Automated and Scalable QoS Control for Network Convergence

Wonho Kim*, Puneet Sharma+, Jeongkeun Lee+, Sujata Banerjee+,
Jean Tourrilhes+, Sung-Ju Lee+, and Praveen Yalagandula+
*Princeton University +HP Labs

Abstract

Network convergence is becoming increasingly important for cost reduction and management simplification. However, this convergence requires strict performance isolation while keeping fine-grained control of each service (e.g. VoIP, video conference etc.). It is difficult to guarantee the performance requirements for various services with manual configuration of the Quality-of-Service (QoS) knobs on a per-device basis as is prevalent today. We propose a network QoS control framework for converged fabrics that automatically and flexibly programs a network of devices with the necessary QoS parameters, derived from a high level set of application requirements. The controller leverages our QoS extensions of OpenFlow APIs, including per-flow rate-limiters and dynamic priority assignment. We also present some results from a testbed implementation to validate the performance of our controller.

1 Introduction

Network Quality of Service (QoS) has been a difficult target to achieve for quite a while even though newer applications such as video conferencing, VoIP etc. demand performance guarantees. Despite a large volume of work, QoS has not been widely deployed in today's networks. A primary reason for this is the complexity of proposed QoS solutions and largely manual per-device configuration of QoS knobs by network administrators [2]. Such operations are not only prone to human errors leading to critical service disruptions, but can only support coarse-grained QoS to different applications [3].

Therefore, the two most commonly adopted techniques to provide QoS today are physical network isolation and network overprovisioning. Data Center networks, for instance, have dedicated networks with specialized hardware and communication protocols for each class of traffic, e.g., Fiber Channel for storage and Infiniband for High Performance Computing (HPC) traffic. In some cases, networks are highly over-provisioned by a large factor of 6x or even more to avoid QoS violations [8]. However, these solutions not only lead to increased installation costs but also significant increase in management and operational costs. Additionally multiple dedicated networks cannot leverage statistical multiplexing of traffic from different applications leading to poor utilization of available network resources even for best effort traffic.

Network convergence has been recently getting lots of attention as it is highly desirable to serve traffic from multiple applications on a single network fabric (e.g, Ethernet network) for cost reduction and simplified management. With integrated network infrastructure, networks administrators do not have to install and manage multiple fabrics with different protocols and configurations, leading to reduced costs as well as flexibility in deploying the management workforce. There are two dimensions to network convergence. First, convergence for traffic from different applications such as storage, VoIP, VoD etc. onto a single network. Second, convergence of traffic from different tenants/users in a multi-tenancy environment such as Amazon's EC2 offering.

To enable the convergence of multiple services on the same fabric, it is required to create virtual network slices on the fabric where each slice can provide strict performance isolation to the assigned traffic without interfering with traffic from other slices. However, the current state of the art QoS deployments lack fine-grained control and can only support DiffServ-like class-based traffic controls. Furthermore, even the simple actions require complicated manual configurations in the network devices.

In this paper, we present a QoS control framework for automated fine-grained management of converged network fabric. The QoS controller can create network slices to assign different applications traffic to different slices, and provision the slices dynamically to satisfy the performance requirements across all applications. With the QoS controller, what network adminis-

trators only have to do is to specify simple and high level slice specifications for services (or customers), then the controller automatically *"reserves"* network resources to *"preserve"* given performance requirements. To maximize flexibility, the slice specifications can be applied to individual flows, aggregate traffic of certain flows, or even combination of them based on customer's requirements.

Our main contribution is to solve a crucial problem in deploying network QoS - automation of the low level QoS knobs with high level service requirements as input. We propose a new set of QoS APIs and QoS controller for fine-grained automated QoS control in networks having multiple slices. Specifically, we defined QoS APIs as an extension of OpenFlow [13].

This paper is organized into the following sections. In Section 2 we describe the proposed control framework in detail. We introduce our QoS controller implementation and our OpenFlow testbeds in Section 3. We discuss evaluation results in Section 4, survey related work in Section 5, and conclude in Section 6.

2 QoS Control System Description

In this section, we describe the design decisions and each of the components of the QoS controller.

The design goals for our system are the following:

Automated but fine-grained control. The controller should be able to automatically find and apply the best configurations for flows based on given performance requirements at a fine-grain without requiring administrators' manual intervention.

Adaptive to dynamic workloads. Unlike traditional approach of setting class-based static priorities, our controller adapts QoS configurations based on monitored network state for better utilization of network resources.

Deployable in existing networks. In order to provide end-to-end QoS guarantees, the control framework should be deployable on legacy devices and only leverage simple and commonly available QoS knobs.

Support large-scale networks. The controller's computation of resource allocation for various flows as well as QoS control resources such as TCAM entries and rate limiters should scale with network size.

Efficient use of resources. The controller should provide network-wide optimization in resource allocation by utilizing a global view of the network (e.g., admit more flows into networks without QoS violations).

Figure 1 shows the architecture of the QoS control framework we designed. The architecture needs the administrators to simply specify the high level QoS requirements (of the network slices or services) and automates the process of deriving individual per-device configuration specifications and then configuring the switches. When a new flow arrives, the controller first applies the

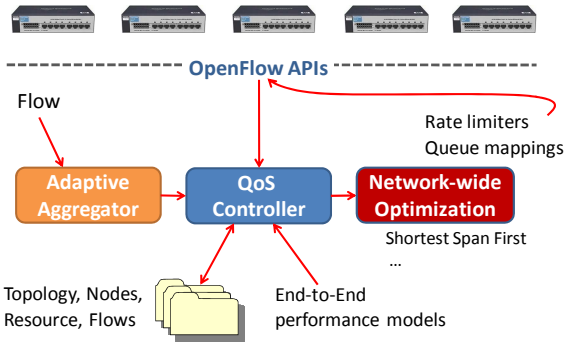

Figure 1: Architecture

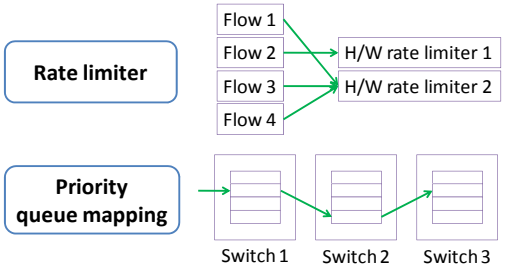

Figure 2: QoS APIs

adaptive aggregator component to the new flow for better scalability, and decides QoS configurations based on the measurement of network states and network-wide optimizations. The resultant configurations are instantiated on the switches through QoS APIs, and resources are then reserved to provide the requested performance to the new flow. The following sections describe more details of each component.

2.1 QoS APIs

In order to automate the configuration and management of available QoS knobs by the controller, we added QoS APIs to OpenFlow. OpenFlow [13] is an open specification that provides a rich set of APIs to enable the control of packet flows in a network device.An OpenFlow switch maintains a flow table, with each entry defining a flow as a certain set of packets by matching on 10 tuple packet information.As shown in Figure 2 the QoS APIs expose the most common existing hardware switch QoS capability, namely rate-limiters and priority queues, to the remote controller.[1] The QoS APIs enable us to attach flows to the rate limiters and priority queues. By using the rate limiter APIs, the controller maps an individual flow or a set of flows to one of the rate limiters to enforce aggregate bandwidth usage Primarily, the rate-limiters are used at the network edge. By using the queue mapping

[1]Recently released OpenFlow v1.0 specification also has mechanisms for network slicing. Though it has more complex queue configurations(not supported by most hardware vendors currently), it does not support the rate-limiter API. Our QoS controller can be extended to leverage the complex queue configuration.

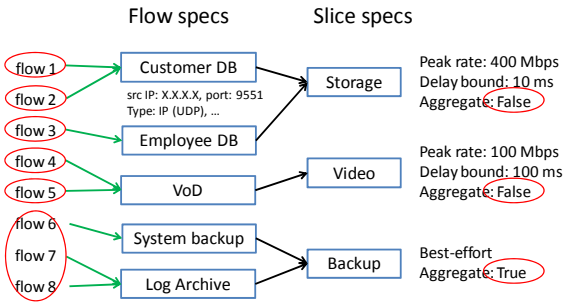

Figure 3: Adaptive Aggregation

API, the controller maps a flow to one of the priority queues in the outgoing port of a switch, thus managing the bandwidth and delay allocation for the flow at each switch.

The controller dynamically manages these mapping across all flows and on a per switch basis, enabling fine grained reactive control. These dynamic mappings are more flexible than the conventional static priority tagging because the controller can decide the mappings based on the current workload at each switch.

When a new flow arrives, according to OpenFlow protocol, the first packet of the flow is delivered to the controller. Based on the policy configurations, the controller determines if the flow is a QoS-sensitive flow and what its performance requirements are. The output of the QoS controller is the setting of rate limiters at the edge switches and priority queues for flow at each path hop. These settings are deployed on the switches via the QoS APIs.

2.2 Adaptive Aggregator

In large-scale networks, it is not feasible for the controller to compute the resource allocation for every single flow and implement QoS actions on switches on a per-flow basis. Otherwise, it would incur large computational and storage overhead on the controller and the switches may end up with a huge amount of flow entries exceeding available TCAM entries and rate-limiters. Hence, the QoS controller implements an adaptive flow aggregator that categorizes individual flows into groups, and allocates resources based on the groups whenever possible. For example, in Figure 3, 8 flows from 5 services are grouped into 3 network slices based on slice specification. We note that not every service requires per-flow isolation. Instead, some services require only aggregate resource reservation for their traffic, which has been traditionally provided by a separate network for the service.

In Figure 3, a *Flow Spec* represents a set of flows for each service in network. Like the OpenFlow specification, the spec is given as a set of header fields with wildcard fields to define the scope of flows belonging to the slice. *Slice Spec* shows performance requirement for a network slice such as maximum bandwidth, minimum delay, etc..

In addition to the performance requirements, the slice spec also has an *aggregate* marker field, and if it is true, the controller reserves resources for the aggregate flows in the slice. For example, the controller will configure QoS knobs for all flows from the *System backup* and *Log archive* services. Once the configuration is done for the first flow from the services, all the following flows will share network resource without contacting the controller.

2.3 Network State Information Management

The controller builds and maintains information about the current network state and uses this information base for deciding QoS configurations. The network state information includes network topology, active flows, performance requirements, and available resources in each switch. It is important to keep this information up-to-date with the current state of the network because inconsistency can lead to under-utilization of network resources as well as performance violations.

The controller combines passive and active monitoring on networks to dynamically update the network state. For passive monitoring, the controller uses packets forwarded to the controller. For example, the controller intercepts LLDP and DHCP packets exchanged in networks, and updates the network topology accordingly. The controller updates the available resource database when it adds or deletes QoS configurations in switches. However, there is always a possibility of inconsistency due to unexpected network events. We leverage the OpenFlow protocol and APIs to enable the controller to query flow tables from switches to periodically check and fix any inconsistency. We also extended the OpenFlow APIs to query QoS configurations in switches from the controller.

2.4 Performance Model

The controller should be able to estimate the performance that a flow will experience in network. We devise an end-to-end performance model that provides the worst case bandwidth and delay for a flow with a given QoS configuration. Based on this model the controller ensures that the new flow will receive its requested performance and the addition of the new flow will not cause performance violation for existing flows in the network.

We developed our performance models for our QoS APIs based on the commonly available QoS knobs: rate limiters and static priority queues. We use a modified version of RCSP queuing discipline [14] because the rate limiter in our QoS APIs is different from the traffic policer in RCSP.

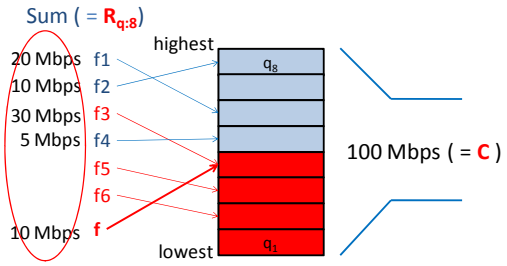

Figure 4: Delay model

Regarding performance requirement of a flow f, we consider bandwidth and delay requirement denoted by (r_f, d_f) respectively. Bandwidth requirement is relatively simple to satisfy than the delay requirement. To guarantee f receives the requested bandwidth r_f, we can simply check if the max rate of traffic from the other flows in the shared links is under $(C - r_f)$ where C is a link capacity. In our framework, the controller installs a rate limiter in the ingress edge switch of the flow to limit the amount of traffic injected into the network from the flow source.

However, the end-to-end delay requirement is more complicated to model because we should consider the dependent behavior of priority queues. Figure 4 represents a switch output port with 100 Mbps capacity. Here, we assume that packets are queued and dropped at output ports. The port has 8 static priority queues q_i, and 6 flows (f_1, f_2, ..., f_6) are mapped to one of the queues. Now assume that a new flow f is mapped to the fifth highest priority queue q_5. Then the per-hop delay of f, D_f, is determined by 1) how fast packets arrive at queues having higher or equal priorities (from q_5 to q_8) and 2) how fast the link can drain the packets.

Therefore, the simple equation for D_f is defined as $f(q, R_{q:8}, C)$ where q is the priority queue for f (in this example $q = 5$), and $R_{q:8}$ is the sum of max rates of flows between q_q and q_8.

2.5 Network-wide optimization

The important implication of the delay bound model is that flows in the same port can impact each other in terms of delay bound. For example, in Figure 4, f's per-hop delay is affected by f_1, f_2, f_3 and f_4, and f increases per-hop delay (and end-to-end delay) for f_3, f_5, and f_6 at the same time. As a result of adding f, some flows now have higher end-to-end delay, so the controller might not be able to find a queue for f in the next hop if one of the affected flows passes the same hop and its end-to-end delay exceeds the delay requirement. So, the controller should consider both the interactions between flows in the same hop and the interactions with flows in the remaining hops to decide queue assignment for a new flow. However, as there can be a large number of flows over multiple hops in practice, it is computationally expensive to find the

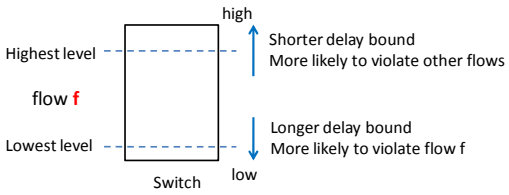

Figure 5: Highest level and Lowest level

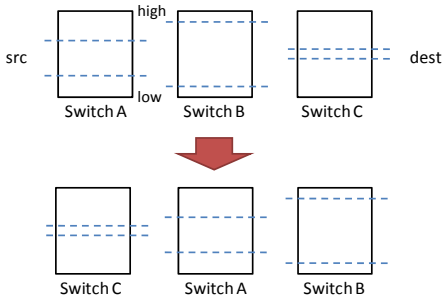

Figure 6: Shortest Span First (SSF)

optimal queue assignment that satisfies f's delay requirement while not violating the requirements of the existing flows. It is important to reduce delay in processing new flows for faster flow setup time.

We develop a range of techniques from heuristics to optimization modeling, and introduce SSF (Shortest Span First) in this paper. The goal is to maximize the probability of satisfying a new flow's performance requirement while minimizing the number of rejected flows. The output is queue assignments in every output port that f passes. The intuition behind SSF is that we first pay more attention to a port with less available options to avoid rejection of f while trying to find the best delay bound for f. Given an output port, we can choose a queue for f from q_1 to q_8 (q_8 is the highest priority queue). If we put f into q_8, f will get the shortest per-hop delay bound, but negatively affect all the existing flows in the port. Likewise, f will affect a small number of flows but will get high delay bound if it is put into q_1. In each port, we compute *highest level* for f, the highest possible priority queue not violating existing flows, and *lowest level* for f, the lowest possible priority queue not violating f (see Figure 5). We denote these two priority queue levels by $high_f$ and low_f respectively. Then, the span between two levels, $high_f - low_f$, represent the available options for f in the port. If we assign f into a queue in the first port, then the spans for the other ports will shrink because of the new constraint, and f should be rejected if a port has zero span. Therefore, in SSF, the controller sorts ports by the computed spans, assign f to the highest level in the first port, and recompute spans for the remaining ports. The controller repeats the process until there is no remaining port for f (see Figure 6).

Unlike static tagging mechanisms [3, 10], the controller can map f to different priority queues at each hop

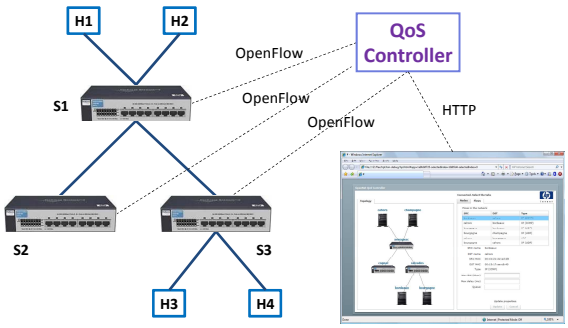

Figure 7: QoS Controller Prototype

Flow name	Route	Rate
Customer DB	H3-S3(8)-S1(8)-H1	400 Mbps
Employee DB	H4-S3(8)-S1(8)-H2	300 Mbps
VoD	H3-S3(7)-S1(7)-H1	100 Mbps
Systems Backup	H4-S3(1)-S1(1)-H2	bursty

Table 1: Generated flows in testbeds

depending on the current workloads on the path. For example, if a switch port has many delay-sensitive flows, SSF will map f to a low priority queue to avoid violating the existing delay-sensitive flows. It can lead to longer per-hop delay for f, however, the controller can increase f's priority in the other switches on the path in order to satisfy the end-to-end delay requirement of f.

3 Implementation

We prototyped QoS controller using two kinds of OpenFlow-enabled switches, a HP ProCurve 5406zl hardware switch and also a Linux software switch based on Open vSwitch [11]. Our QoS controller is implemented on top of the NOX platform [7], an open-source OpenFlow controller. QoS controller has TCP connections to the switches in our testbeds, and communicates with the switches through standard OpenFlow APIs to collect information about network states such as topology (Figure 7). When a new flow arrives in the network, the controller calculates resource allocation based on the database and performance models. Then, the controller installs rate limiter in the flow's edge switch, and configures priority queues in switches on the flow's path. Lastly, the controller provides a web interface to network administrators. The interface provides the current states of flows in network, interface to controller operations, and statistical information about flows on run-time.

4 Evaluation

For evaluation, we use three ProCurve 5406zl switches that implement our QoS extension APIs as well as OpenFlow platform (Figure 7). We generate contending flows from the four hosts (H1, H2, H3, and H4) connected to

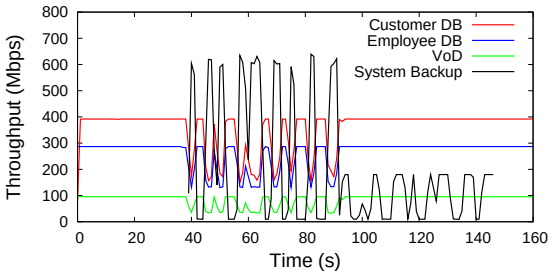

Figure 8: Throughput with UDP cross traffic

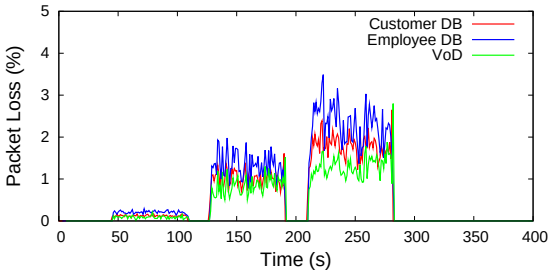

Figure 9: Packet loss with TCP cross traffic

the switches via 1G links and observe how the QoS controller affects the performance of contending flows.

To emulate multiple services in real networks, we customized iperf network testing tool and generated four test flows as shown in Table 1. The performance requirement of each flow is shown in Figure 3. The first three flows act as QoS flows with guaranteed performance requirements. The System Backup flow is a best-effort cross traffic. To observe the direct effect of QoS control on the flow performance, we used UDP for the three QoS test flows while both UDP and TCP are used to generate the cross traffic.

In the first set of experiments, we started Customer DB, Employee DB and VoD (Video on Demand) flows at time zero while the System Backup flow started at $40s$. Figure 8 shows the time-varying throughput of each flow. Because the System Backup flow is bursty UDP, it causes throughput fluctuations of other flows. We turned on the QoS controller at $90s$ and then each QoS flow gets its requested bandwidth while the System Backup flow is limited not to overload the 1G bottleneck link capacity by the controller. Based on the derived performance models and the given performance requirements, the controller automatically decides the best priority queues for each flow. The two DB flows are assigned to the highest priority queue q_8 while the VoD flow is assigned to a lower priority queue q_7 because it has higher delay bound. Since the System Backup flow requires best-effort service without any guaranteed performance, the controller maps the flow to the lowest priority queue q_1.

Unlike UDP flows, TCP flows should have less impact on other flows because of TCP congestion control mech-

anisms. To see the effect of QoS control with TCP cross traffic, we generated a TCP flow that has the same route as the System Backup flow, and measured the throughput and packet losses of the three QoS flows. In Figure 9, a single TCP cross flow starts at $45s$ and ends at $110s$. According to TCP congestion control, a TCP flow always tries to increase its congestion window, so it also causes packet losses of the QoS flows even though its impact is much smaller than the case with UDP cross traffic. However, as we generate more TCP cross flows, the aggregate effect of the TCP flows on the QoS flows increases. In Figure 9, we generated 10 and 30 parallel TCP flows during the time windows of [$127s$,$190s$] and [$210s$, $400s$], respectively; the QoS flows suffer from more packet losses accordingly. As we turn on the QoS controller at $280s$, the three QoS flows suffer no packet loss and it clearly indicates the QoS controller is needed even with congestion-controlled TCP cross flows. The result implies that we need fine-grained QoS control even when most traffic in network is TCP.

The QoS controller incurs certain processing and communication overheads to run the admission control and queue assignment algorithms and to implement rate limiter and queue mapping configurations on switches. To observe the controller induced overhead, we measured the additional delay of the first packets of the newly arrived flows. The result shows that the controller makes average $11.7ms$ additional delay for the first packets, but there is no change on the following packets.

5 Related Work

The QoS mechanisms proposed in the past have either failed to be deployed [4] or lack fine-grained adaptive control [3]. RSVP-TE [1] allows resource reservation along a flow path. Similarly MPLS can be used to setup tunnels with QoS guarantees. But an automated controller with global view could improve network resource utilization further. FlowVisor [12] enables network slicing by providing virtualized views of network resources, and can be used with QoS controller to provide more strict isolation between network slices in congestion. QoS Policy manager from Cisco [5] alleviates the problems of manual configuration by automation but is limited by the lack of flow-management flexibility in today's non-programmable network devices. Recently there have been several proposals for network convergence [6, 9]. Though these protocols provide better QoS control, that we plan to leverage in future, they still suffer from the limited number of classes as well as the lack of automated control for adaptive flow aggregation Our approach provides automated scalable QoS - by disaggregating to the level of sub-flows, or to aggregate flows into classes as required.

6 Conclusion

In this paper, we propose a QoS control framework for converged network fabrics. It aims at solving the crucial problem in deploying QoS - that of automating the process of setting low level QoS knobs from high level descriptions of application/service requirements. In addition, by utilizing a global view of the network at the controller, we implement network-wide optimizations in the controller for better utilization of available resource in a converged network fabric.

Our ongoing work includes a evaluation of the optimization algorithms by simulation. Based on the current prototype implementation, we also plan to extend the QoS controller algorithms for legacy network device awareness. This will allow true end-to-end QoS in a heterogeneous network environment. Automatic detection of the different application flows and their performance requirements will further alleviate the manual configuration errors.

References

[1] D. Awduche, L. Berger, D. Gan, T. Li, V. Srinivasan, and G. Swallow. RFC 3209: Rsvp-te: Extensions to rsvp for lsp tunnels, 2001.

[2] G. Bell. Failure to thrive: Qos and the culture of operational networking. In *Proceedings of the ACM SIGCOMM workshop on Revisiting IP QoS*, 2003.

[3] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. RFC 2475: An architecture for differentiated services, 1998.

[4] R. Braden, D. Clark, and S. Shenker. RFC 1633: Integrated services in the Internet architecture: an overview, 1994.

[5] CiscoWorks QoS Policy Manager. `http://www.cisco.com/en/US/products/sw/cscowork/ps2064/index.html`.

[6] FCoE (Fibre Channel over Ethernet). `http://www.fcoe.com/`.

[7] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. Nox: towards an operating system for networks. *Computer Communication Review*, 38(3):105–110, 2008.

[8] Y. Huang and R. A. Guèrin. Does over-provisioning become more or less efficient as networks grow larger? In *Proceedings of the 13th IEEE International Conference on Network Protocols (ICNP)*, 2005.

[9] IEEE 802.1 Data Center Bridging (DCB) Task Group.

[10] IEEE 802.1p Traffic Class Expediting and Dynamic Multicast Filtering.

[11] Open vSwitch: An Open Virtual Switch. `http://openvswitch.org`.

[12] R. Sherwood, G. Gibby, K.-K. Yapy, G. Appenzellery, M. Casado, N. McKeowny, and G. Parulkary. Flowvisor: A network virtualization layer. Technical Report TR-2009-01, OPENFLOW, 2009.

[13] The OpenFlow Switch Consortium. `http://www.openflowswitch.org`.

[14] H. Zhang and D. Ferrari. Rate-controlled static-priority queueing. In *Proceedings of the IEEE INFOCOM*, 1993.