Providing Bandwidth Guarantees with OpenFlow

Hedi Krishna, Niels L. M. van Adrichem, and Fernando A. Kuipers
Network Architectures and Services, Delft University of Technology
Mekelweg 4, 2628 CD Delft, The Netherlands
{HediKrishna@student., N.L.M.vanAdrichem@, F.A.Kuipers@}tudelft.nl

***Abstract*—Quality of Service (QoS) control is an important concept in computer networking, as it is related to end-user experience. While providing QoS guarantees over the Internet has long been deemed too complicated, the emergence of Software-Defined Networking (SDN), and OpenFlow as its most popular standard, may facilitate QoS control.**

In this paper, we consider how to enable bandwidth guarantees with OpenFlow. Our design allows QoS flows to send more than their guaranteed rates, as long as they do not hinder other guaranteed and/or best-effort flows.

Furthermore, our design uses OpenFlow's meter table to aggregate traffic. Our traffic aggregation functionality only adds overhead to the first switch, but no other complexity is incurred at the subsequent switches.

I. Introduction

Quality of Service (QoS) control typically refers to mechanisms used in a network to guarantee a certain service from the network. By using QoS control, and the related notion of traffic engineering, network administrators are able to manage their resources more efficiently and can offer tailor-made service, without having to over-provision the network. QoS is of particular importance to applications that, in order for them to function properly, need specific network guarantees. For example, applications like voice conversation and video streaming require small delay and jitter. On the other hand, data communication is less sensitive to delay and jitter, but more sensitive to packet loss. Despite its importance, the adoption of QoS control in the Internet has been slow. Guaranteed QoS, such as Integrated Service (IntServ) [1] is deemed too complex and not scalable. On the other hand, Differentiated Service (DiffServ), [2] with its aggregation model, is less complex, but does not provide strong QoS guarantees. Presently, service providers opt to over-provision their network, even though it is costly and inefficient, simply because it is less complicated than deploying QoS control.

Software Defined Networking (SDN), as a new paradigm in networking, offers the opportunity to speed-up the adoption of QoS control in the Internet. The centralized nature of SDN is expected to reduce the complexity that is commonly associated with QoS guarantees. With its backing by leading companies in the tech industry, SDN is well on its way to become a de-facto standard. Hence, by having QoS control included in SDN, the future Internet might have native QoS support.

In this paper, we will demonstrate how to guarantee bandwidth with OpenFlow [3], a popular SDN protocol. In Sec. II we discuss the QoS features of OpenFlow, in Sec. III, we will present our framework for guaranteeing bandwidth, in Sec. IV we test our proof-of-concept implementation, and we conclude in Sec. V.

II. QoS control in OpenFlow

QoS in OpenFlow is supported by two features, namely the queue and the meter table. A queue is an egress packet queuing mechanism in the OpenFlow switch port. The queue was first supported in OpenFlow 1.0 with only a guaranteed minimum rate property. Later, it was extended in OpenFlow 1.2 with a maximum rate, which limits the maximum throughput of a queue. Although it is specified in the OpenFlow switch specification, the OpenFlow protocol does not handle queue management. Queue management (creation, deletion, alteration) is handled by the switch configuration protocol, such as OF-Config or Open vSwitch Database (OVSDB). OpenFlow itself is only able to query queue statistics from the switch.

The meter table is a feature introduced in OpenFlow 1.3. Metering allows ingress rate monitoring of a flow and to perform operations based on the rate of the flow. There are two operations that can be performed: dropping packets and remarking DSCP bits of the packets. Unlike a queue, which is a property of a switch port, a meter is attached to flow entries.

In [4], the authors use OpenFlow queues to provide bandwidth guarantees. For each QoS flow, a queue is created in the ingress switch and intermediate switches with `min-rate` and `max-rate` equal to the guaranteed bandwidth. While correct, the system does not scale, since the number of queues grows with the number of flows. In [5], aggregation is proposed. Instead of using an exclusive queue for every QoS flow, only one queue is used to forward all QoS flows in a switch port. The rate of the queue itself is dynamic, equal to the sum of all QoS flows forwarded via the switch port. However, the maximum rate of the QoS flows is limited to their guaranteed rate. If a QoS flow violates this limit, it might contend for bandwidth with other QoS flows.

III. Bandwidth guarantees

To address the scalability and utilization problems of [4] and [5], we have designed an OpenFlow controller with Ryu [6]. Our QoS framework contains several elements.

A. Traffic prioritization

We allow for two types of flows: (1) QoS flow and (2) best-effort flows. QoS flows have a minimum guaranteed bandwidth, while best-effort flows do not have such guarantees. The QoS flows are allowed to send more than their

978-1-5090-4361-3/16/$31.00 2016 IEEE

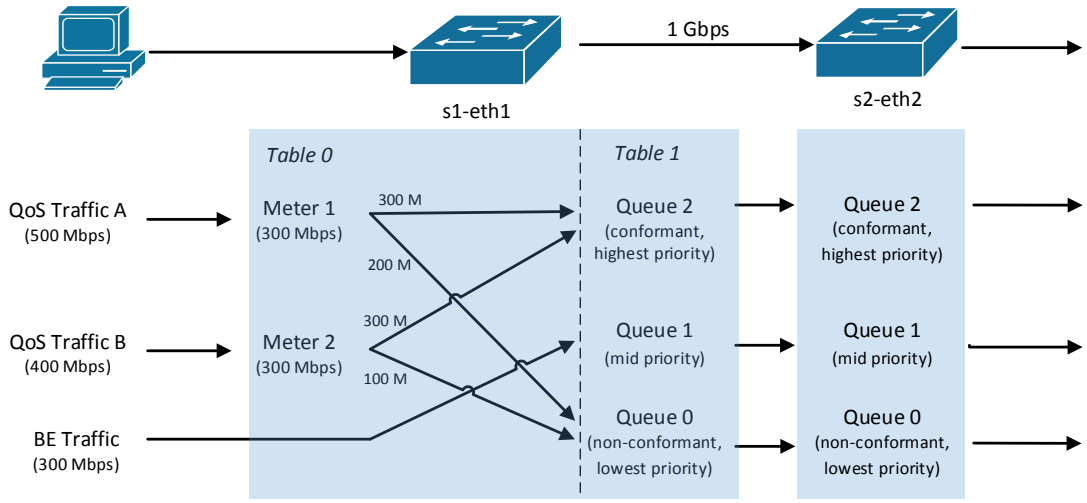


Fig. 1. Metering-aggregation concept

guaranteed rates. Conformant QoS flows are QoS flows that stay within their guaranteed rates, while non-conformant QoS traffic exceeds the guaranteed rate.

In IntServ Guaranteed Service [7], non-conformant QoS packets are treated as best-effort packets. In our model, non-conformant traffic is given lower priority than best-effort traffic. In case of congestion, the switch will drop the non-conformant packets before dropping best-effort packets. One could opt to discard this option and treat non-conformant packets as best-effort, by simply marking the packets differently.

B. Guaranteeing bandwidth in controller and switch

The system guarantees bandwidth for QoS flows at two levels. First, the controller conducts an admission test for new QoS flows. In the admission process, the controller tries to route the QoS flow via a path that can accommodate the required rate. If such a path is not available, the flow will be dropped. Best-effort flows skip the admission procedure and will always use the shortest path between the end hosts. The admission process ensures that admitted QoS flows will not contend for bandwidth with each other.

The controller keeps track of how much bandwidth is allocated in the network. Using this information, the controller decides where to route the path. In our controller, we implement the Widest Shortest Path algorithm [8], though more extensive monitoring and routing algorithms, such as OpenNetMon [11] and SAMCRA [10] may be applied.

The second level of guaranteeing bandwidth is at the switch. OpenFlow queue is used to give priority to certain traffic. In each switch port, three queues are used. Queue 2 has highest priority and forwards conformant QoS traffic. Queue 0 has lowest priority and forwards non-conformant QoS traffic. Best-effort traffic uses queue 1.

Bandwidth reservation happens through the controller, while at the switch level, available bandwidth can be used by any type of traffic. By allowing non-conformant QoS flows and best-effort traffic to use the available bandwidth, the overall throughput can be maximized.

C. Traffic aggregation

The metering operation *dscp_remark* allows the switch to split flows by altering the DSCP bits of non-conformant packets. Each QoS flow is passed through a meter table entry at its ingress switch with the meter's rate equal to the guaranteed rate. A QoS flow with a rate exceeding the meter rate will have its excess packets' DSCP bits remarked.

Conformant QoS flows are aggregated and forwarded via a single queue. Aggregation is also performed for non-conformant QoS and best-effort traffic. Conformant and non-conformant packets are distinguished through their distinct DSCP bits, on whose values hop-by-hop forwarding also aggregates.

We take advantage of the multi-table pipeline processing in OpenFlow, which allows the switch to process the packets multiple times in a sequential order. In this case, the metering is performed in flow *Table 0*. Then, flow *Table 1* matches DSCP bits and forwards the packets to the aggregation queue. With aggregation, every switch port will only need three queues: one for conformant QoS, one for excess QoS, and one for best-effort traffic. Figure 1 illustrates the concept.

D. Hard state reservation

One of the drawbacks of IntServ is that it uses RSVP as its resource reservation protocol [9]. RSVP is a soft-state reservation that requires periodic exchange (every 30 s) of refresh messages between end hosts, which leads to significant overhead. If these messages are not received, the end hosts will assume that the connection has ended.

In our model, we use hard-state reservation. The QoS reservation begins when there is a QoS flow request to the con-

TABLE I
TRAFFIC GENERATED FOR ADMISSION TEST

Host pairs	Type	Requested bandwidth	Actual rate
h1 - h5	QoS	70 Mbps	70 Mbps
h2 - h6	QoS	70 Mbps	70 Mbps
h3 - h7	QoS	70 Mbps	20 Mbps
h4 - h8	BE	N/A	20 Mbps

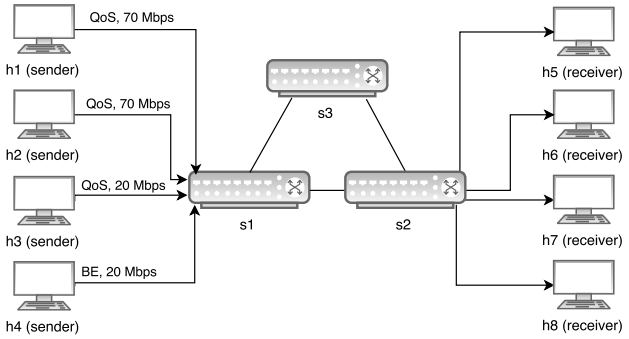


Fig. 2. Topology for the admission test

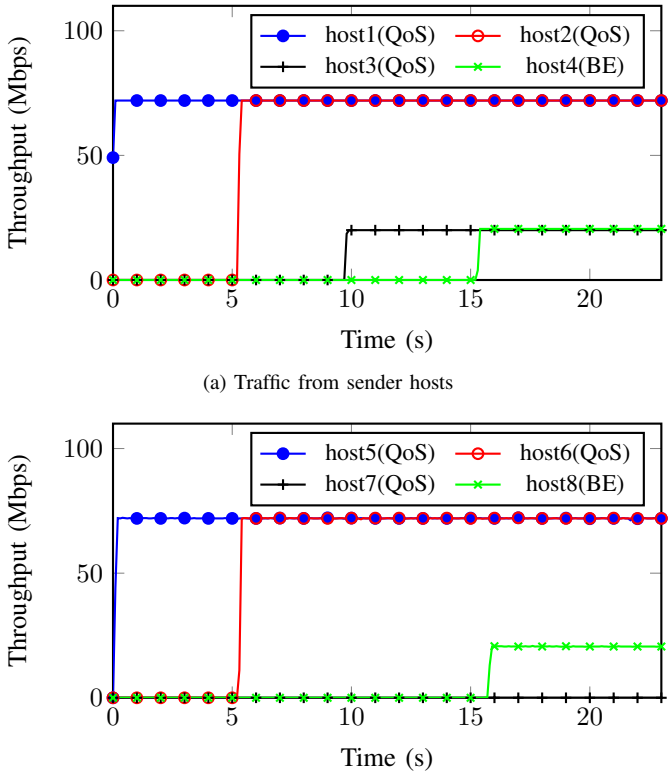
troller. The request is signaled with the `OFPT_PACKET_IN` message with appropriate DSCP bits. The resources are freed from reservation when the flow entry is removed from the switches, signaled by the `OFPT_FLOW_REMOVED` message. Flow removal messages are sent by switches to the controller after a flow reaches its lifetime limit, defined by `idle_timeout` or `hard_timeout`. Both `OFPT_PACKET_IN` and `OFPT_FLOW_REMOVED` are standard OpenFlow messages. There is no extra signaling exchanged between the switch and the controller.

IV. PROOF OF CONCEPT

The fundamental aspects of our model, the admission-reservation process and traffic prioritization are demonstrated using two experiments. The experiments are carried out on two different testbeds. The admission control experiment is conducted on our own testbed consisting of Open vSwitch (OVS) 2.3.2 switches, supporting OpenFlow 1.3. Each switch runs on a server with Quad Intel(R) Xeon(TM) CPU 3.00GHz processor and 4 GB memory. Unfortunately, the software implementation of OVS does not support meter table. For the prioritization experiment, which has to use meter table for aggregation, we therefore turn to hardware switches of the Dutch SURFnet SDN testbed, which do provide meter functionality. The SURFnet testbed uses Pica8 P5101 switches, running PicOS 2.7.1 with OpenFlow 1.3.

A. Admission control

Our topology is depicted in Figure 2. The network consists of three switches in a ring topology and eight hosts, with both switch s1 and switch s2 connected to four hosts. The path via s3 provides an alternative route between s1 and s2. Each host in s1 is paired with a host in s2 for traffic generation. The bandwidth of the links is 100 Mbps.



(a) Traffic from sender hosts

(b) Traffic captured at receiver hosts

Fig. 3. Admission-reservation test result

QoS flows request a minimum bandwidth of 70 Mbps. Best-effort and QoS packets are differentiated by their DSCP bits. A flow is identified by a matching field of the tuple <IP source address, IP destination address, IP DSCP bits>. Three pairs of hosts (h1-h5, h2-h6, h3-h7) exchange QoS traffic, while h4-h8 exchanges best-effort traffic.

The traffic used in the experiment is UDP, generated with iperf. Table I shows the actual rate of traffic generated by sender hosts.

Figure 3 shows the results of the experiment. The first graph is the rate of traffic generated by the sender hosts. The second graph reflects the throughput captured at the receiver hosts.

The experiment begins at time $t = 0$ when host h1 sends 70 Mbps QoS traffic to h5. Since both paths have the same amount of available bandwidth, this flow is routed using the shortest path (link s1-s2). The controller reserves 70 Mbps in this link for flow h1-h5, and updates the link weight. The new weight is 30 Mbps, equal to the currently available bandwidth. At time $t = 5$, host h2 starts a QoS flow to h6. Link s1-s2 no longer has enough available bandwidth to accommodate the new QoS flow, because it is less than 70 Mbps. Thus, flow h2-h6 is routed through the alternative path (s1-s3-s2). Host h3 starts QoS flow to h7 at time $t = 10$. The actual throughput of this flow may be 20 Mbps, however, as QoS flow, it requests 70 Mbps bandwidth, which exceeds both paths' availability. Because no path between sender and receiver can satisfy this request, the flow is blocked by admission control. Switch s1

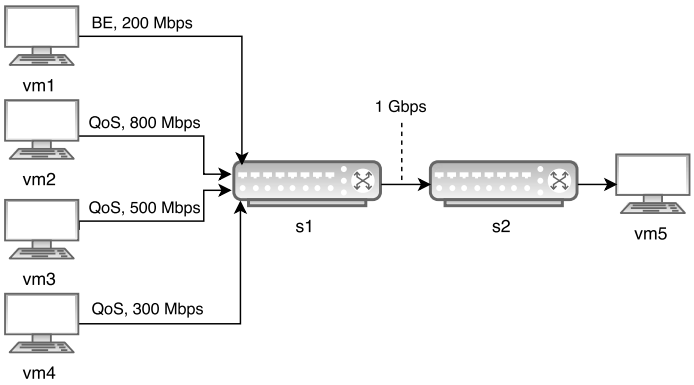
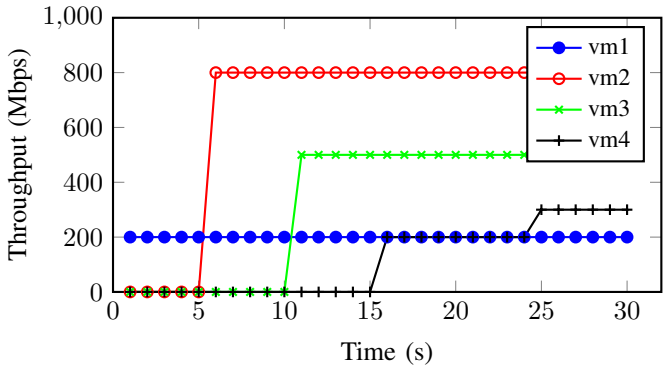
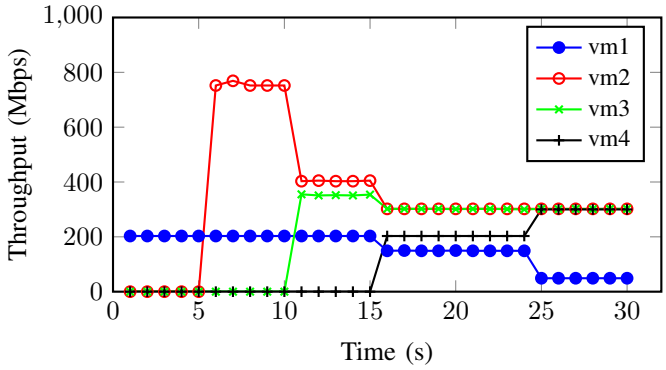


Fig. 4. Topology for the aggregation and prioritization test



(a) Data rate generated at sender vm hosts



(b) Actual throughput at receiving vm hosts

Fig. 5. Aggregation-prioritization test result

installs a flow rule to drop subsequent packets that match this source-destination-DSCP tuple.

On the contrary, a 20 Mbps best-effort flow from h4 to h8 (starting at $t = 15$) is delivered via link s1-s2 (shortest path). This flow is allowed, because a best-effort flow does not have specific bandwidth requirements. The available bandwidth in this path is 30 Mbps, which is more than this flow's throughput. Therefore, the traffic is received at h8 at the full rate of 20 Mbps. This flow is routed via the shortest path s1-s2.

From the experiment, we see that the admission process prevents QoS flows from competing with each other. On the other hand, best-effort flows bypass the admission process and excess QoS traffic is also allowed; thus, maximizing bandwidth utilization. Nevertheless, to ensure that this traffic does not disturb conformant QoS traffic, traffic prioritization at the switch level is needed.

B. Traffic aggregation and prioritization

The aggregation concept in our model requires pipeline processing of meter and queue. The pipeline processing of packets with meter-remarked DSCP values requires that this rewritten header value is evaluated in the match filter of the next flow table. However, we found that the used switches incorrectly match on the original packet headers in subsequent flow tables and defer packet rewriting until the packets exit the switch, a problem introduced by mapping OpenFlow functions to the fixed-functionality ASIC implemented by the manufacturer. To replicate the necessary functionality, we created a self-loop at the ingress switch to simulate packet rewriting between flow tables. The switch first processes the QoS flow with meter table and forwards it to the self-loop, the packets then re-enter the ingress switch, and this time they are forwarded via queues.

Figure 4 shows the network used in our experiment. QoS flows are generated from vm2, vm3 and vm4, while vm1 generates a best-effort flow. All QoS flows have a guaranteed rate of 300 Mbps. The packets are UDP, generated with iperf. The link between the switches has a capacity of 1 Gbps (the effective usable data rate was a bit lower).

Figure 5 shows our experiment results.

1) **0s to 5s**. At time $t = 0$, vm1 starts a best-effort flow to vm5 with a rate of 200 Mbps. At vm5, the throughput measures 200 Mbps.
2) **5s to 10s**. vm2 starts its QoS flow with a throughput of 800 Mbps. Although only 300 Mbps are guaranteed, it is allowed to send more. The excess traffic is served as long as link bandwidth is unused. At $t = 5$, there is exactly 800 Mbps free bandwidth, and vm5 receives this flow with a throughput of 800 Mbps.
3) **10s to 15s**. vm3 sends a QoS flow of 500 Mbps. The bandwidth of the links is now insufficient to accommodate all flows at their full rate. The total conformant traffic at the moment is 600 Mbps. This traffic is forwarded via queue 2, which has highest priority, making sure that the guaranteed rate is satisfied. The bandwidth is then offered to the best-effort traffic of vm1, which uses 200 Mbps. The remaining 200 Mbps bandwidth is used by non-conformant QoS traffic. Since the excess traffic from vm2 is higher than that of vm3, more packets from vm2 are passed through the queue, resulting in a higher received rate at vm5.
4) **15s to 25s**. At $t = 15$, 400 Mbps bandwidth is available (not reserved by QoS flows). However, the links are fully used by vm1, vm2 and vm3. When QoS flow from vm4 starts (200 Mbps) at $t = 15$, the switch drops all the non-conformant packets from vm2 and vm3 to accommodate

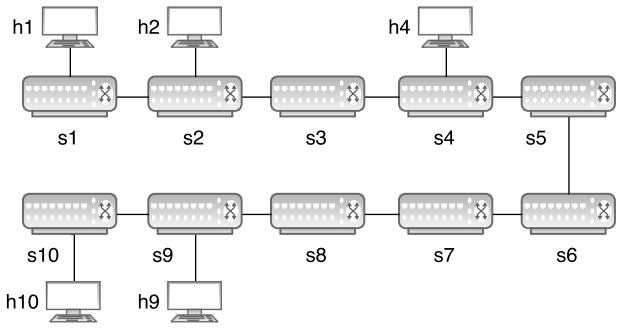


Fig. 6. Topology w.r.t. the bandwidth borrowing test

the newly arrived QoS flow. Since vm2 and vm3 now only have their conformant traffic forwarded, their flows receive 300 Mbps each. Albeit at a slightly reduced rate, the best-effort flow from vm1 is still forwarded by the switch.

5) **25s to 35s**. At $t = 25$, vm4 increases its data rate from 200 Mbps to 300 Mbps. The best-effort flow of vm1 is dropped in favor of the three QoS flows, which now each receive 300 Mbps, equal to their guaranteed rates.

From the experiment, we observe that the bandwidth guarantees still hold when the traffic is aggregated. The aggregation itself significantly reduces the number of queues needed in the switches. No matter how many flows are present in the network, all switches will only need three queues. Other than the flow entries, there are no additional flow states stored in the intermediate switches. The flow entries themselves form an absolute OpenFlow requirement that exists in any OpenFlow application. Additional complexity is only present in the first switch in the form of meter table and pipeline processing, which have to be added for every individual QoS flow.

C. Responsiveness

It is important that the bandwidth "borrowed" by non-conformant and best-effort traffic is returned whenever conformant traffic requires it. Failing to do so might cause packet loss for conformant traffic. In the following experiment, we investigate any adverse effect due to bandwidth borrowing. The experiment is conducted on our own testbed of Open vSwitch 2.3.2 switches with OpenFlow 1.3.

A linear network with 10 hosts, shown in Figure 6 is used.

QoS flows are generated from host h2 to host h4 and h9. Flow h2-h4 has 2 hops on its path, while flow h2-h9 has 7 hops. Both of these flows have a guaranteed bandwidth of 30 Mbps. The actual traffic sent is 30 Mbps, equal to the guaranteed bandwidth. For each QoS flow, 5 MBytes of data are sent from sender to receiver. Flows h2-h4 and h2-h9 are generated alternately, one flow at a time, and repeated 1000 times. There is a 5-second interval between flows to make sure flow entries from the previous flow are already removed. Table II summarizes the traffic.

We consider several scenarios. In the first scenario, a best-effort flow is generated between two end nodes, h1 and h10,

TABLE II
TRAFFIC GENERATED FOR BANDWIDTH TEST

Host pairs	Traffic type	Guaranteed rate	Actual rate	Lifetime
h1-h10	Best-effort	N/A	100 Mbps	Throughout the experiment
h2-h4	QoS	30 Mbps	30 Mbps	5 MBytes data
h2-h9	QoS	30 Mbps	30 Mbps	5 Mbytes data

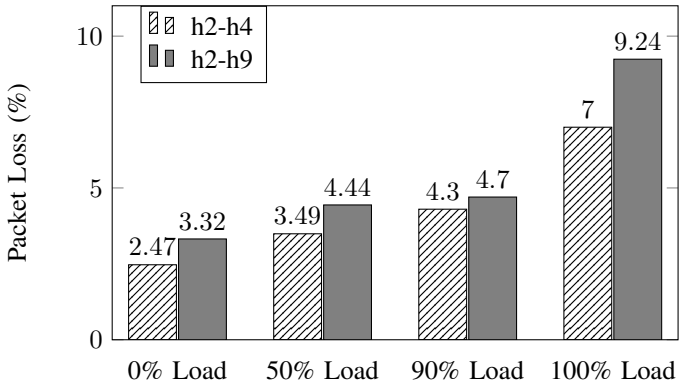


Fig. 7. Packet loss in traffic shaping

before the QoS flows start. The rate of this best-effort flow is 100 Mbps, equal to the capacity of the links. This flow is long-lived. Since the QoS flows use a subset of the best-effort flow's path, we expect to see some disturbance in the QoS flows. We want to see how fast the "borrowed" bandwidth by the best-effort flow is "returned" to the QoS flows, and how much it affects their performance. For comparison, in the other scenarios, the best-effort flow uses 90%, 50% and 0% of the capacity of the links.

Both best-effort and QoS flows are generated using iperf. The measurement is performed at the receiver using iperf's own statistics report. Figure 7 shows our results.

For the scenario with no initial network load (0% load), the QoS flow is treated normally with relatively low packet loss. The loss occurs because of the time needed to install flow entries in the switches.

During the flow entry installation process, the arriving packets are buffered in the OpenFlow switch's buffer. These packets are waiting for this process to finish, because the switch does not know how to process this packet. The switch's buffer has a limited memory. Because the QoS flow rate is 30 Mbps, which is quite high, the switch's buffer cannot accommodate all newly arriving packets. When the buffer is full, some packets are dropped. Flows with more hops need to install flow entries in more switches. Consequently, the switches drop more packets and the packet loss is higher.

Open vSwitch uses the Linux Hierarchical Token Bucket (HTB) as underlying mechanism for the OpenFlow queue implementation. In the token bucket filter algorithm, packets are forwarded using tokens. If no tokens are available, packets are queued up to the queue size. Tokens are generated at a rate

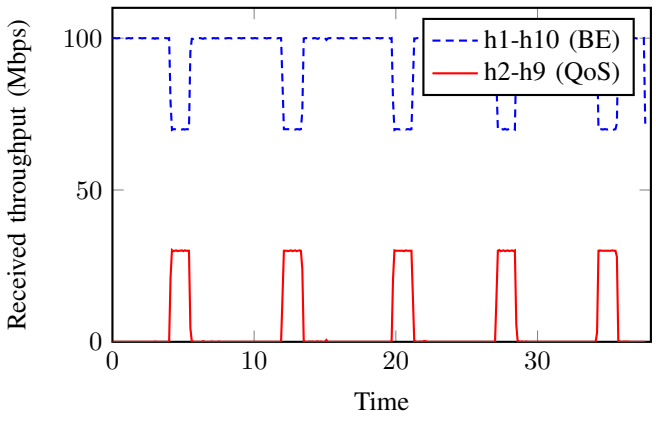


Fig. 8. Received throughput of long-lived best-effort and on-off QoS flow

that corresponds to the configured rate; in our case 100 Mbps. The burst size is set to 125 kbytes to accommodate the bursty traffic from sender hosts.

In the scenario with no best-effort traffic (0% load), all the generated tokens are conserved, as no other packets use the HTB instance. As soon as the flow entries are installed, the packets are dequeued from the buffer and forwarded through the queue specified in the flow entry. The HTB class has accumulated enough tokens in HTB to send the queued packets with minimal loss.

For the scenario with 50% and 90% occupancy, the tokens are still generated and reserved. But because some of the tokens are used by the existing best-effort packets, the conserved tokens are not as abundant as in the previous scenario. After the flow entry is installed, the packets are moved from the switch's buffer to the queue. Since the queue size is limited, and there are insufficient tokens to dequeue all the packets at once, some packets are dropped resulting in higher packet loss.

For the scenario with 100% link occupancy, no tokens are conserved, as the token generation rate is equal to the usage rate. All the packets are moved to the queue after the flow entry is installed. None of the packets are forwarded to the output port instantly; they have to wait until new tokens are generated. The queue becomes full faster than in the previous scenarios, resulting in more packet loss.

In our experiments, packet loss during the installation of flow entries only occurred in the first few milliseconds ($<$ 10 ms for 9 hops). In our experiment, a single QoS flow only sends 5 MBytes of data. Its lifetime is very short, around 1.5 seconds. Iperf statistics (with a time resolution of 500 ms) show that high packet loss only occurs in the first 0.5 seconds.

For comparison, we conducted another experiment with a similar setup: the network is initially 100% loaded by a best-effort flow from host h1 to host h10. A QoS flow with on-off pattern (sending 5 MBytes, then off for 7s) is sent from host h2 to h9 (7 hops). The flow has actual rate of 30 Mbps and guaranteed rate of 30 Mbps. Different from the previous experiment, the flow entries are already installed.

The result and comparison with the previous experiment are shown in Figure 8. Although the bandwidth is fully used by the best-effort flow, the QoS flow has zero packet loss. There are no signs of quality degradation caused by the bandwidth borrowing. We therefore conclude that the traffic shaping by Linux HTB in OVS is precise and reliable enough to support our bandwidth guaranteeing framework.

V. Conclusion

In this paper, we have demonstrated how to provide bandwidth guarantees with OpenFlow. We maximize bandwidth utilization by allowing idle bandwidth to be used by other QoS flows and best-effort flows. Although at first glance the "bandwidth borrowing" mechanism might seem to have an adverse effect on the QoS flows, our experiments show that the Linux HTB in OVS is reliable enough to guarantee bandwidth for QoS flows at zero packet loss.

We have also facilitated the option to prioritize best-effort traffic over non-conformant QoS traffic. This was made possible by splitting QoS flows into conformant and non-conformant traffic using OpenFlow's meter table.

Compared to Intserv, our model uses less signaling overhead. And by using aggregation, the switches in the network do not need to store every single flow state. We believe that by enabling scalability and efficient utilization, QoS is ready to be implemented in production networks.

Acknowledgment

We thank SURFnet, and in particular Ronald van der Pol, for allowing us to conduct experiments on their SDN testbed.

References

[1] R. Braden, D. Clark, and S. Shenker, "Integrated Services in the Internet Architecture: an Overview," IETF RFC 1633, 1994. url:http://www.ietf.org/rfc/rfc1633.txt.

[2] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss, "An Architecture for Differentiated Services," IETF RFC 2475, updated by RFC 3260, 1998. url: http://www.ietf.org/rfc/rfc2475.txt.

[3] 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, no. 2, pp. 69-74, April 2008

[4] S. Tomovic, N. Prasad and I. Radusinovic, "SDN control framework for QoS provisioning," Proc. of 22nd Telecommunications Forum (TELFOR), 2014.

[5] S. Dwarakanathan, L. Bass and L. Zhu, "Cloud Application HA using SDN to ensure QoS," Proc. of the 8th IEEE International Conference on Cloud Computing, 2015.

[6] Ryu SDN Framework. Accessed 2016. url: https://osrg.github.io/ryu/.

[7] S. Shenker, C. Partridge, and R. Guerin, "Specification of Guaranteed Quality of Service," IETF RFC 2212, 1997. url: http://www.ietf.org/rfc/rfc2212.txt.

[8] Q. Ma and P. Steenkiste, "On path selection for traffic with bandwidth guarantees," Proc. of IEEE ICNP, 1997.

[9] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin, "Resource ReSerVation Protocol (RSVP) Version 1 Functional Specification," IETF RFC 2205, 1997. url:http://www.ietf.org/rfc/rfc2205.txt.

[10] P. Van Mieghem and F.A. Kuipers, "Concepts of Exact Quality of Service Algorithms," IEEE/ACM Transactions on Networking, vol. 12, no. 5, pp. 851-864, October 2004.

[11] N.L.M. van Adrichem, C. Doerr, and F.A. Kuipers, "OpenNetMon: Network Monitoring in OpenFlow Software-Defined Networks," Proc. of IEEE/IFIP NOMS, 2014.