Adaptive QoS for Data Transfers Using Software-Defined Networking

Joshua M. Boley
Northern Illinois University
jboley@anl.gov

Eun-Sung Jung
Hongik University
ejung@hongik.ac.kr

Rajkumar Kettimuthu
Argonne National Laboratory
kettimut@anl.gov

Abstract

The research community has devoted much attention to the use of classic network technologies for advanced traffic engineering, but the use of software-defined networking (SDN) for this purpose, particularly in regards to Quality of Service (QoS) optimizations, remains relatively unexplored. We have developed a QoS framework that leverages SDN capabilities to achieve optimal throughputs for all QoS flows on a congested network. Specifically, we propose an approach that reassigns bandwidth unused by one or more flows to other flows in priority order, while simultaneously ensuring that QoS flows can reclaim their reassigned bandwidth whenever they are able to use that bandwidth. At the heart of our framework is the Adaptive Quality of Service (AQoS) algorithm, inspired by Integrated Services (IntServ) principles, which enables fine-tuned, real-time control over per-flow bandwidth allocations, combined with classic Differentiated Services (DiffServ) priority classes that determine how bandwidth is distributed. We evaluate and compare the results of the AQoS throughput optimizations to those achieved by several baseline algorithms common to both classic and SDN networks, with promising results.

I. Introduction

Software-defined networking (SDN) is a relatively new technology that attempts to solve the age-old problem of homogenizing and centralizing both the management and routing intelligence of a network. SDN separates the control plane from the network fabric, consolidates all the routing logic in a central controller, and establishes a single southbound API or protocol through which a controller communicates with all devices in the network. The OpenFlow protocol has become the predominant open standard for the southbound protocols and is the mainstay of the major open source SDN controller platforms such as BigSwitch Networks' Floodlight [1] and OpenDaylight [2].

The conventional two quality-of-service (QoS) models are the integrated service (IntServ) model and the differentiated service (DiffServ) model, which correspond to per-flow-based QoS control and class-based QoS control, respectively. In practice, QoS models are implemented by queues and meters (leaky buckets) in switches. In SDN, the first official draft of the OpenFlow specification (1.0) provided for basic class-of-service queues that attach to a specific port. These queues, like the traditional ones, must be configured directly on the switch and cannot be administered by the controller. Bandwidth unused by any class of traffic will be proportionally split across the rest of classes based on the static assignment. The OpenFlow standard was later expanded (1.3) to include the notion of metering flows. Meters impose a rate limit on both an aggregate and per-flow basis and can be created, updated, and deleted by the controller ad hoc, providing a mechanism for fine-tuning bandwidth allocation between flows.

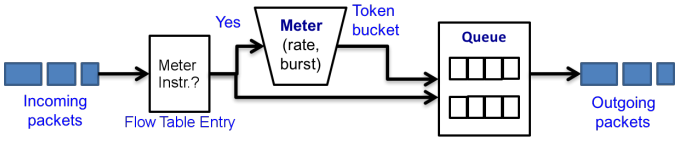

Fig. 1: Typical packet processing pipeline

A typical packet processing pipeline for the OpenFlow 1.3 meter is shown in Figure 1. Incoming packets are forwarded to a meter as instructed by a matching flow table entry. The switch looks up the corresponding meter on the switch's meter table, which buffers and passes or drops packets by the associated token bucket's rate and burst size.

Monitoring and allocation logic is required in addition to OpenFlow meters to dynamically allocate unused bandwidth to specific flows. It is important to do this in a nondisruptive fashion; the flow whose unused bandwidth is allocated to other flows should be allowed to reclaim that bandwidth whenever it can do so.

The following is one example of how an application's network bandwidth requirement can go below or above the reserved bandwidth. In the current distributed high-performance computing environments, most common deployments have data transfer nodes (DTNs) and compute nodes mount a global shared parallel storage system, which is connected through a storage area network. In such scenarios, one cannot contention-free access to the storage system. Also, DTNs do not support scheduling CPU resources. As a result, the application can sometimes use only a fraction of the reserved bandwidth on wide-are networks (WANs) and local-area networks (LANs), and the rest can go unused when circumstances conspire to throttle the throughput of the hosts connection. (We note that even though bandwidth may not go unused in a queues only approach, the static nature of queue weight setting may result in an undesired distribution of the unused bandwidth.) Moreover, sometimes (when is there no contention for DTN CPU and storage resources), the application might be able to use more than what is reserved on the network.

We introduce an adaptive quality-of-service (AQoS) flow management algorithm that takes advantage of the SDN controller's ability to gather per-flow statistics and shape flows through the OpenFlow protocol. The algorithm—built

on open-source SDN technology—adjusts to optimize QoS throughputs in real time while observing QoS guarantees on a per-flow basis. Lightweight network load monitoring, flow classification, and QoS enactment components support the control logic. We evaluate our design and implementation on a virtual network of Open vSwitch instances and host processes.

In Section II, we present a short history and state of the art of SDN. Related work is covered in Section III. In Section IV, we describe the problems we address, target networks, and applications. In Section V, we present our proposed QoS control methods. In Section VI, we evaluate our approach. In Section VII, we present our conclusions and briefly discuss future work.

II. Software-Defined Networking

Software-Defined Networking is a relatively new networking paradigm that encompasses a wide variety of networking technologies designed to enhance the agility and flexibility of physical (and by extension, virtual) network infrastructure. Interest in SDN technologies was initially driven by problems arising from the exploding trend towards virtualization of the data center. The application of SDN has since expanded far beyond its original scope. Much of SDNs appeal arises from its fundamental design principle of separating control logic from the physical devices and operating systems that implement the data plane. Network logic is centralized and thus a controller has a top-level view of the entire network.

While the foundational principles and concepts underlying the SDN concept are not new, recent SDN work builds on the OpenFlow protocol [3], [4]. This protocol bridges the controller software and network operating system devices with an open specification. Because of its flexibility and transparency, the OpenFlow protocol has become widely recognized as a de facto standard and a driving force in the network industry's adoption of SDN technology.

Prominent open source SDN controllers include BigSwitch Networks Floodlight [1] and OpenDaylight [2]. Both controllers are written in Java and so enjoy portability across multiple platforms. They additionally allow a user to implement custom functionality through extensive plugin APIs, opening the door to novel, custom-tailored network control frameworks.

III. Related Work

SDN has been applied to various contexts ranging from automated network path reconfiguration to QoS enforcement [6]. The most similar works to our proposal are SWAN [7] by Microsoft and B4 [8] by Google. The most distinguishable point from these works is that they do not provide guarantee on QoS requirements of flows. They instead focus on achieving overall high utilization of networks; if flows are increasing or decreasing (as reported by clients or detected by a controller), bandwidth is redistributed among flows based on fairness algorithms such as max-min. In [9], the authors propose the use of OpenFlow meters to control bandwidth for individual flows, which is also similar to the work presented here except that meter rates are static and contention is addressed via resource-scheduling.

Others have pursued similar approaches in the classic (non-SDN) networking space. For example, the GARA framework presented in [10], [11] also emphasized adaptation of the framework to network conditions in real time. Similar to the DANCES framework in [9], previous work on UltraScience Net [12] presented in [13] relied on a bandwidth-scheduling algorithm and a signaling daemon to build up and tear down virtual circuits using the TL1 protocol, which can be loosely considered a forerunner to the OpenFlow protocol. Those efforts faced difficulties in surmounting the distributed nature of the networking intelligence. Software-defined Networking eliminates many of these barriers, simplifying the development of contemporary, related frameworks.

IV. Problem Statement

We target distributed applications sending and receiving QoS flows through end-to-end network paths spanning local-area and wide-area networks. We assume that all routers and switches related to an end-to-end network path are OpenFlow compatible and connected to a centralized OpenFlow controller through a control plane. Flows are categorized into QoS and Best-effort (BE). Quality of service is considered only in terms of bandwidth requirements. Best-effort refers to all flows with no implied SLAs; thus, BE flows receive no bandwidth guarantees.

QoS flows are further categorized into multiple classes with different priorities. We assume that each class has preassigned relative bandwidth share (e.g., best-effort flow class gets 10% of link capacity) or fixed bandwidth value (e.g., QoS class 4 gets 1 Gbps).

We formally define a flow as follows: $flow = (s, sp, d, dp, AT, ST, Proto, D, class, minBW)$, where s is source IP, sp is source port, d is destination IP, dp is destination port, AT is arrival time, ST is start time, $Proto$ is network protocol type, D is duration, $class$ is the class that the flow belongs to, and $minBW$ is minimum bandwidth requirement. Our goal is to achieve high bandwidth utilization (for QoS flows in particular) by reassigning bandwidth unused by one or more flows to other flows (in the order of their priority), while ensuring that reassigned bandwidth of QoS flows is handed back to them when they need it.

V. Novel Adaptive QoS Control Methods

A. Algorithm overview

Our algorithm tackles the bandwidth-sharing problem by leveraging the following principles.

- **Redistribution of unused bandwidth.** We distinguish between *QoS flows*, which have a bandwidth allocation and priority, and other *Best-effort flows*. Bandwidth from under-performing QoS flows is redistributed among other flows, with QoS flows taking precedence.
- **Priority-driven bandwidth assignment.** We selectively allow flows to expand into available bandwidth on the basis of priority rating. Highest-priority flows are allowed

Fig. 2: Illustrative comparison of hypothetical flow rates by algorithm at arbitrary times.

to expand into unused bandwidth first. Remaining unused bandwidth is then passed down to lower-priority flows. Leftover bandwidth is finally handed over to BE.

Figure 2 compares expected performance for each algorithm in the following hypothetical scenario. Two QoS flows (QoS1 and QoS2) with initial reservations at 1 and 2 Mbps, respectively, with the remaining 7 Mbps going to BE, on a bottleneck link with total capacity of 10 Mbps are shown at times t_1, t_2, t_3, and t_4. We assume that, due to unspecified but changing conditions outside of the local network, incoming flow rates are subject to the given changes at each time. The Static Meter illustration assumes rate limits for QoS1, QoS2, and BE at 1, 2, and 7 Mbps respectively. The DiffServ queue example assumes three queues with ratios 1:2:7. The hybrid approach assumes a DiffServ two-queue configuration with a 3:7 ratio combined with metering at the same rates as Static Meter for the QoS1 and QoS2 flows only. Ingress rates are specified at each interval.

The postulated results presented here illustrate the AQoS algorithm's unique preference for QoS flows. The AQoS algorithm is specifically designed to secure bandwidth for growing QoS flows while simultaneously ensuring that other QoS flows have the bandwidth necessary to meet QoS requirements.

B. Algorithm design

The AQoS algorithm was designed as an extension to the control plane. Most open-source controller platforms offer APIs for adding features and enhancements. The Floodlight plugin API, for instance, offers fine-grained control over packet event and OpenFlow message processing via the southbound API. The northbound (REST) API, on the other hand, has seen little standardization and is much less suited to fine-grained, novel data plane management approaches. Extensions to the northbound API are possible but introduce an unacceptable level of complexity for our purposes.

Algorithm 1 AQoS Controller Algorithm
MANAGE-FLOWS(networkGraph, flowDB, networkMonitor)

```
1: for each flow in flowDB do
2:     ADD-FLOW-DATA(networkMonitor, flow.id, flow.history)
3:     CLASSIFY(flow.history)
4: end for
5: SEIZE-BW(networkGraph, flowDB)
6: RETURN-BW(networkGraph, flowDB)
7: SHARE-AVAILABLE-BW(networkGraph, flowDB)
8: for each flow in flowDB do
9:     COMMIT-DROP-RATE(flow)
10: end for
```

The AQoS algorithm additionally relies on a network monitoring component, also implemented as an extension to the controller. This component gathers flow state information from the network switches through the OpenFlow table stats query protocol. Throughputs are computed and then made available to the algorithm on a per-flow, per-ingress-node basis. The ingress nodes are the only point at which the throughputs seen by the algorithm are gathered. All throughputs gathered by the network monitor component are logged for analysis. For our experiment we additionally gather throughputs on the flow egress switches.

The overall algorithm, shown in Algorithm 1, comprises three primary components that we describe in the following. These components execute in sequence, each responsible for handling a different aspect of analysis and control. They are supported by other minor subroutines that perform such tasks as calculating flow rates from the raw data returned from the network devices and classifying flows.

1) SEIZE-BW() function: This component is responsible for identifying flows that have gone below or are in the process of shrinking below their reserved rates. All QoS flows are examined, and any unused bandwidth is added to the pool of bandwidth available to other flows along the flow's path through the network. Under-performing QoS flows are flagged and subsequently ignored by the remaining components. The detailed algorithm is shown in Algorithm 2. The algorithm calculates a threshold based on previous observations. Any flow with a current rate below the threshold is eligible for seizure of unused bandwidth. Note that the modeled rate is not necessarily equal to the measured rate but could reflect a previous stable rate or a rate the algorithm anticipates the flow will soon be at. Then the algorithm gets the latest observation statistics from the flow's history and uses these to identify any sharp decreases that could signal a developing downward trend.

2) RETURN-BW() function: The second component is responsible for identifying flows that have gone below their reservations but are starting to grow back. All under-performing flows have been identified and their unused bandwidth logged, so any flow growing back into its reserved rate may borrow from unused bandwidth available end-to-end. In the case there is insufficient end-to-end bandwidth the component starts looking at other QoS flows, from lowest priority on up, and takes required bandwidth from flows above their reservation. As flows are not admitted unless sufficient

Algorithm 2 SEIZE-BW(networkGraph, flowDB)

```
for each flow in flowDB do
    lowerBound ← CALC-THRESHOLD(flow.tracking, flow.rate, flow.history)
    if flow.class = DECREASE and flow.rate < lowerBound then
        decrease ← CALC-OPTIMAL-DECREASE(flow.tracking, flow.rate, flow.history)
        flow.tracking ← flow.tracking - decrease
        ADD-TO-AVAILABLE-POOLS(networkGraph, flow.route, decrease)
    end if
end for
```

Algorithm 3 RETURN-BW(networkGraph, flowDB)

```
for each flow in flowDB with flow not flow.flagged do
    if INSIDE-GROWTH-MARGIN(flow.rate, flow.margin) then
        available ← BW-ON-PATH(networkGraph, flow.route)
        optimal ← CALC-OPTIMAL-INCREASE(flow.tracking, flow.rate, flow.history)
        returned ← MIN(available, optimal)
        flow.tracking ← flow.tracking + returned
        CLAIM-BW(networkGraph, flow.route, returned)
        remaining ← optimal - returned
        if remaining > 0 then
            other ← GET-LOWEST-PRIORITY(flowDB, flow)
            while remaining > 0 do
                returned ← returned + TAKE-BW(networkGraph, flowDB, other, remaining)
                flow.tracking ← flow.tracking + returned
                remaining ← remaining − returned
                other ← GET-NEXT-LOWEST(flowDB, other, flow)
            end while
        end if
        flow.flagged ← GROWING
    end if
end for
```

bandwidth is available, a QoS flow is always guaranteed its original reservation. The detailed algorithm is shown in Algorithm 3.

3) SHARE-AVAILABLE-BW() function: The third component handles the sharing out of available bandwidth to QoS flows that are near or above their reservations and can use it. The amount given to a flow is constrained by the least available per-link bandwidth on the flow's path. The detailed algorithm is shown in Algorithm 4.

VI. Experimental Evaluation

We next describe the testbed on which we evaluate our methods and compare the performance of our traffic-engineering approach with existing QoS management methods. We evaluate our algorithm against the time-honored Diff-Serv Queueing technique, as well as against static metering and a static meter and DiffServ Queueing hybrid approach. We show that the algorithm proposed in this paper can deliver both better overall and per-flow performance for QoS flows than other schemes.

A. Testbed setup

The virtual network used in our experiments was hosted on a custom-built server with a single Intel 8-core processor (with Hyper-Threading enabled) at 3.2 GHz and a Solid State Drive

Algorithm 4 SHARE-BW(networkGraph, flowDB)

```
for flow in flowDB from flow.priority = HIGHEST to flow.priority = LOWEST with flow not flow.flagged do
    lowerBound ← flow.reserved − CALC-NOMINAL-WINDOW(flow.rate, flow.history)/2
    if flow.tracking ≥ lowerBound then
        available ← BW-ON-PATH(networkGraph, flow.route)
        optimal ← CALC-OPTIMAL-INCREASE(flow.tracking, flow.rate, flow.history)
        borrowed ← MIN(available, optimal)
        flow.tracking ← flow.tracking + borrowed
        CLAIM-BW(networkGraph, flow.route)
    end if
end for
```

for storage and swap. Previous attempts to gather data from host machines with lesser specifications yielded inconsistent results, making comparisons between competing algorithms unreliable. The Ubuntu Server 16.04 LTS distribution with the kernel backported to 4.2 (for compatibility reasons) was installed as it is a well-supported and relatively simple Linux distribution to maintain. A version of Open vSwitch that our team patched with an existing meter implementation for the user space datapath was then installed.

The simulation architecture is composed of one controller, multiple virtual switches, and virtual hosts running traffic generators. A ten-host, six-switch bottleneck topology was designed for our tests, with source and corresponding sink hosts on opposite sides of the bottleneck, one source-sink pair per flow. All flows intersect in the bottleneck before fanning out to their destinations. This configuration was chosen for several reasons. First, we wanted as compact a design as possible given the expected limitations of running a virtual network on commodity hardware. Second, the bottleneck design is specifically intended to guarantee a point of congestion, which we need to demonstrate QoS throughput gains in our framework. Third, the two ingress-switch design demonstrates the AQoS algorithm's ability to coordinate traffic control mechanisms across multiple points of ingress on the network.

Figure 3 illustrates the topology used for all tests. Inter-switch and host link capacities are 5 and 3 Mbps, respectively. The host link capacities are an artifact of the implementation method; we found that the combination of user space switch instances, virtual Ethernet pairs, and Linux network namespaces (to encapsulate host processes) seriously throttles host link capacities.

We create a Floodlight controller instance on the host OS and preconfigure the reservations for our experiments during initialization. Supporting code in the AQoS algorithm module finds a circuit based on reservation endpoints (i.e., host addresses) via an OSPF-like search of the virtual network topology, which is known to the controller. After the virtual switches finish connecting to the controller, the controller then pushes IPv4 circuits down to the virtual network fabric and creates the required meters and metering instructions on the

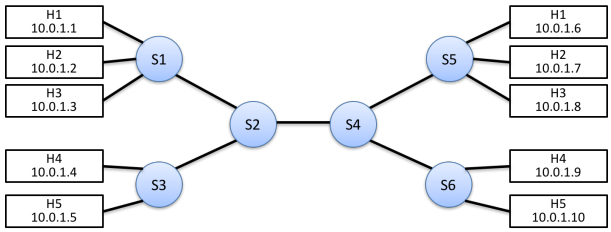

Fig. 3: Network topology for simulation.

ingress switches. Once traffic generation begins, the incoming packets are sent to the meters associated with the flows' circuits.

The controller then begins monitoring flow rates on both ingress and egress switches on a per-flow basis. The observed ingress rates in turn drive the algorithm's logic as it attempts to maximize bandwidth utilization among priority flows.

B. Methodology

The evaluation is divided into four experiments, distinguished by what percentage of the bottleneck link capacity is initially given over to Best-effort traffic: 20% in the first experiment, 30% in the second, 40% in the third, and 50% in the fourth. This design is intended to illustrate increasing QoS gains of the AQoS algorithm compared to the other schemes with increasing saturation of non-priority traffic.

In each experiment we gather performance metrics for the AQoS, DiffServ Queues, Static Meter, and Hybrid algorithms in separate trials, one trial per algorithm for a total of four per experiment. We define an interval as a period of time during which a flow is generated at a (relatively) constant rate. Each trial lasts for 200 seconds, comprised of 10 intervals of 20 seconds. Traffic generators are programmed to produce traffic at a given rate during this time before switching to a new rate during the next interval.

Each trial uses four QoS flows and one Best-effort (BE) flow, designated QoS1, QoS2, QoS3, QoS4, and BE. We reserve a set amount of bandwidth, varied by experiment, for each QoS flow. Table I lists the minimum and maximum rates (or ranges) within which QoS and BE traffic generation rates may vary (by experiment). All trials in an experiment use the same rates and reservations. Cumulative flow rates are designed to saturate the 5 Mbps bottleneck during most (but not all) intervals. Queue configurations for trials requiring DiffServ Queues are given in Table II.

The use of TCP flows was initially considered but discarded since UDP flows provide the most efficient means of congesting the bottleneck link. UDP flows also allow us to examine the behavior of the AQoS algorithm under specific, controlled conditions that are difficult to emulate with TCP. Evaluation of the AQoS algorithm with TCP flows is, however, an important next step.

C. Analysis and discussion

The same flow rate data that drove the algorithm during the AQoS algorithm trial was also logged in all trials for use in our evaluation of the different approaches we tested. Egress rate observations were recorded in addition to ingress rates. These data points were used to profile flow performance in terms of egress against ingress rates on a per-flow basis, including BE.

Representative comparisons of the AQoS with baseline algorithms, taken from the 40% BE trials, are shown in Figure 4. Percent gains in the egress rates for AQoS over competing algorithms are plotted against flow duration. Average gains for QoS1, QoS2, QoS3 and QoS4 relative to DiffServ Queues (Queues Only) were 12.9%, 7.0%, 13.6% and 13.0%, respectively; relative to Static Meter, 14.5%, 14.6%, 22.0% and 9.6%; and relative to Hybrid, 16.6%, 16.0%, 24.6% and 11.7%. Plots for QoS2 and QoS4 are omitted from Figure 4 due to space constraints. The results generally align with expectations. DiffServ Queues yielded the next best QoS throughputs after AQoS, followed by Static Meter and Hybrid.

The results also highlight a flaw in the traffic generation scheme. At several points over a flow's duration breaks between consecutive *iperf*s cause ingress rates to momentarily plummet. These events occur across all algorithms but rarely coincide, simultaneously yielding artificially low gains in some places and high gains in others. Frequencies are relatively consistent across trials and therefore have minimal effect on averages. Rate changes were also observed to gradually fall out of sync between trials, with similar although less-exaggerated results.

The AQoS algorithm showed significantly improved QoS throughputs over all competing algorithms, as Figure 5 illustrates. The 20% BE experiment yielded the most modest gains for the AQoS algorithm, which showed a 7.29% gain over DiffServ Queues, 10.0% over Static Meter, and 13.0% gain over Hybrid. The highest returns were observed in the 50% BE experiment, with AQoS yielding an 18.6% gain over DiffServ Queues, 29.2% gain over Static Meter, and 33.6% gain over Hybrid. Results are proportional to the amount of BE traffic, which is fair game for QoS flows, and the average ingress rates of the QoS flows.

On the other hand, overall AQoS throughputs fell somewhat below those yielded by other methods. The loss appears insensitive to QoS or BE traffic proportions, averaging 2.48%. We consider this acceptable but intend to explore the problem in future work.

The results show that given a congested link the AQoS algorithm delivers improvements to QoS throughputs, with returns becoming more substantial as Best-effort traffic increases. Moreover, the AQoS algorithm is consistently able to guarantee the bandwidth requirements of all QoS flows.

VII. Conclusions

We presented an adaptive QoS management framework that uses software defined networking (i.e., OpenFlow) methods to effectively transfer multiple QoS data flows along with Best-effort flows. We have evaluated this framework, together with our adaptive QoS reallocation algorithm, relative to three existing QoS methods: the queue only method, the meter only method, and a hybrid of queues and meters. Simulation results show that our framework outperforms those other methods by

TABLE I: Flow reservation and min/max rate specifications (in Kbps)

			Experiment 1 (BE at 20%)			Experiment 2 (BE at 30%)			Experiment 3 (BE at 40%)			Experiment 4 (BE at 50%)		
Flow	Src	Dst	Reserv.	Min	Max	Reserved	Min	Max	Reserved	Min	Max	Reserved	Min	Max
QoS1	10.0.1.1	10.0.1.6	700	550	750	500	300	900	450	300	650	350	350	650
QoS2	10.0.1.2	10.0.1.7	1200	1200	1850	1200	1000	1300	1000	1000	1300	950	900	1650
QoS3	10.0.1.4	10.0.1.8	1100	600	1250	800	700	1100	600	600	950	400	400	800
QoS4	10.0.1.5	10.0.1.10	1000	950	1200	1000	900	1350	950	850	1350	800	700	1050
BE	10.0.1.3	10.0.1.8	-	1000	1000	-	1500	1500	-	2000	2000	-	2500	2500

TABLE II: Queue configurations per experiment and per-queue flow assignments

Minimum Rates (Kbps)

Experiment	**Queue 1**	**Queue 2**	**Queue 3**
20% Best-Effort	1900	2100	1000
30% Best-Effort	1700	1800	1500
40% Best-Effort	1450	1550	2000
50% Best-Effort	1300	1200	2500

Flow Assignments

Queue 1	**Queue 2**	**Queue 3**
QoS1	QoS3	BE
QoS2	QoS4	

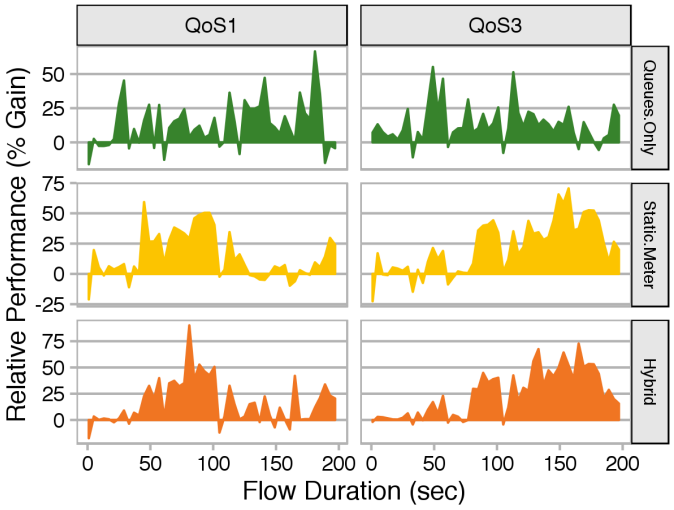

Fig. 4: AQoS performance gain against competing algorithms over the lifetimes of representative QoS flows, taken from 40% Best-effort experiment. Flow durations = 200 secs.

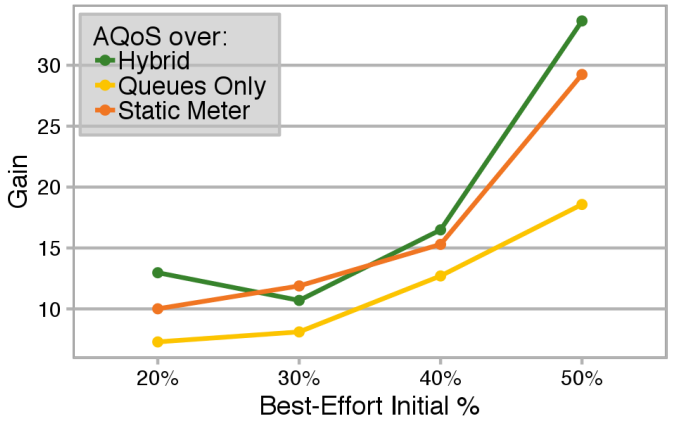

Fig. 5: AQoS percentage gains in overall QoS throughputs over competing algorithms by best-effort initial percentage

up to 18%, while meeting the QoS requirements of all QoS flows. In future work we plan to conduct extensive experiments on a real testbed such as ESnet. Work is in progress to evaluate how well the algorithm scales to larger, more complex network topologies and greater numbers of QoS and Best-effort flows.

Acknowledgments

This work was supported in part by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research Program, under Contract DE-AC02-06CH11357 and the Software Defined Network Science Flows project.

References

[1] "Project Floodlight." [Online]. Available: http://www.projectfloodlight.org/floodlight/

[2] "OpenDaylight: Open Source SDN Platform." [Online]. Available: https://www.opendaylight.org

[3] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008.

[4] "Openflow switch specication." [Online]. Available: http://archive.openflow.org/documents/openflow-spec-v1.0.0.pdf

[5] "Open vSwitch." [Online]. Available: http://openvswitch.org/

[6] D. Kreutz, F. M. V. Ramos, P. E. Verssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-Defined Networking: A Comprehensive Survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, Jan. 2015.

[7] C.-Y. Hong, 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*, ser. SIGCOMM '13. New York, NY, USA: ACM, 2013, pp. 15–26.

[8] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hlzle, S. Stuart, and A. Vahdat, "B4: Experience with a Globally-deployed Software Defined Wan," *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 3–14, Aug. 2013.

[9] 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*, 2016, p. 29.

[10] 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*, ser. IWQOS2000. Pittsburgh, PA, USA: IEEE, 2000, pp. 181–188.

[11] 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.

[12] S. M. C. Q. W. N. S. V. Rao, W. R. Wing, "Ultrascience net: Network testbed for large-scale science applications," *IEEE Communications*, vol. 43, no. 11, pp. s12–s17, November 2005.

[13] S. D. S. M. C. W. R. W. A. B. D. G. N. S. V. Rao, Q. Wu 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.