controls bandwidth allocation to QoS flows on a per-flow basis. Byte count and duration samples for every registered flow are collected from flow ingress switches concurrently and used to derive rate statistics, which in turn drive the AQoS algorithm's logic. A timer configured during initialization controls the frequency with which both flow states are sampled and the algorithm operates on the data that has been gathered up to that point. Flow data is organized, converted into usable rate statistics and compiled into histories on a per-flow basis. All samples gathered regarding known flows at any given time constitute a snapshot, or an image of network state at the time of sample gathering. Over the course of a single flow management cycle, the rate data is analyzed and rudimentary behavioral statistics (such as the direction a flow's throughput is trending over the past several cycles) is computed and compiled with previous statistics.

The operation of the algorithm can be divided into four high-level phases, or steps: Characterization of the flows; seizure of bandwidth from underperforming flows; return of seized bandwidth to needy QoS flows; and loaning out surplus bandwidth to greedy QoS flows (i.e., flows that can make use of the extra bandwidth).

1. **Characterization.** During this step, a QoS flow's rate history is inspected to gather insight into its current behavior. This step is primarily concerned with identifying whether the flow is in a relative steady state or has begun (or is continuing) to exhibit an increasing or decreasing trend. The flow is marked appropriately. The algorithm keeps the last 10 rate samples gathered from the network. At this time, as a trade-off between the algorithm's ability to react quickly and accurately, the past four samples are actively used to extract short-term trends that predict the next few seconds with reasonable accuracy.

2. **Bandwidth share-out.** QoS flows from which downward trends have been extracted are inspected. Those that have previously taken on loaned bandwidth return whatever portion they are no longer using. Flows that have gone below their reserved rates similarly have any unused, allocated bandwidth added to a pool of bandwidth available for loaning out in subsequent steps. For each flow that goes under its reservation, a small slice of the unused bandwidth (currently 5%) is kept in reserve; if the flow should begin to expand back into the bandwidth originally assigned, the algorithm will be able to detect it and react accordingly.

3. **Bandwidth return.** QoS flows having throughputs beneath their reserved rates are evaluated to determine if they have begun to expand back into the bandwidth assigned

to their reservations. If this is the case then the algorithm first attempts to reallocate unassigned bandwidth from collected bandwidth pools on a flow's path. If that is not sufficient then the algorithm will begin taking bandwidth back from other QoS flows, from least to highest priority. Since Best Effort flows do not have priority, bandwidth is simply taken from them while the QoS flows are free to continue using any extra bandwidth they have acquired. This behavior holds except under extreme circumstances where the network is so heavily loaded that Best Effort flows are effectively choked out.

4. **Bandwidth assignment.** If any free bandwidth remains on their paths, QoS flows that are at or near their reserved rates are tested to see if they will expand into it, from highest- to lowest-priority. As Best Effort flows have no priority, their collective bandwidth is fair game for the QoS flows—that is, if a QoS flow is able to expand it is allowed to do so at the expense of Best Effort traffic. Once all QoS flows have been serviced, any remaining bandwidth is handed over to the Best Effort flows.

At the heart of the algorithm is the network utilization graph, a specialized data structure that the AQoS algorithm consults for the most current network usage data. Data structures track port egress capacities, the outgoing bandwidth reserved for QoS flows and bandwidth not utilized by QoS flows, per switch. These data structures are embedded in unidirectional edges that connect the nodes representing individual network devices. Each node is connected by two edges representing full duplex links between switches; one edge for each direction that flows may travel across the link.

An Open Shortest Path First (OSPF) algorithm searches through the graph to find a route consisting of the fewest possible hops over links with sufficient remaining bandwidth during the reservation creation process. Links with insufficient remaining capacity are automatically excluded from the resulting virtual circuit. A reservation subsystem persists flow path information, which the AQoS algorithm uses to trace a flow's path and gather relevant information during decision-making. As the AQoS algorithm generates rate data for the flows, the network utilization graph is continually updated to reflect real network load as new data becomes available.

The AQoS algorithm's control subroutines are broken up into three primary components, following the major steps outlined above, that execute in series during a flow management cycle. Each is responsible for handling a different portion of analysis and control and is largely dependent on the results generated by previous processing, if any—for example, seizing bandwidth from under-performing is the first activity performed and so has no dependencies, while returning bandwidth to needy flows is partially dependent on bandwidth collected in the first step.

Every flow has an associated tracking variable, which can loosely be thought of as modeling a flow's observed rates, though it is not directly associated with any specific rate measurement. This is a critical point to understand the design of the AQoS algorithm; it models how expected flow behavior, derived from observed behavior, changes over time. It therefore has a correspondingly tremendous impact on decision-making. The tracking variables serves two primary functions: When a flow is over its reservation and increasing, the tracking variable acts as a tentative upper limit for the possible rates that the algorithm expects to see and directly corresponds to the meter rate set on the flow's ingress switch. When it is shrinking below its reservation, the tracking variable more tightly follows the measured rates and is used to determine how much of its reserved bandwidth can be handed out to other flows. Most importantly, tracking variable values carry over from previous management cycles and so form the basis for the algorithm's next round of decision-making. It persists the most critical pieces of information from the algorithm's previous decisions. The compiled flow history provides the context.

Algorithm 1 SEIZE-BW(networkGraph, flowDB)

```
 1: for all $flow \mathbf{in} flowDB$ do
 2:     $lowerbound \leftarrow flow.tracking -$ $CALC\text{-}WINDOW(flow.rate, flow.history)$
 3:     if $flow.rate < lowerBound$ then
 4:         $decrease \leftarrow$ $CALC\text{-}OPTIMAL\text{-}DECREASE(flow.tracking, flow.rate, flow.history)$
 5:         $flow.tracking \leftarrow flow.tracking - decrease$
 6:         $flow.flagged \leftarrow DECREASING$
 7:         if $flow.rate < flow.reservedBw$ and $flow.history.last.rate \geq flow.reservedBw$ then
 8:             $reserved \leftarrow GROWTH_BUFF$
 9:         else
10:             $reserved \leftarrow 0$
11:         end if
12:         $ADD\text{-}TO\text{-}AVAILABLE\text{-}POOLS(networkGraph, flow.route, decrease - reserved)$
13:     end if
14: end for
```

The first component is responsible for identifying flows that are either under or are trending significantly downwards from their allotted bandwidth when above their reservations. Each flow is examined in order of precedence. Algorithm 1 below outlines the logic in pseudocode. Ideally, a lower bound is calculated based on a model of the flow's waveform trend and variance that could trigger bandwidth seizure, based on a measure of confidence, as a means of correctly distinguishing shrinkage from noise in the ingress rates. In practice this is not a simple exercise; several variations were tried and all failed to consistently isolate actual flow shrinkage cases from simple noise. Addressing this problem remains an open research question. In the meantime, the *CALC-WINDOW* subroutine is a stub that reduces to 0, so that the lower bound calculated is actually the value of the tracking variable. In preliminary trials this proved an adequate substitute, though the algorithm does occasionally take away bandwidth only to hand it back within the next few cycles. Optimizing this behavior would require substantial investment in developing statistical methods that can consistently and correctly differentiate between noise in the data and gradual downward trends.

If the measured rate falls below the lower bound then the flow is eligible for bandwidth seizure and a decrease amount is calculated and subtracted from the tracking rate and added

Algorithm 2 RETURN-BW(networkGraph, flowDB)

```
for all $flow$ in $flowDB$ with $flow$ not $flow.flagged = DECREASING$ do
    $lowerbound \leftarrow flow.reserved - CALC\text{-}WINDOW(flow.rate, flow.history)$
    if $flow.tracking < flow.reserved$ and $FIND\text{-}INCREASE(flow.rate, flow.history) = TRUE$ or $flow.rate \geq lowerBound$ then
        $available \leftarrow FIND\text{-}MAX\text{-}AVAILABLE\text{-}ON\text{-}PATH(networkGraph, flow.route)$
        $optimal \leftarrow CALC\text{-}OPTIMAL\text{-}INCREASE(flow.tracking, flow.rate, flow.history)$
        $returned \leftarrow MIN(available, optimal)$
        $flow.tracking \leftarrow flow.tracking + returned$
        $REMOVE\text{-}FROM\text{-}AVAILABLE\text{-}POOLS(networkGraph, flow.route, returned)$
        $remaining \leftarrow optimal - returned$
        if $remaining > 0$ then
            $other \leftarrow GET\text{-}LOWEST\text{-}PRIORITY(flowDB, flow)$
            while $remaining > 0$ do
                $returned \leftarrow returned + TAKE\text{-}BW(networkGraph, flowDB, other, remaining)$
                $flow.tracking \leftarrow flow.tracking + returned$
                $remaining \leftarrow remaining - returned$
                $other \leftarrow GET\text{-}NEXT\text{-}LOWEST(flowDB, other, flow)$
            end while
        end if
        $flow.flagged \leftarrow GROWING$
    end if
end for
```

to the pool of available bandwidth along the flow's circuit. If the measured rate has fallen below the reserved rate then a growth margin, or bandwidth buffer, is withheld and put in reserve to allow the algorithm room to grow back into its reservation. The bandwidth decrease calculation uses the maximum of either the difference between the tracking and mean rate over the past few cycles or a bare minimum decrease in an attempt to keep seizures conservative; the algorithm relies on the fact that continuing decreases will be caught in subsequent cycles. The flow is also tagged as decreasing to disqualify it from consideration by following components.

The second component, outlined by Algorithm 2, is responsible for identifying flows that have shrunk below their reservations but begun to grow back into them. Now that all flows eligible for bandwidth seizure have been identified, tagged and unused bandwidth added to

Algorithm 3 REDISTRIBUTE-BW(networkGraph, flowDB)

```
1: for all $flow$ in $flowDB$ from $flow.priority = HIGHEST$ to $flow.priority = LOWEST$ where $flow$ not $flow.flag = NULL$ do
2:     $lowerbound \leftarrow flow.reserved -$ $CALC\text{-}NOMINAL\text{-}WINDOW(flow.rate, flow.history)/2$
3:     if $flow.tracking \geq lowerbound$ then
4:         $available \leftarrow FIND\text{-}PATH\text{-}UNRESERVED(networkGraph, flow.route)$
5:         $borrowed \leftarrow MIN(available, optimal)$
6:         $flow.tracking \leftarrow flow.tracking + borrowed$
7:         $REMOVE\text{-}FROM\text{-}PATH(borrowed, networkGraph, flow.route)$
8:     end if
9: end for
```

the bandwidth pools along their paths (i.e, on the network graph), any flow expanding back towards its reserved rate may borrow from any remaining unused bandwidth along its path—with one exception. Only the minimum available along the path may be borrowed (more would exceed the available bandwidth at the bottleneck), so if there is a link on the path that has no available bandwidth then the flow still will not be able to utilize other unused bandwidth pools on its path. Flows are examined from high to low priority, excluding any that have already been tagged as shrinking.

In the case that a flow cannot expand into unused bandwidth, the RETURN-BW component examines other QoS flows, from low to high priority, and will start taking bandwidth back from the lowest QoS flow currently above its reserved rate. If that is not enough then the next lowest is found, bandwidth is reclaimed, and so on. The flow is then flagged as growing back into its reserved rate and subsequently ignored by the bandwidth loaning component.

The final component is outlined in Algorithm 3. It manages the distribution of available bandwidth to QoS flows that appear to be growing beyond the bandwidth reserved for them. The algorithm consults each flow's tracking variable to compare expected behavior to a rate-increase threshold.

3.4 Implementation

The algorithm is built on BigSwitch Network's open source Floodlight controller [21], a 3rd-party controller framework that implements the OpenFlow 1.3 specification, a stateful representation of the network topology, basic routing intelligence (following the so-called learning switch paradigm) and little else—which makes it ideal as there are no extraneous features to interfere with the algorithm. The algorithm belongs to a code module that is dynamically loaded into the Floodlight runtime at initialization, placing it as close to the controller's southbound API as possible. The module integrates with the controller through an internal API that exposes both controller functionality and provides a programmatic abstraction of the OpenFlow protocol using the OpenFlowJ-Loxigen library [26].

3.4.1 Reservation Subsystem

The AQoS algorithm is built on two significant subsystems. The first is the reservation subsystem, which is responsible for admitting new flows to the network, via the OSPF-like topology search function discussed previously, updating reservation data on the network graph as new QoS flows are admitted or expired, and pushing reservation (virtual) circuits to the network. When a client reserves bandwidth across the network, the reservation system attempts to find a path across the network that can meet the client's bandwidth request. If a path meeting the client's criteria is not found the reservation is rejected. The reservation system attempts to load balance across the network by assigning new flows to any least-loaded paths it can find. Note that while no protocol allowing a client to negotiate a reservation with the controller has been implemented, the code implementing admissions decisions is fully in place.

In a realistic setting, reservations may need to be made at multiple levels of the TCP/IP protocol stack. The reservation system currently supports L2 (vlan) and L3 (IP) circuits; each reservation type's implementation also encapsulates the required details of creating virtual circuits on the network fabric in a generalized fashion. The current implementation only allows for preemptive circuit creation, meaning that circuits are created before flows are admitted. This is more than adequate for the purposes of this experiment; a more flexible strategy would allow a reservation to made in advance of the actual flow, and not push the circuit to the network until traffic arrives on an in ingress node. This becomes rather complicated when the necessary scheduling mechanisms that allow for real-time optimal bandwidth allocations are introduced, and so is avoided for now.

3.4.2 Network Monitor Subsystem

The network state monitor subsystem is responsible for gathering network load information from the fabric. This is handled through the mechanisms built into the OpenFlow specification, which allows the controller to poll individual switches for byte/packet count and duration data on a per-flow basis. Each switch is required to maintain this state information for each flow matched on its flow tables. In addition, a switch may be polled for the same statistics for each meter band, which tracks the bytes, packets that have been sent to the meter band in addition to duration (how long the meter has existed).

To accommodate a large number of flows, the network monitor starts a new thread of execution for each flow (reservation) that is registered, in which runs the code required to specify a statistics request message, have it written out to the ingress switch in question, and process the switch's response message. The extracted (raw) data is then used to calculate rate information. In this way, incoming flow rates are captured from ingress nodes on the network before packets are sent to the meter; these flow rate samples are then cached for later access by the algorithm. Because tracking a flow's past behavior is highly relevant to determining its current behavior, each flow has an associated history that is compiled and stored by the network monitor component.

At the lowest level, the network monitoring component runs a timer which is used to coordinate polling requests. When the timer signals the network monitor component that a given interval of time (configured during initialization) has expired, it in turns signals the polling threads. Thus, flow statistics requests are sent out and received by the switches (barring latency in the control network) nearly simultaneously, capturing a complete snapshot of the total load across the entire network on a node-by-node, per-flow basis.

CHAPTER 4

RESEARCH METHODS

Our evaluation of the algorithm addresses two primary questions: 1) Does it increase overall efficiency of the data transfers when it is possible for such flows to take advantage of bandwidth sharing; and 2) Is there a case for using it in place of alternative traffic engineering approaches such as DiffServ queues, standard (OpenFlow) metering or a hybrid approach?

4.1 Topology

A simple six-switch topology (shown in Figure 4.1 below) is used to establish the correctness of the algorithm in a straightforward, uncomplicated scenario as a baseline, and to benchmark our algorithm against the DiffServ queues, static meters and queue-meter hybrid approaches. Five traffic source hosts are linked to five traffic sink hosts on the far end of the topology. Traffic coming in on the ingress switches merges on the near bottleneck link

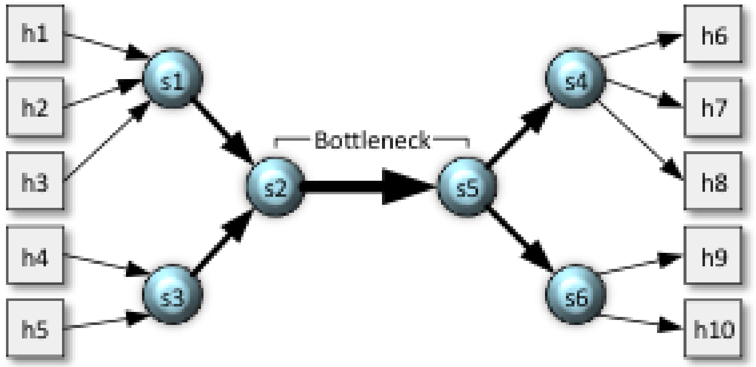


Figure 4.1: Simple six-switch topology with bottleneck link and flows directed from left to right

switch and then forks off to the appropriate egress switch on the far side. This configuration was specifically designed to demonstrate the AQoS algorithm's capability to identify a common bottleneck on the network and coordinate flow management mechanisms across multiple ingress nodes.

Comparison data for the alternative approaches is gathered on the egress nodes. During trials using competing approaches, the algorithm is disabled and the appropriate rate-shaping mechanism is set by the controller (static meters) or configured by hand on the switches (DiffServ queues).

4.2 Organization of Experiments

Data concerning a single QoS method is gathered during a trial. A single experiment consists of four trials, one for each competing approach: AQoS, Static Metering, DiffServ Queues, and Hybrid. To recap, the Static Metering method uses OpenFlow meters to carve out static bandwidth slices. The DiffServ Queues method uses the queuing strategy described in 2.2. The Hybrid method combines the Static Metering and DiffServ Queues strategies so that QoS flows are "metered" and all traffic is sent to one of several queues, with Best Effort receiving a dedicated queue.

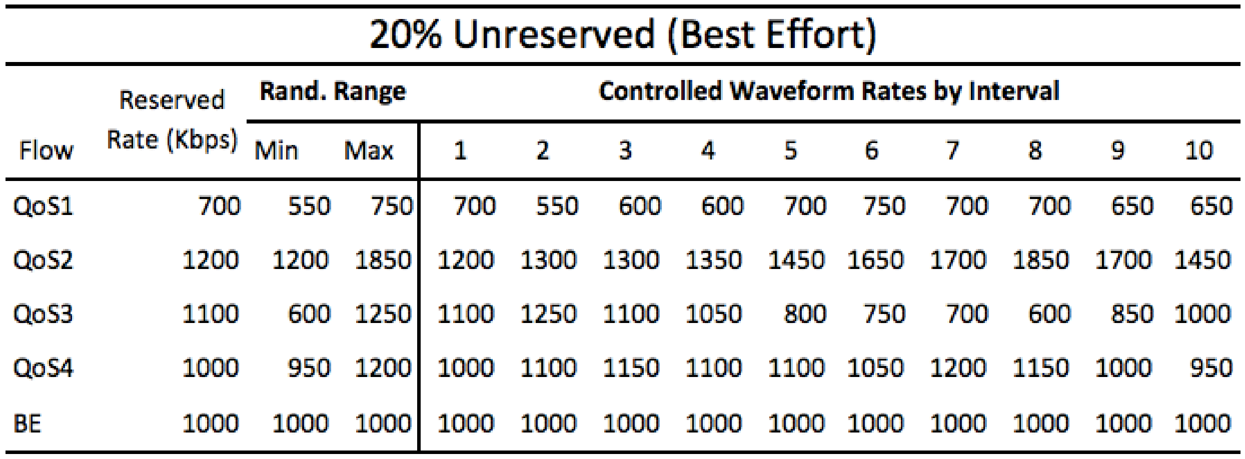
Experiments are grouped into complementary sets of two which differ in traffic generation strategy. One experiment uses traffic generated at specific waveforms at specific times. The second uses randomized traffic that is generated at rates within ranges that directly correspond to the counterpart experiment rates. Both traffic generation strategies are designed to guarantee congestion at all times on the bottleneck. These are covered in more detail in the following section.

At the highest level, experiments are organized into four major scenarios of interest, differing in the percentage of the bottleneck link capacity left unreserved (i.e, given over to Best Effort flows). Bottleneck link capacity is split 20/80, 30/70, 40/60 and 50/50, with slices given to unreserved (Best Effort) and QoS, respectively: In the 20/80 scenario, 80% of link capacity is set aside for QoS flows and 20% left unreserved; in the 30/70 battery, QoS flows are reserved 70% of link capacity and 30% is left to Best Effort, and so on.

One objective while conducting background research regarding the design of the experiment was to find descriptions of QoS queuing policies in use in real-world systems. Unfortunately, there seems to be little to find in the way of queuing profile specifications. The motivation for evaluating several BE/QoS traffic proportion configurations was born from this lack of background information. In any case, different networking applications are bound to have different needs depending on a multitude of factors, including size and capacity of the network, the average number of client applications loading the network with traffic at once and the size and characteristics of the flows themselves—to name only a few. There are therefore likely to be a wide range of real-world configuration examples; the chosen configurations should adequately cover the gamut of those likely encountered in real-world networks.

A final argument for having this specific range of configurations concerns the fact that the performance of the AQoS algorithm is expected to increase with increasingly greater proportions of Best Effort traffic. The AQoS algorithm, as illustrated in the above conceptual mockup, is designed to take bandwidth from lower-priority flows to optimize higher-priority flow performance. Higher BE-to-QoS ratios are expected to show increasing gains in favor of the QoS flows over competing QoS approaches. The point at which those gains become significant is critical to any evaluation of the AQoS algorithm's overall performance—and ultimately how useful it would (or would not) be in a real network.

Table 4.1: Scenario 1 QoS, BE specifications for randomized and controlled waveforms



20% Unreserved (Best Effort)

Flow	Reserved Rate (Kbps)	Rand. Range		Controlled Waveform Rates by Interval									
		Min	Max	1	2	3	4	5	6	7	8	9	10
QoS1	700	550	750	700	550	600	600	700	750	700	700	650	650
QoS2	1200	1200	1850	1200	1300	1300	1350	1450	1650	1700	1850	1700	1450
QoS3	1100	600	1250	1100	1250	1100	1050	800	750	700	600	850	1000
QoS4	1000	950	1200	1000	1100	1150	1100	1100	1050	1200	1150	1000	950
BE	1000	1000	1000	1000	1000	1000	1000	1000	1000	1000	1000	1000	1000

4.3 Traffic Generation Parameters

Traffic generation follows two distinct schemes, which we characterize as synthetic and randomized. Synthetic flows (also sometimes referred to as designer flows) are tightly controlled; that is, traffic generation follows specified waveforms, with incoming traffic having rates which change at controlled intervals. In contrast, randomized flows are not constrained to prescribed waveforms, though they still fall within given intervals. These two schemes provide opportunities to both examine the behavior of each approach under very specific conditions—that is, targeted QoS-to-BE load ratios—as well as more chaotic conditions that are intended to be representative of real-world networks under significant load.

Flow specifications for each scenario are given in Tables 4.1 through 4.4. All synthetic flows follow the specification for their QoS priority. Intervals are set to 10 seconds, and after each interval elapses the rate generation for a QoS flow increases or decreases to the next specified. The specifications for synthetic flow waveforms are given under the Controlled Waveform columns, and are used across trials with the AQoS, static meters, DiffServ queues and hybrid approaches. Due to their relative stability synthetic rate trials use only 10

Table 4.2: Scenario 2 QoS, BE specifications for randomized and controlled waveforms

30% Unreserved (Best Effort)

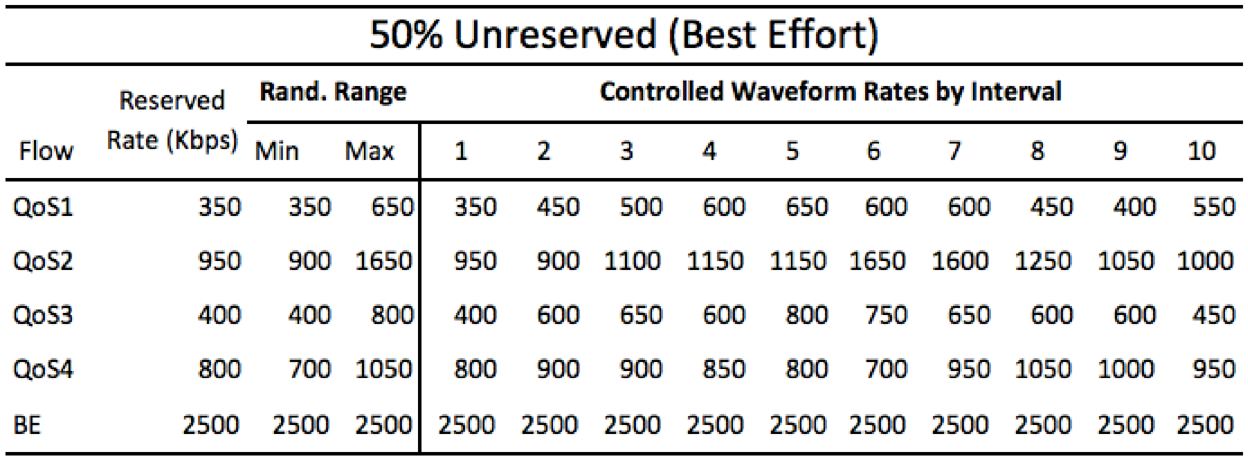
	Reserved	Rand. Range		Controlled Waveform Rates by Interval									
Flow	Rate (Kbps)	Min	Max	1	2	3	4	5	6	7	8	9	10
QoS1	500	300	900	500	600	725	900	700	550	300	450	400	550
QoS2	1200	1000	1300	1100	1300	1300	1200	1150	1250	1200	1100	1000	1300
QoS3	800	700	1100	900	1100	1100	900	800	750	900	1000	900	700
QoS4	1000	900	1350	1000	900	900	1100	1050	1350	1250	1050	1000	950
BE	1500	1500	1500	1500	1500	1500	1500	1500	1500	1500	1500	1500	1500

Table 4.3: Scenario 3 QoS, BE specifications for randomized and controlled waveforms

40% Unreserved (Best Effort)

	Reserved	Rand. Range		Controlled Waveform Rates by Interval									
Flow	Rate (Kbps)	Min	Max	1	2	3	4	5	6	7	8	9	10
QoS1	450	300	650	450	450	550	600	650	550	300	450	400	550
QoS2	1000	1000	1300	1000	1250	1300	1300	1150	1050	1200	1100	1000	1050
QoS3	600	600	900	600	600	650	600	800	750	800	900	900	700
QoS4	950	850	1350	950	900	900	850	1000	1350	1250	1050	1000	950
BE	2000	2000	2000	2000	2000	2000	2000	2000	2000	2000	2000	2000	2000

Table 4.4: Scenario 4 QoS, BE specifications for randomized and controlled waveforms



50% Unreserved (Best Effort)

Flow	Reserved Rate (Kbps)	Rand. Range		Controlled Waveform Rates by Interval									
		Min	Max	1	2	3	4	5	6	7	8	9	10
QoS1	350	350	650	350	450	500	600	650	600	600	450	400	550
QoS2	950	900	1650	950	900	1100	1150	1150	1650	1600	1250	1050	1000
QoS3	400	400	800	400	600	650	600	800	750	650	600	600	450
QoS4	800	700	1050	800	900	900	850	800	700	950	1050	1000	950
BE	2500	2500	2500	2500	2500	2500	2500	2500	2500	2500	2500	2500	2500

intervals, and flow data is collected from the egress switches at a rate of 1 sample per second for a total of 200 samples per flow per trial.

The controlled waveforms have been specifically tailored to keep the bottleneck link reasonably congested, while still bearing a reasonable semblance to real traffic; the AQoS algorithm's performance is also expected to improve as congestion increases, up to the point at which Best Effort bandwidth is exhausted and the algorithm must try to scavenge bandwidth from lower-priority QoS flows if it can. Keeping the link congested as a result of QoS traffic gives the AQoS algorithm the most opportunities to distinguish itself from competing approaches. However, care has been taken not to allow the bottleneck link to become oversaturated with QoS traffic, and so there is always sufficient remaining bandwidth to accommodate Best Effort traffic.

A randomized flow for a given priority is generated at a rate based on the ranges given in the Randomized (Rand) Range columns of Tables 4.1 through 4.4. These limits are designed to constrain QoS flow to ranges similar to the minimum and maximums of the corresponding synthetic flows. Synthetic flow ranges were chosen for the randomization trials to keep random flow results relatively consistent with synthetic flow results; in combination, the ranges do a reasonable job of guaranteeing the bottleneck link is congested as average QoS rates exceed the reservations.

Like the synthetic flows, a randomized flow holds at a given rate for a 10 second interval before the traffic generator randomly picks another throughput value. Due to the random nature of the resulting data each trial lasts for 30 intervals, and with a sampling rate of once per second, a grand total of 600 samples is collected per flow per trial.

4.4 Analysis Strategy

The data collected from the trials is first analyzed on a qualitative basis. The cumulative (total of all flows) ingress and egress traffic rates are calculated for each trial and then compared to illustrate each method's traffic-shaping characteristics. A per-flow ingress, egress rate comparison is then drawn to illustrate how each method handles incoming traffic in excess of a QoS flow's reserved rate; profiles are also constructed for Best Effort to illustrate the differences between each method's treatment of the lowest-priority flow.

The increase (or decrease) in the observed throughputs obtained from the AQoS algorithm against competing approaches, expressed as percentage gains, are then evaluated over the lifetimes of the individual QoS flows. This comparison is only valid for experiments that rely on designed traffic patterns (the so-called static flow trials), as all flows including Best Effort must be nearly identical at all times across all trials belonging to a scenario.

Next the average throughputs for each flow are computed, which are then used to calculate overall percent gain of throughput between the algorithm and competing approaches. Gain is calculated on both per-flow and aggregate bases for each scenario (e.g., 20% unreserved, 30% unreserved, etc.), once for synthetic flows and again for the randomized flows. Gains in terms of total overall throughput of all flows are also calculated. The QoS gains are expected to illustrate how performance improves with increasing Best Effort loads and to provide the quantitative groundwork for determining the point at which the AQoS algorithm begins to show significant QoS flow performance benefits.

CHAPTER 5
EVALUATION

5.1 Testbed Architecture

The simulation architecture used to evaluate the AQoS algorithm is composed of a single controller, multiple virtual switches and traffic generators on simulated hosts:

- Controller: Floodlight
- Switches: Open vSwitch 2.5.1 running on Ubuntu Server 16.04 (physical host)
- Traffic generation: *iperf2* processes running in UDP mode chained in sequence to simulate variable rate flows
- Network monitor: Floodlight monitoring component that queries switches for flow statistics

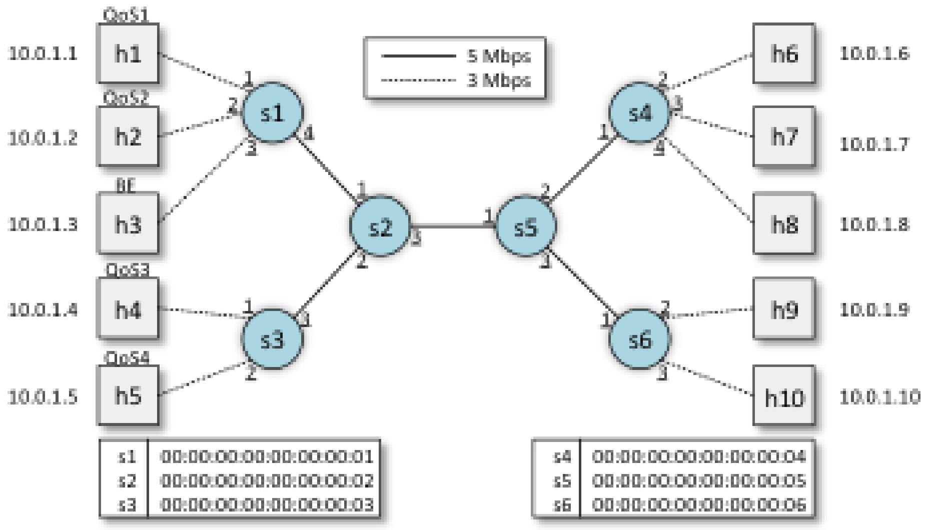
The virtual network—including virtual hosts—is built on a single physical machine. The virtual network switches connect to a Floodlight controller running on a separate physical host via a physical network. No traffic between virtual hosts generated on the virtual network leaves the virtual network, and so the physical network is dedicated entirely to network control traffic. The virtual network host is a single CPU system with 8 physical cores running at 3.2 GHz, scalable to 4 GHz under load, for a tally of 16 logical cores running at approximately 2.0 GHz under load. The virtual network host additionally boasts 16 GBs of RAM in addition to 16 GBs of swap space allocated on a solid-state drive with a theoretical read/write bandwidth of (approximately) 4 to 6 Gbps. The controller host is somewhat less powerful, with 4 cores running at 2.2 GHz.

5.1.1 Virtual Network Configuration

The 2.5.1 stable branch of the Open vSwitch software switch project [27] provides the switch instances on the virtual network. The standard release of Open vSwitch does not include support for OpenFlow 1.3 meters; an experimental code patch, proposed by an Open vSwitch developer, provides the missing functionality. The code patch provides solely a user space datapath implementation, requiring that switch instances must be run in user space mode. This had a direct impact on the design of the experiment; kernel space switch instances are capable of switching at Gbps rates, while user space instances are constrained to Mbps ranges. Additionally, because of its experimental status the meter implementation is relatively crude and lacking optimization, and this was expected to have an impact on the quality of the data gathered during the AQoS, static meters and hybrid trials.

Switch instances are created and configured with OpenFlow datapath IDs, which are required for the controller to track and communicate with them. The IPv4 address and listening port of the controller are also configured per switch. Once the configuration work is done the switches automatically begin sending handshake packets to the set controller address.

Inter-switch links are implemented with virtual Ethernet (or veth) pairs, which implement kernel space data pipes between a pair of virtual interfaces that can be bound to by host processes. Each endpoint is attached to a switch instance, forming a direct path for network traffic to traverse between switch instances. Veth endpoints are TAP interfaces, meaning that they provide L2 data frame encapsulation for IP packets. Virtual links created from



s1	00:00:00:00:00:00:00:01
s2	00:00:00:00:00:00:00:02
s3	00:00:00:00:00:00:00:03

s4	00:00:00:00:00:00:00:04
s5	00:00:00:00:00:00:00:05
s6	00:00:00:00:00:00:00:06

Figure 5.1: Topology configuration specifications

veth pairs function in the same manner as physical Ethernet cables; kernel space buffers implement a full duplex data pipe that connects the link endpoints.

The core network topology is built using this method. Referring to Figure 5.1, ingress switches *s1* and *s3* are connected to *s2*. The bottleneck link is then created between *s2* and *s5*. The egress switches *s4* and *s6* are then connected to *s5*.

The implementation of a virtual host is fairly straightforward, though not obvious. Traffic between processes, even when bound to veth endpoints, defaults to the host's loopback. This generally useful feature has the unfortunate side effect of complicating network emulation, as any traffic sent from one host process to another will always short-circuit and bypass the virtual network. Fortunately Linux provides a natural extension to the kernel's interface and routing table implementation that allows a user to isolate both physical and virtual interfaces within a network namespace. Traffic sent between processes running in separate network namespaces is forced to use the interfaces embedded in those namespaces; as each network namespace implements its own loopback, there is no way for traffic to get from one process to another over the default namespace loopback. Each network namespace also receives its own routing table and iptables (firewall), so each can literally be configured to behave like a physically distinct host. Processes, such as ping, netstat and iperf, can be executed from within a network namespace via the *ip netns exec* command. Note that executing a process in a network namespace requires root privileges.

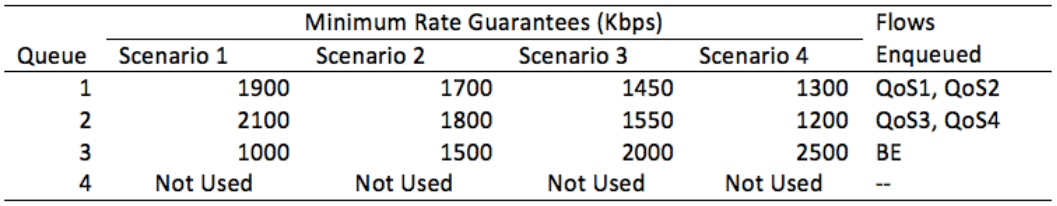
Veth pairs can also be used to create links between processes in different namespaces. They are in fact the only mechanism that Linux provides out-of-the-box that can do so. Referring again to Figure 5.1, virtual hosts *h1* through *h3* are created in this fashion and connected to virtual switch *s1*, hosts *h4* and *h5* are configured and connected to switch *s3*, hosts *h6* through *h8* are configured and connected to *s4*, and hosts *h9* and *h10* are configured and connected to *s6*.

Host NIC bandwidths are set (via the AQoS network topology graph) to 3 Mbps and inter-switch links are configured at 5 Mbps. These numbers are a consequence of the virtual network creation method; kernel space to user space translation and buffering of the traffic (recall that Open vSwitch must run in user space mode) imposes considerable constraints on the amount of traffic that can be generated per host. Though the cause was never identified, it was established early on that flows bound for the ingress switches become increasingly unstable as they cross network namespace boundaries as they grow in volume; this phenomenon never manifested when running Open vSwitch in kernel space mode.

Two of the competing approaches used to benchmark the AQoS algorithm's performance also require the use of DiffServ queues. Open vSwitch provides queuing that uses Linux Queuing Disciplines (or QDisc) to implement rate shaping. Queues are configured to use the HTB QDisc algorithm, which provides a policing mechanism most similar to traditional DiffServ queues. A maximum rate cap across all queues is specified in addition to the minimum rates enforced (under load) by the queues.

Three queues with separate minimum rate configurations are used in each scenario. Table 5.1 details the queue configurations for each and which flows are directed to them. Note that

Table 5.1: DiffServ queue configurations. Scenarios 1, 2, 3 and 4 refer to 20%, 30%, 40% and 50% unreserved bandwidth scenarios, respectively



	Minimum Rate Guarantees (Kbps)				Flows
Queue	Scenario 1	Scenario 2	Scenario 3	Scenario 4	Enqueued
1	1900	1700	1450	1300	QoS1, QoS2
2	2100	1800	1550	1200	QoS3, QoS4
3	1000	1500	2000	2500	BE
4	Not Used	Not Used	Not Used	Not Used	--

in the queues-only trials, multiple flows directed to a single queue will create congestion on the parent switch with the associated packet loss expected when flows compete for bandwidth. To keep the configuration as simple as possible, and due to the fact that no congestion is allowed by design on the ingress switches themselves, the QoS queues are configured on the near bottleneck endpoint switch *s2*, port 3.

5.1.2 Controller Configuration

The controller's network topology graph must be configured to match the virtual network topology. Due to time constraints, the management of this graph must currently be done by hand, though a planned extension to the AQoS module will eventually allow so-called management plane software to remotely configure the AQoS topology graph via the controller's REST API. An internal graph management API currently allows users to easily configure a topology, though a recompile of the module is required after any change. The topology graph is also built to the specifications given in Figure 5.1. By design the AQoS algorithm does all policing on the ingress nodes. Referring once again to Figure 5.1, using the configured network topology graph the algorithm will create and manipulate the rate caps of meters located on switches *s1* and *s3* to preemptively minimize (and ideally eliminate) congestion on switch *s2*.

Reservations must likewise be configured within AQoS module code. Another planned extension will allow client applications to negotiate a reservation, but both the protocol and the REST interface must be designed and implemented. Reservations therefore must also be configured by hand, which unfortunately limits the potential to test the AQoS algorithm with truly dynamic, more realistic networking scenarios where hosts requiring reservations across the network are not necessarily known at design time. An internal API makes configuring reservations relatively painless. The controller does need to know about node-specific flow configurations—queuing flows for DiffServ policing is a prime example—which tends to make configuring TCP/UDP circuits a bit more complex.

5.1.3 Traffic Generation

Experiments rely on UDP traffic that is transmitted from each sender host at specific bandwidths at specific times via the *iperf2* IP trace client. The UDP protocol is used instead of TCP since the *iperf2* UDP client allows transmission bandwidth to be specified. Other IP trace clients such as nuttcp [28]—which does allow trace clients to send TCP traffic at specified bandwidths—were examined but discarded due to negative impacts on the performance of the virtual network.

The *iperf-series* shell script encapsulates logic that chains together a series of *iperf2* processes to generate flows with controlled waveforms. The target virtual host and *iperf* receiver process address are provided to the script. Additionally, the desired interval—or duration of the individual *iperf* processes in the chain—and a directive indicating whether the script should generate traffic conforming to a specified waveform or simply randomize send rates is provided. If traffic rates are not randomized the intended waveform is given as a series of transmission rates, one for each interval. If traffic rates are randomized, upper and lower bounds for the transmission rates as well as the lifetime of the flow are provided instead. The *iperf2* client streams traffic at a constant rate during each interval in both cases. In practice, there are always fluctuations in *iperf2* send rates, which is a useful trait given that real flows on real networks typically will not converge to perfectly steady throughputs; this lends a small degree of authenticity to the traffic patterns to be managed by the AQoS algorithm.

Referring back to Figure 5.1, *iperf-series* jobs are initiated in virtual hosts *h1* through *h5*. Iperf2 servers running in virtual hosts *h6* through *h10* sink the generated flows. Synthetic flow traffic is generated according to the waveform specifications given in Tables 4.1 through 4.4 of §4.3. As previously mentioned, this is accomplished very simply by running the *iperf-series* script with the spec mode argument and supplying a list of the desired rates. Guaranteeing a level of consistency useful for a comparative analysis of the competing approaches during randomized flow trials is somewhat more involved, but still made as simple as possible by the *iperf-series* script, which logs the random rate sequence generated when the script is run with the random mode argument. The first trial (typically AQoS) in a category is run with a randomized *iperf-series* job; the rates recorded in the log are then used to create *iperf-series* jobs for subsequent trials.

In practice, using multiple concurrent series of *iperfs* to generate traffic imposes practical limitations on the length of the intervals and the number rate changes that can be used. This is related to the fact that as concurrent *iperf* clients are launched and terminate, they tend to fall out of synchronization for a variety of reasons. The largest factor though seems to be that, as a side effect of the rate shaping with a sub-optimal meter implementation, *iperf2*'s L7 FIN/ACK packets tend to be lost in policing. This has the effect of delaying the *iperf2* client from closing as it resends its FIN packet up to ten times in an attempt to elicit a response from its *iperf2* server counterpart. Maximizing the quality of the data requires both increasing the length of the intervals so that a majority of the send rates across the

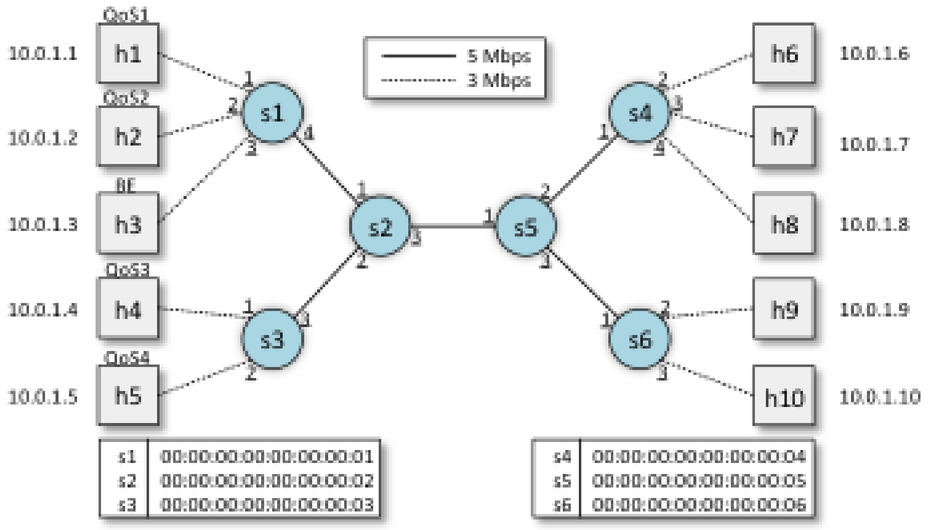

QoS1
QoS2
BE
QoS3
QoS4
10.0.1.1
10.0.1.2
10.0.1.3
10.0.1.4
10.0.1.5
h1
h2
h3
h4
h5
5 Mbps
3 Mbps
s1
s2
s3
s4
s5
s6
h6
h7
h8
h9
h10
10.0.1.6
10.0.1.7
10.0.1.8
10.0.1.9
10.0.1.10
s1 00:00:00:00:00:00:00:01
s2 00:00:00:00:00:00:00:02
s3 00:00:00:00:00:00:00:03
s4 00:00:00:00:00:00:00:04
s5 00:00:00:00:00:00:00:05
s6 00:00:00:00:00:00:00:06

Figure 5.2: Topology configuration specifications (repeated)

flows are as synchronous as possible, and reducing the number of rate changes since each is directly related to how often an *iperf* client attempts to FIN/ACK.

5.1.4 Data Collection

Data is gathered by the monitoring component through the controller's so-called southbound interface via the OpenFlow protocol. In addition to providing the data that drives the algorithm's control logic, the monitoring component logs all observed throughputs for known flows on their respective egress nodes, as well as control information such as the rate limit enforced for a flow at any given time and that flow's priority rating. Egress rate data is collected on switches *s4* and *s6* (Figure 5.2).

Throughputs collected from the egress node are not used by the AQoS algorithm but provide experimental data that can verify that flow rates are correctly adjusted. The data used to evaluate the algorithm's correctness and performance is taken from both sets of logs (ingress and egress). Data is collected in the same way during the benchmarking trials for

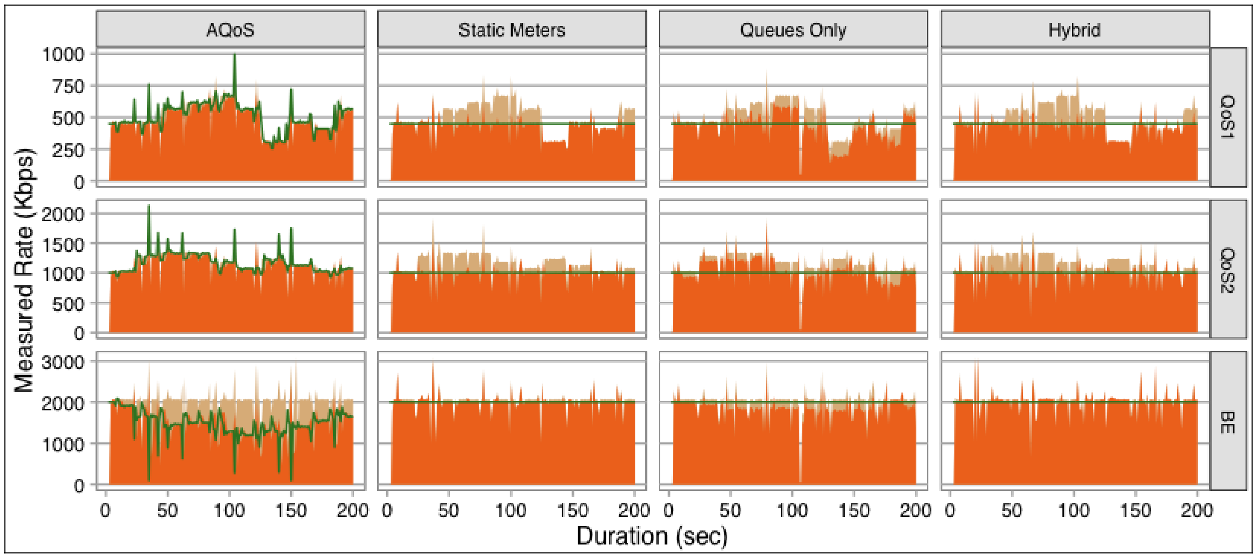


Figure 5.3: Measured ingress against egress rates, overlaid with tracking/reservation for QoS1, QoS2 and BE Flows, from 40% BE scenario

the alternative approaches, though the ingress rates are simply documented instead of used as a driver for the examined approach.

5.2 Analysis

The per-flow analyses shown in Figure 5.3 illustrate the behavior typical of the approaches considered. The results shown are taken from the 40% unreserved synthetic flow experiment. They are highly representative of the results gathered from all categories; the ingress-egress rate translation characteristics are nearly identical, differing only in how the differences scaled. They are also somewhat easier to digest than their randomized flow counterparts.

The AQoS algorithm results show that the QoS egress rates closely follow the ingress rates in all cases. Note that both are closely bounded by the tracking variable's value. The algorithm does exhibit sensitivity to significant upward spikes in the measured ingress rates.

This has the effect of eating into any remaining bandwidth that could be used by Best Effort traffic, with a corresponding reduction in the overall throughput of Best Effort flows.

Figure 5.3 also shows that the static meters method caps traffic at an unchanging rate so that flows coming into the network at higher rates are consistently throttled down to the static rate cap as expected. Unless ingress rates fall at or below the reserved rate, the static metering approach invariably results in considerably reduced throughput of QoS flows when compared to the AQoS algorithm results. On the other hand, Best Effort traffic fares better on average as no QoS flows are allowed to coopt available bandwidth.

The hybrid method results shown differ very little from the static meters results, which is not terribly surprising given the fact that only the Best Effort traffic is uncapped (i.e., not throttled back by a meter). Just as QoS flows were throttled down in the static meters method, the hybrid method results show considerably reduced QoS egress rates when compared against the AQoS results.

The final competing method for consideration is DiffServ queues. Figure 5.3 illustrates how DiffServ queues enforcing minimum guarantees shape traffic; in general, egress rates are still depressed, but the difference is proportional to the minimum guarantees of the other queues and the volume of traffic sent to them. While there is a marked improvement over both static metering and hybrid methods, QoS flow results under the DiffServ queue method still cannot compete with the AQoS algorithm's results in terms of QoS throughput. However, the situation is very different from the perspective of Best Effort traffic; Best Effort does extremely well under DiffServ queues. In randomized flow trials, typical Best Effort performance is second only to the hybrid method.

Some variability in the egress rates, where more traffic is sometimes observed exiting the network than should be observed given employed rate-shaping mechanisms, can be found on closer inspection of nearly all figures. This phenomenon manifests across all trials, and will be revisited in later discussion.

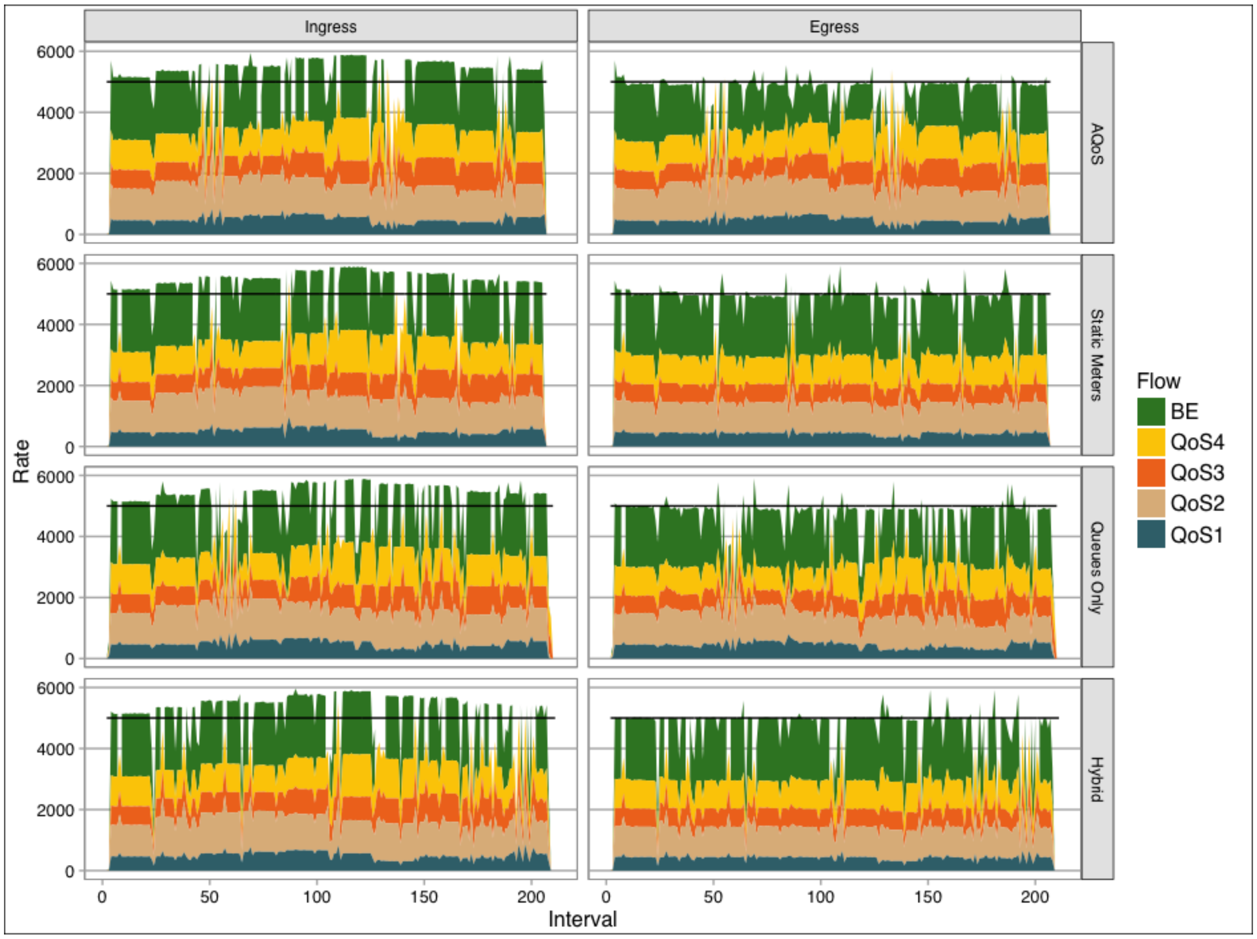


Figure 5.4: Cumulative ingress and egress rates against bottleneck capacity (rates in Kbps)

Figure 5.4 (above) shows cumulative ingress and egress observations for each class of flow. The graphs in these figures are drawn from the 40% Unserved synthetic flows experiment. Note from the top row of panels in Figure 5.4 that the AQoS algorithm results in cumulative QoS egress throughputs that are nearly identical to ingress rates, just as the individual flow analyses showed matching rates. Best Effort traffic alone is shaped down so that the total of all traffic is nearly 5 Mbps at all sampled intervals.

The Queues Only approach has a somewhat different effect on the overall measured throughputs. A comparison of ingress and egress rates in the second row of panels in Figure 5.4 corroborates earlier observations of the scaling effects of the queues. The overall shapes, or geometric properties of the throughput waveforms are preserved at all sampling intervals. Some attrition on the rates is apparent, but affects all traffic types. As expected, Best Effort throughputs are affected least by traffic shaping in the queues.

The egress throughputs resulting from the Static Meters method are the most uniform across all samples. Recall that the rate caps imposed by the meters throttle back all flows that are over their reservations. The third row of panels in Figure 5.4 illustrates the sum results of this approach to traffic shaping. The near uniformity is a result of the rates chosen for the synthetic flow trials, as QoS rates are often over. As expected, occasional drops of the QoS throughputs below their rates show as slight dips in the graphs. Unfortunately, the total throughputs of all traffic types does not fall below 5 Mbps and so this example fails to illustrate the tendency for the Static Meter approach to waste link capacity when there is not enough traffic to congest it.

The fourth panel row in Figure 5.4 illustrates an interesting point concerning the Hybrid approach. The observation that egress rates of the QoS flows in the third panel row are nearly identical clearly demonstrates that in terms of QoS flow performance, there is essentially no difference between the two approaches. The only salient difference is to the benefit of Best Effort traffic, as the Hybrid approach allows it alone to expand into unreserved bandwidth.

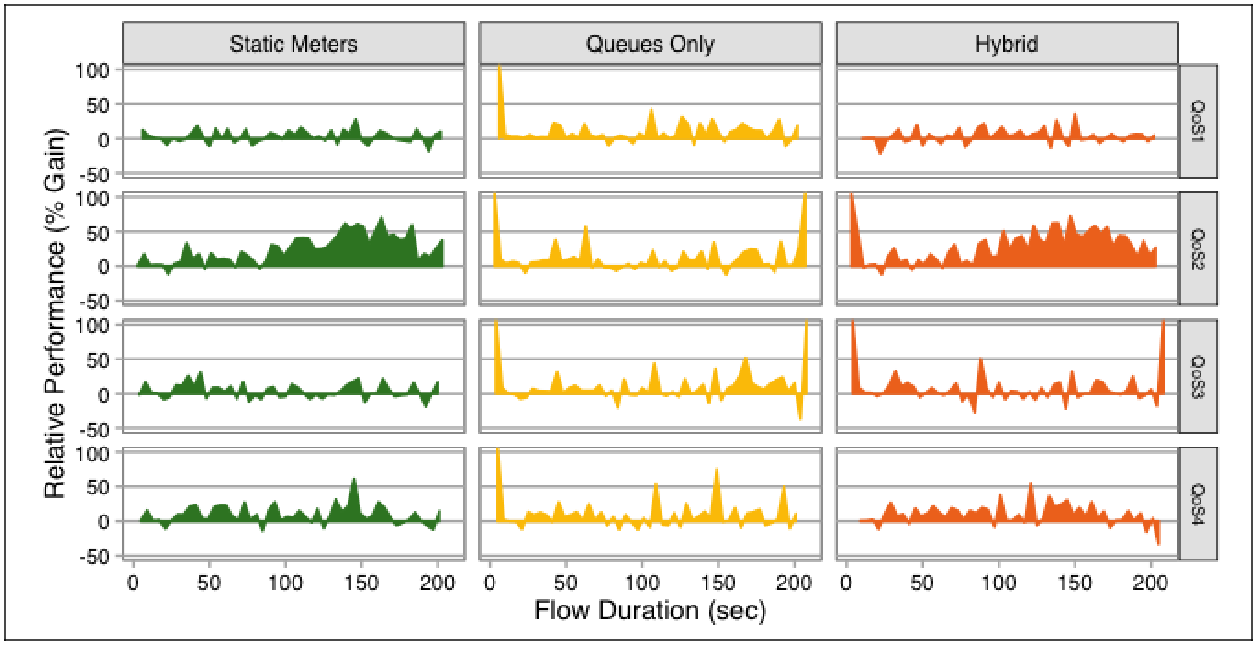


Figure 5.5: Performance gains for 20% Unreserved (BE) Scenario

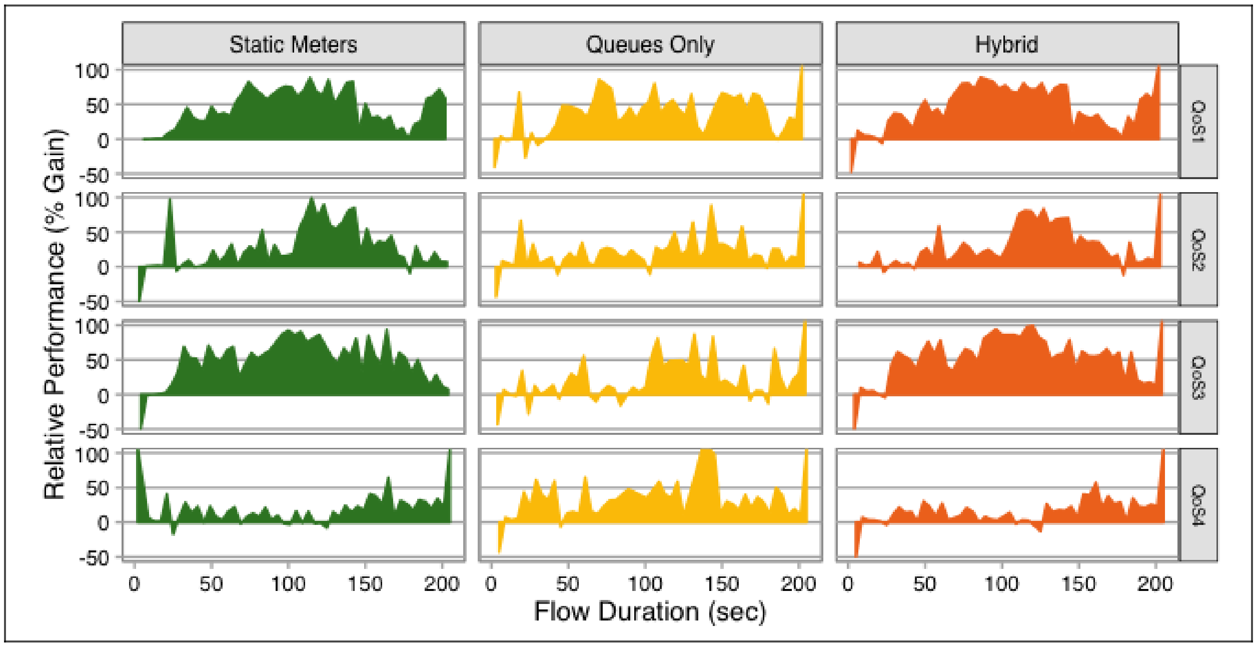


Figure 5.6: Performance gains for 50% Unreserved (BE) Scenario

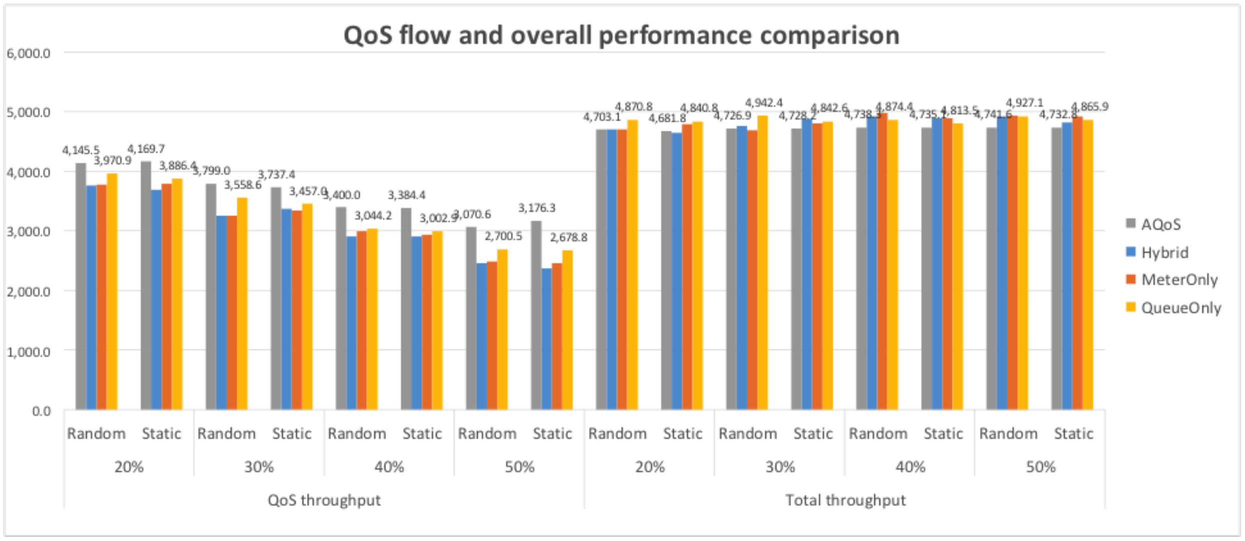
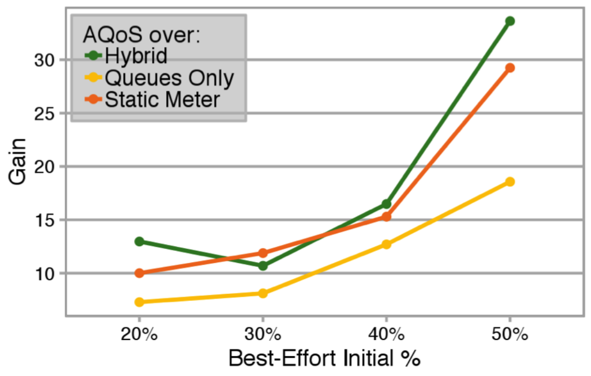


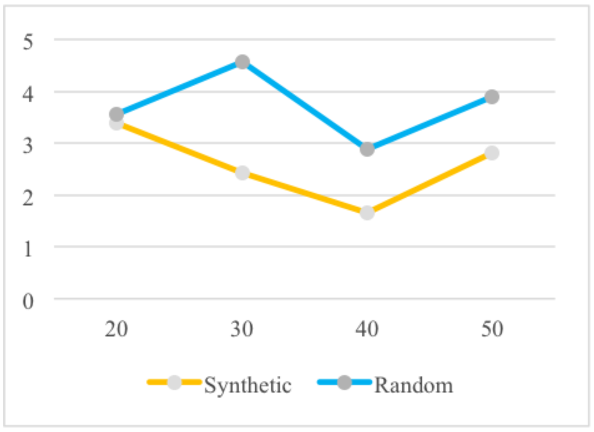
Figure 5.7: QoS and total average throughputs by traffic type and category

Figures 5.5 and 5.6 show QoS lifetime percent gains per flow achieved by the AQoS algorithm against competing approaches. Figure 5.5 illustrates the worst case performance in terms of amount of unreserved bandwidth available, while Figure 5.6 clearly illustrates highly favorable gains. Despite some dips into negative numbers, both cases clearly show the AQoS algorithm consistently giving better average performance against all other considered methods. These results by themselves are highly suggestive, but an increase in gains by unreserved bandwidth percentage becomes clearly evident when results from 30% and 40% unreserved scenarios are included (omitted for brevity).

Despite efforts to prevent it, some amount of noise enters the gains calculations where traffic generation rates do not quite match between AQoS and competitor, resulting in the minor downward spikes observed throughout. An examination of ingress measurements reveals that the significant spikes, both upwards and downwards, occur where AQoS or competitor QoS flow rates taper off asynchronously, which illustrates a more serious kind of mismatch between flow generation rates and points to flaws in the traffic generation implementation.



(a) AQoS QoS percent gain over DiffServ queues



(b) AQoS overall percent loss against DiffServ queues

Figure 5.8: AQoS QoS gains and overall throughput losses against DiffServ queues by scenario

Figure 5.7 shows a comparison of QoS and total throughputs by category. The AQoS algorithm shows average QoS throughput gains in every category for both flow generation schemes, synthetic and randomized. Figure 5.8a plots the gains as a function of unreserved (BE) bandwidth percentage for AQoS against the Queues Only method; the synthetic traffic gains exhibit an exponential increase, evident in the worst-case comparison (Queues Only) and becoming markedly more pronounced for other methods. In the worst case, taken from the 20% Best Effort randomized results, the AQoS algorithm shows a 4.40% gain over DiffServ queues. In the best case, taken from the 50% Best Effort synthetic results, the AQoS algorithm yields an 18.57% gain over DiffServ queues.

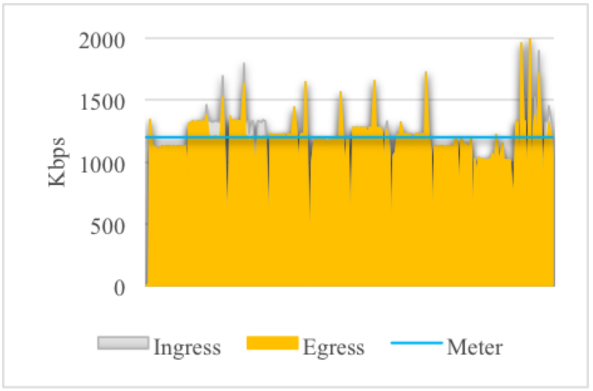
The situation differs when the total average throughput of all flows is considered; the AQoS algorithm performs consistently worse than all other competing methods. However, the calculated loss is consistently below 5% and as Figure 5.8b shows there does not appear to be any significant trend in any direction as Best Effort percentage grows.

CHAPTER 6
DISCUSSION AND CONCLUSIONS

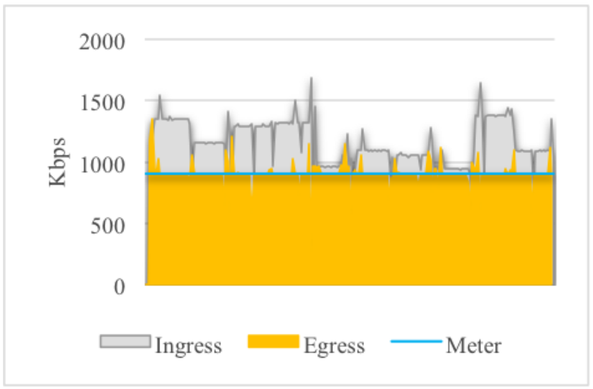
The results given in the average throughputs table are generally favorable towards the AQoS algorithm. QoS flows are clearly favored over the Best Effort flows, and as a result the average throughputs are substantially increased over the average QoS throughputs of all competing methods.

The results show however that there is a caveat; the total overall throughputs (including Best Effort) take a small hit when comparing the AQoS algorithm results against the DiffServ Queues and Hybrid methods. Some reduction in total average throughput was expected; the algorithm reserves a small percentage of bandwidth allocated away from underperforming QoS flows and uses it as a growth-detection buffer. That collective bandwidth is essentially lost to all flows when QoS flows underperform. However the average amount of total bandwidth lost during the AQoS algorithm trials is somewhat surprising. This in addition to the noise in measured rates and the occasional drops in traffic generation could explain some of the reduction. However, because the algorithm takes time to react to changing flows we expect there will always be a trade-off between QoS gains and total utilization.

These observations are reasonably correlated by the data. Note that, as illustrated by Figure 5.8b in §5.2, there is no significant increasing trend in total average bandwidth loss as unreserved bandwidth percentage increases. The amount of congestion on the line is also not expected to have any significant influence. However, increasing the number of QoS flows on the topology could have an impact if a substantial number were to underutilize their assigned bandwidths. This suggests that in addition to reassigning bandwidth, a better strategy might include adjusting the initial reservation to more closely match observed rates. Such a policy



(a) Egress rates consistently exceeding meter cap (QoS 2, 30% BE synthetic Static Meter)



(b) Egress rates generally conforming to meter cap (QoS 4, 30% BE randomized Static Meter)

Figure 6.1: Behaviors of the OpenFlow meter implementation

may or may not appeal to network administrators, though considering the algorithm's ability to meet QoS demands in the presence of sufficient unreserved bandwidth it likely makes little difference in practice.

There are also anomalies in the total average throughput results in the competing methods shown in Figure 5.7 of §5.2, notably static metering and DiffSev queues. In the case of static metering, the total average throughputs are unusually close to the bottleneck link capacity given the fact that, particularly in the synthetic trials, the flow rates were designed to guarantee that while some QoS flows were over their reservations, others would be under at least some of the time. However, in both the 30% and 40% Best Effort scenarios static metering gives total average throughputs higher than the DiffServe queues results. Barring measurement inaccuracies, the situation should be consistently reversed.

Figure 6.1a illustrates a significant problem with the current Open vSwitch meter implementation that sheds some light on these observations; for each flow shown there is a considerable amount of egress traffic in excess of the rate cap set by the meters. Furthermore, this excess is not consistent; a good portion of the time the metering is reasonably close to where it should be, as in Figure 6.1b. The problem appears in the results of all trials that relied on the OpenFlow meter mechanism but is most apparent in the static metering trials. Conversely, the results also show considerable drops in throughput below the meter rate, especially evident in the AQoS results. The fact that our experiment relies on a largely untested and unproven meter implementation must be taken into account, and casts some doubt upon the quality of the results.

Despite the fact that the reliability of the static metering trial results must be called into question, the main thrust of the benchmarking has always been targeted towards the AQoS algorithm and the DiffServ queues approaches. The truly critical question is, how much does this affect confidence in the AQoS trials results? As already discussed, the static metering trials exhibit the worst excess; though still present to some degree, the excessive egress rates are much less prominent in the AQoS algorithm trials. For example, in the 30% synthetic traffic AQoS algorithm trial, on average the observed egress rates exceeded their meter rates by better than 5% approximately 5% of the time, whereas the 30% synthetic static metering egress rates exceeded their meter rates (with the same margin of error) approximately 35% of the time on average. These percentages are fairly representative of the random rate trials as well as those of other Best Effort percentage categories. It has additionally been observed that egress rates tend to fall below rate limits during the AQoS trials; approximately 5% of QoS traffic is lost on average. In some isolated cases QoS flows lost as much as 10% of their expected egress traffic. The obvious difference between the AQoS trials and static meter trials is that rate limits are constantly adjusted in the former. This further reinforces the conclusion that faulty packet buffer management in the meter mechanism is at the heart of the observed inconsistencies.

The DiffServ queues total average throughputs were also somewhat lower than they should have been. Several known factors likely influenced the results: 1) Measurement inaccuracies due to clock drift; 2) the virtual nature of the network, which creates both significant memory and CPU scheduling overhead; and 3) the HTB Queuing Discipline

(QDisc), which is implemented in the network stack (i.e., kernel space) and heavily used by a user space virtual switch, where all traffic must be buffered and copied over between the switch process and HTB system calls. Finally, the user space switch instances are especially susceptible to the non-deterministic side effects of OS resource scheduling and virtual paging.

Despite these measurement and implementation challenges, the algorithm is expected to scale well in supplemental work. There is sufficient reason to expect that the algorithm could also scale in physical deployments with substantially larger and more complex topologies and hundreds or thousands of flows. Next steps include testing greater numbers of flows on larger, more complex topologies, verifying that the algorithm's resource footprint scales reasonably well as the number of managed flows increases, verifying that results are consistent when run on completely physical networks, and examining the algorithms' impact on TCP performance.

6.1 Conclusions

The results demonstrate that the AQoS algorithm delivers an overall increase in efficiency of QoS data transfers over competing approaches, from approximately 4% in the worst case to better than 18% in the best case. In contrast to initial expectations, this is accomplished at the expense of overall utilization of the network bottleneck links. The cost is fairly minimal, averaging around 2.4% in more ideal conditions, however any real-world application of this algorithm would need to take this factor into account; if a network application's primary goal is overall utilization then the AQoS algorithm would not be ideal. This may be a hard pill for network administrators to swallow in practice.

Another factor that must be considered in a real networking application is expected levels of congestion. As demonstrated, the gains achieved with the AQoS algorithm increase as both the volume of non-essential traffic and overall congestion increase. Only a close assessment of the expected frequency, duration and severity of competition between priority and non-priority traffic will answer the question of whether or not the AQoS algorithm provides sufficient returns to justify its overhead—in terms of controller resources and the network carrying control traffic, out-of-band or in-band—and a somewhat diminished overall network utilization.

REFERENCES

[1] W. Allcock, I. Foster, and S. Tuecke, "Protocols and services for distributed data-intensive science," in *AIP Conference Proceedings*, vol. 583, pp. 161–163, AIP, 2001.

[2] R. Kettimuthu, G. Vardoyan, G. Agrawal, and P. Sadayappan, "Modeling and optimizing large-scale wide-area data transfers," in *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, (Chicago), pp. 196–205, IEEE, 2014.

[3] W. Allcock, J. Bresnahan, R. Kettimuthu, M. Link, C. Dumitrescu, I. Raicu, and I. Foster, "The globus striped gridftp framework and server," in *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, (Washington, DC), pp. 54–55, 2005.

[4] C. Guok, D. Robertson, M. Thompson, J. Lee, B. Tierney, and W. Johnston, "Intra and interdomain circuit provisioning using the oscars reservation system," in *Conference on Broadband Communications, Networks and Systems, 2006. BROADNETS 2006. 3rd International*, (San Jose), pp. 1–8, 2006.

[5] E. Jung, R. Kettimuthu, and V. Vishwanath, "Cluster-to-cluster data transfer with data compression over wide-area networks," *Journal of Parallel and Distributed Computing*, vol. 79–80, pp. 90–103, May 2015.

[6] J. Bresnahan, M. Link, R. Kettimuthu, D. Fraser, and I. Foster, "Gridftp pipelining," in *TERAGRiD 2007 Conference*, (Madison), 2007.

[7] S. Schenker, "Prof. Scott Shenker - gentle introduction to software-defined networking - Technion lecture," 2012. Available: https://www.youtube.com/watch?v=eXsCQdshMr4.

[8] P. Goransson and C. Black, *Software Defined Networks: A Comprehensive Approach.* Waltham: M. Kaufmann, 2014.

[9] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, pp. 69–74, 2008.

[10] S. Vegesna, *IP Quality of Service.* Indianapolis: Cisco Press, 2001.

[11] B. A. A. Nunes, M. Mendonca, X. Nguyen, K. Obraczka, and T. Turletti, "A survey of software-defined networking: past, present and future of programmable networks," *IEEE Communcations Surveys Tutorials*, vol. 16, pp. 1617–1634, 2014.

[12] OpenFlow.org, "Openflow switch specification v1.0.0," 2009. Available: http://archive.openflow.org/documents/openflow-spec-v1.0.0.pdf.

[13] K. Nichols, S. Blake, F. Baker, and D. Black, "RFC 2474: Definition of the differentiated services field (DS field)," 1998.

[14] J. Case, M. Fedor, M. Schoffstall, and J. Davin, "RFC 1157: Simple network management protocol," May 1990.

[15] O. N. Foundation, "Openflow switch specification v1.3.0," 2013. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.0.pdf.

[16] N. S. V. Rao, W. R. Wing, S. M. Carter, and Q. Wu, "Ultrascience net: Network testbed for large-scale science applications," *IEEE Communications*, vol. 43, pp. s12–s17, November 2005.

[17] N. S. V. Rao, Q. Wu, S. Ding, S. M. Carter, W. R. Wing, A. Banerjee, D. Ghosal, and B. Mukherjee, "Control plane for advance bandwidth scheduling in ultra high-speed networks," in *High-Speed Networking Workshop: The Terabits Challenge. IEEE INFOCOM Workshop on*, IEEE, 2006.

[18] I. Foster, A. Roy, and V. Sander, "A quality of service architecture that combines resource reservation and application adaptation," in *Quality of Service, 2000. IWQOS. 2000 Eighth International Workshop on*, IWQOS2000, (Pittsburgh, PA, USA), pp. 181–188, IEEE, 2000.

[19] I. Foster, M. Fidler, A. Roy, V. Sander, and L. Winkler, "End-to-end quality of service for high-end applications," *Computer Communications*, vol. 27, no. 14, pp. 1375–1388, 2004.

[20] R. Wallner and R. Cannistra, "An sdn approach: Quality of service using big switchs floodlight open-source controller," in *Proceedings of the Asia-Pacific Advanced Network*, vol. 35, pp. 14–19, 2013.

[21] B. Networks, "Project Floodlight." Available: http://www.projectfloodlight.org/floodlight.

[22] S. Abdellatif, P. Berthou, P. Gelard, T. Plesse, and S. El-Yous, "Exposing an openflow switch abstraction of the satellite segment to virtual network operators," in *IEEE 83rd Vehicular Technology Conference*, (Nanjing, China), IEEE, 2016.

[23] V. Hazlewood, K. Benninger, G. Peterson, J. Charcalla, B. Sparks, J. Hanley, A. Adams, B. Learn, R. Budden, D. Simmel, J. Lappa, and J. Yanovich, "Developing applications with networking capabilities via end-to-end SDN (DANCES)," in *XSEDE16 Conference on Diversity, Big Data, and Science at Scale*, (Miami), p. 29, 2016.

[24] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wandere, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat, "B4: experience with a globally-deployed software defined wan," in *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, (Hong Kong, China), pp. 3–14, 2013.

[25] C. Wong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, "Achieving high utilization with software-driven wan," in *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, (Hong Kong, China), pp. 15–26, 2013.

[26] B. Networks, "OpenFlowJ-Loxi Project." Available: https://github.com/floodlight/loxigen/wiki/OpenFlowJ-Loxi.

[27] L. Foundation, "Open vSwitch." Available: http://openvswitch.org/.

[28] E. S. Network, "nuttcp." Available: https://fasterdata.es.net/performance-testing/network-troubleshooting-tools/nuttcp/.