P4QoS: QoS-based Packet Processing with P4

Turkovic, B.; Biswal, S.; Vijay, A.; Hüfner, A.E.; Kuipers, F.A.

DOI
10.1109/NetSoft51509.2021.9492539

Publication date
2021

Document Version
Accepted author manuscript

Published in
Proceedings of the 2021 IEEE Conference on Network Softwarization

P4QoS: QoS-based Packet Processing with P4

Belma Turkovic, Soovam Biswal, Abhishek Vijay, Antonia Hüfner and Fernando Kuipers
Delft University of Technology, the Netherlands

***Abstract*—Networks often need to concurrently process millions of flows with varying Quality-of-Service (QoS) requirements. Doing so by deploying flow-specific rules at network nodes would require significant memory and overhead.**

In this paper, we take a fundamentally different approach, called *P4QoS*, by embedding QoS requirements in the packets themselves and leveraging P4-programmable network switches to process the traffic based on them. We illustrate and evaluate our approach with latency as our QoS metric, but our concept can be applied to other metrics as well. Our evaluation, both in software (Mininet) and in hardware (Intel Tofino), shows that *P4QoS* can satisfy application-specific QoS requirements with negligible memory overhead.

I. Introduction

Over the past years, services with very strict QoS requirements (e.g., on loss, bandwidth, latency) have emerged. Consequently, the range of application requirements that need to be simultaneously supported by the network is increasing. Moreover, these application requirements may vary during the connection, depending on the user's actions (e.g., with online gaming or emerging tactile internet applications) [3], [10].

Considering that switches can process millions of flows at a time, accommodating these varying requirements by deploying per-flow rules would cause significant management and resource overhead (e.g., queues, memory [5]). Moreover, provisioning networks to satisfy all applications with varying requirements at their peak would be inefficient [10]. In contrast to per-flow rules, nearly parameterless queue-management mechanisms, such as CoDel (Controlled Delay [9]) can be deployed without any management/configuration overhead. However, they treat all packets in the same way and cannot accommodate packets that, for example, have diverse latency constraints.

In this paper, we leverage P4-programmable networks to enable applications to request a specific QoS from the network. To do so, we embed the maximum allowed latency a packet can experience (i.e., packet deadline) in the packet itself. Subsequently, we enable P4-programmable switches [2] to decrement the packet's allowed latency based on the delay experienced on each link/switch, and deploy mechanisms that – while processing packets – ensure packet deadlines are met without any of the flows overpowering each other.

Contributions & Outline. Our key contributions are:

- *QoS-based packet forwarding*: We program the network switches to process packets based on the latency deadlines specified in the packets (Sec. II).
- *Link-latency estimation protocol (LLEP)*: To estimate the propagation and transmission delay, we develop a custom protocol that periodically updates and stores the link-latency information in the switches (Sec. II-A).
- *Cooperative AQM*: We propose a queue management mechanism with a marking target based on packet deadlines (Sec. II-B).

By allowing the applications to specify the required QoS information in the packet itself, our solution (*P4QoS*) not only simultaneously supports a wide variety of application requirements, but also applications with varying network requirements. Consequently, in contrast to existing scheduling or active queue management (AQM) schemes, our solution dynamically adjusts the switches' parameters based on the flow requirements.

II. QoS-based forwarding

Programmable switches, using P4 [2], facilitate new monitoring features based on the exact state packets experience while being processed [6], [7], [11]. We leverage this possibility to keep track of the "amount of latency" the packets already "used" while being processed in the network. To do so, each application appends an additional QoS header containing the maximum end-to-end latency the packet can still experience based on the packet's QoS deadline. Next, we develop a custom protocol that periodically calculates the link latencies (explained further in Sec. II-A). By combining this protocol with packet deadlines d_i, we program the switches to update their threshold $trgt$, which defines when the switch will start dropping or marking processed packets to signal to the end-hosts to back-off, on-the-fly. Finally, on each switch, we subtract the total delay the packet experienced while being processed on the switch (processing and queuing delay) and the delay needed to reach the next switch in the path (transmission and propagation delay) from the packet's deadline. Whenever the switch determines that the packet cannot be successfully delivered to the end-application, it invokes the mechanism to recalculate the switches $trgt$ parameter to match the current network state (explained further in Sec. II-B).

A. Link latency estimation protocol (LLEP)

Switches can use timestamps from different processing stages to estimate the propagation and transmission delay. However, switches have independent clocks and, therefore, timestamps are only useful locally. To evade this limitation, we developed a custom Link Latency Estimation Protocol (LLEP).

The ingress switch (S_1 in Fig. 1) periodically appends an LLEP header, containing an egress timestamp (representing the time the packet left the switch, t_{send} in Fig. 1, step 1), to the outgoing application data packets. Upon receiving a packet

978-1-6654-0522-5/21/$31.00

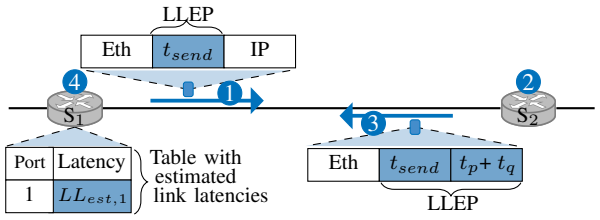


Fig. 1: Link latency estimation protocol (LLEP).

with an LLEP header, the next switch in the path saves its ingress timestamp (the time the packet was received), clones the packet, and forwards the original application data packet further along the path. Next, while processing the clone, the switch calculates the total processing and queuing time (t_p+t_q) as the difference between the current egress and the saved ingress timestamp and appends it to the LLEP header (step 2, Fig. 1). Finally, it sends the cloned packet back to the previous switch in the path (step 3, Fig. 1), which uses the appended information and its own ingress timestamp $t_{ingress}$ (the time the cloned packet was received) to calculate the estimated link latency LL_{est} as follows (step 4, Fig. 1):

$$LL_{est} = \frac{t_{ingress} - (t_{send} + t_q + t_p)}{2} \quad (1)$$

Lastly, the latency register at the ingress switch is updated and used to estimate the packet deadlines for all the flows on the same output link. All other switches use the same procedure to estimate all other link latencies. Note that the propagation and transmission delays usually do not change and, hence, this process can be infrequent. Moreover, since only the two switches forming the link are needed to estimate each link latency, our solution is topology independent.

B. Cooperative AQM

We developed a custom mechanism to determine the maximum allowed queuing delay satisfying the deadlines of all flows processed at the switch. In essence, our solution determines the marking target ($trgt$) that ensures that the flow with the lowest deadline determines the queuing delay.

First, in the QoS header, the ingress switch initializes another field "target" t_t (in addition to the deadline header field) to d. By subtracting only the estimated link latencies ($LL_{Est,i}$) from it, the switches calculate the "amount of latency" that can be spent (in total) on queuing and processing (Alg. 1, line 2). Additionally, the switches estimate the number of bottlenecks (n_{btl}, Alg. 1, line 4) by checking the queuing delay/queue depth that the packet experienced and comparing it to the sensitivity parameter s. The parameter s determines the minimum queuing delay (or minimum size of the queue) needed at a switch for it to be considered a bottleneck. If set too low, it could cause an overestimation (or oscillations) in the number of bottlenecks (i.e., switches that are not bottlenecks would be detected as such). However, if set too high, bottlenecks would be detected too late (i.e., after the queue was already formed).

Algorithm 1: Target calculation: Cooperative AQM.

Input: target trgt, flowID that updated the target $\mathrm{id_t}$

1 **if** *hdr.qos is valid* **then**
2 $\quad \mathrm{t_t} \leftarrow \mathrm{t_t} - \mathrm{LL_{Est,i}}$;
3 $\quad$ **if** $\mathrm{d_{queueing}} > \mathrm{s}$ **then**
4 $\quad\quad \mathrm{n_{btl}} \leftarrow \mathrm{n_{btl}} + 1$; ▷ number of bottlenecks
5 $\quad$ **end**
6 $\quad$ **if** *last_switch is true* **then**
7 $\quad\quad \mathrm{trgt_{new}} \leftarrow \mathrm{t_t}/\max\{\mathrm{n_{btl}}, 1\}$; ▷ calc. new trgt
8 $\quad\quad$ set_valid(hdr.target_update);
9 $\quad$ **end**
10 **end**
11 **if** *hdr.target_update is valid* **then**
12 $\quad$ **if** $\mathrm{trgt_{new}} < \mathrm{trgt}$ ***or*** $\mathrm{id_t} == \mathrm{id}$ **then**
13 $\quad\quad \mathrm{trgt} \leftarrow \mathrm{trgt_{new}}$; ▷ apply new target
14 $\quad\quad \mathrm{id_t} \leftarrow \mathrm{id}$;
15 $\quad$ **end**
16 $\quad$ send_target_update(id, $\mathrm{trgt_{new}}$);
17 **end**

Finally, using this information (remaining "amount of latency" for processing and queuing t_t, and the number of bottlenecks n_{btl}), the last switch on the path calculates the new marking target (Alg. 1, line 7) and forwards a target update packet containing this information to all the other switches in the path (Alg. 1, line 16). Upon receiving this packet, each switch compares the new marking target $trgt$ to its existing target and stores the lower one (Alg. 1, line 13). Furthermore, to filter out short-term traffic fluctuations and account for flows with variable requirements, our algorithm: (1) allows target update packets of the same flow to overwrite the saved value and (2) includes an interval-based approach that allows the target to be updated once each fixed (or adaptable) interval of N packets (seconds, See III-B). Further, every time a connection that determined the target at the switch ends (e.g., a FIN packet is received), the marking target is reset to a value higher than the maximum possible queueing delay on the switch. If one of these switches is a bottleneck (or becomes one), packets belonging to remaining flows will violate their target $trgt$ (due to excess queuing) at the last switch, triggering a recalculation of $trgt$. Therefore, our congestion control solution ensures the maximum latency is lower or equal than the sum of all the marking targets of all the switches the packet passes through. Moreover, it also ensures a fair distribution of the resources among the competing flows present on the switch by signaling to each flow to back-off whenever the lowest latency flow deadline is "in danger" of being violated.

Combination with other AQMs. State-of-the-art AQMs, such as CoDel, can easily be combined with our approach by adjusting the marking target with our approach (see Sec. III).

C. Overhead and limitations

Links towards end-hosts. LLEP does not support latency estimation of links connecting the end-hosts. However, if end-

hosts would be programmable (e.g., via smartNICs), LLEP could be extended to include them. As this is not generally the case, LLEP can make use of the packets involved in connection establishment (e.g., the 3-way handshake for TCP).

Time units. To ensure the accuracy of LLEP, both switches need to support the same time-scale (e.g., nanoseconds). However, this problem can be circumvented by approximating the division using bit shift operations.

Flows with a higher deadline. The switch might mark packets with a higher deadline even if its deadline d is not violated to signal to the applications to reduce their rate. However, if these packets are marked (and not dropped), they will still be received at the end-hosts. Moreover, since their latency might be unnecessarily inflated (due to static queues), forcing them to back-off might improve overall network performance and ensure fairness.

Determining the reverse path for target update packets. If traffic is bidirectional, and packets from the receiver to the sender are processed on the same path as the traffic from the sender to the receiver, no additional rules are needed. The last switch can reverse the source and destination IP (or MAC) addresses and, consequently, the target update packet will be processed using the existing rules. Otherwise, a source route can be created by appending the ingress port on each switch to the QoS header during target calculation.

Delay overhead. The additional QoS header increases the transmission delay on each output link. However, this overhead is constant and, typically, negligible (e.g., 4μs per switch with a $10Mbps$ output link).

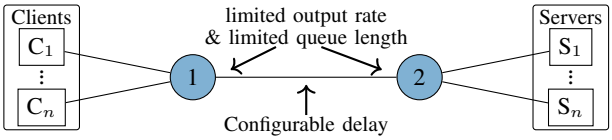
Memory overhead. Our approach uses additional 16 $B{\cdot}n_{ports}$ to store the link latency information (cf. Sec. II-A marking target and the flow that updated the target (cf. Sec. II-B). Thus, on a typical 24-port switch, our approach would consume only 386 B, which is a negligible amount on modern programmable hardware switches.

Packet overhead. Our approach generates additional packets to estimate the link latency and update the marking target. While link latencies can be infrequent, target updates will occur depending on latency deadlines on three different occasions: (1) a new flow is elected as the one with the lowest deadline on a switch, (2) the deadline of the flow determining the deadlines changes, or (3) a change in the traffic pattern causes the formation of a new bottleneck or shifting of an existing one. Thus, target updates can be frequent, for which we explored two periodic filtering approaches (see Sec. III-B).

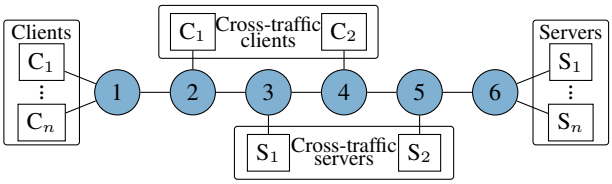
Floating-point operations. To make sure that switches will run at line-rate, certain operations are restricted. In particular, the programmable switches lack division and floating-point operations. Thus, while calculating the target (Alg. 1, line 7), we use the bit-shift operation. Consequently, if the number of switches is not a power of 2, the configured $trgt$ is lower/higher than needed causing the switch to react too fast/slow.

III. Evaluation

Experiment topology. To evaluate our solution, we used the topologies shown in Fig. 2. For the first topology (Fig. 2a) we used the Intel Tofino switch [4]. Since we only had one Tofino switch, nodes 1-2 were run using the same Tofino switch with two different physical ports connected. Since the processing is not the limiting factor (the switch runs at line-rate), this should not affect our results. For the second topology (Fig. 2b) we used Mininet with the software_switch (behavioural model 2 [1]).



(a) Single bottleneck topology (2-switches, 1 links).



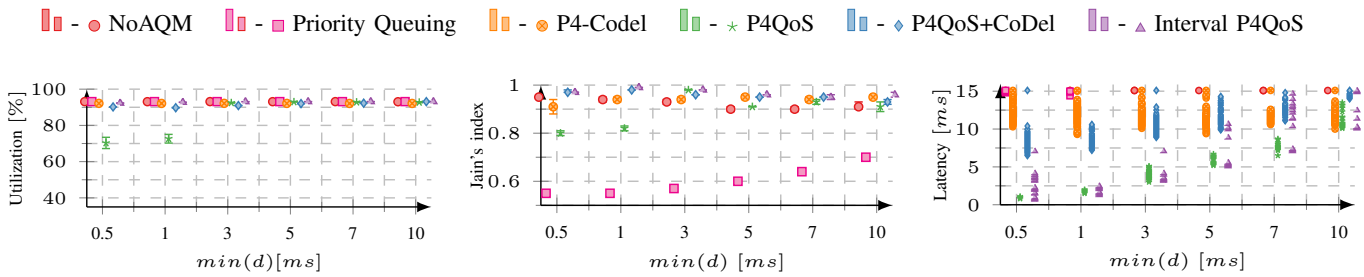
(b) Multiple-bottlenecks topology (6-switches, 5 links).

Fig. 2: Topologies.

Experiment scenarios. We generated two concurrent 60s-long TCP Cubic flows with different deadlines (d_1 and d_2). To perform measurements, we relied on iperf, netem, switch statistics, and tcpdump. Next, in the "multiple bottlenecks" scenario (Fig. 2b), we generated the same two flows, and additional cross-traffic between C1 and S1 from 10-40s, and between C2 and S2 from 20-50s in an attempt to emulate different traffic volumes. Each scenario was run ten times, and for all of them we observed similar results.

Performance metrics. To evaluate our approach, we used the following metrics: (1) the percentage of packets violating their deadline, (2) the average and maximum round-trip times and the one-way delays, (3) utilization as the percentage of the total available bandwidth used by all the connections, (4) Jain's fairness index (that ranges from $1/n$ to 1, where n is the number of flows) and (5) number of sent $trgt$ updates.

Comparison baselines. We compared our solution against (1) a simple switch without any mechanism to either differentiate between flows or address excessive queuing (NoAQM), (2) P4-CoDel as the state-of-the-art AQM solution [8], and (3) priority queuing (Prio) that assigns packets with lower deadlines to higher priority queues. Additionally, we tested three different versions of P4QoS: (1) P4QoS, that marks all packets exceeding the marking target $trgt$, (2) P4QoS+CoDel, that adjusts the marking target of P4-CoDel based on the P4QoS algorithm, and (3) Interval P4QoS, that only marks one packet that exceeds $trgt$ (configured using P4QoS) each interval, where that interval is auto-tuned in the same way as for Codel



(a) Average utilization.

(b) Average fairness index.

(c) Maximum delay per flow.

d_2 [ms] \ d_1 [ms]	0.5	1	3	5	7	10
0.5	99.99	99.98	99.91	99.31	98.46	96.60
1	99.98	99.96	99.72	99.33	98.50	97.00
3	99.91	99.72	99.31	98.86	98.81	97.13
5	99.31	99.33	98.86	98.92	99.27	99.22
7	98.46	98.50	98.81	99.27	99.52	98.26
10	96.60	97.00	97.13	99.22	98.26	98.10

(d) NoAQM. Average percentage of packets that violated their deadline.

d_2 [ms] \ d_1 [ms]	0.5	1	3	5	7	10
0.5	99.99	99.80	99.89	99.78	99.76	99.63
1	99.80	99.97	99.84	99.74	99.71	99.69
3	99.89	99.84	99.88	99.62	99.68	99.56
5	99.78	99.74	99.62	99.73	99.49	98.27
7	99.76	99.71	99.68	99.49	99.31	91.70
10	99.63	99.69	99.56	98.27	91.70	98.29

(e) Priority queueing. Average percentage of packets that violated their deadline.

d_2 [ms] \ d_1 [ms]	0.5	1	3	5	7	10
0.5	99.82	99.36	98.97	81.51	49.59	50.95
1	99.36	99.76	98.94	79.97	50.56	50.22
3	98.97	98.94	97.53	79.83	51.00	48.28
5	81.51	79.97	79.83	62.84	33.56	30.91
7	49.59	50.56	51.00	33.56	2.16	1.41
10	50.95	50.22	48.28	30.91	1.41	0.21

(f) P4-CoDel. Average percentage of packets that violated their deadline.

d_2 [ms] \ d_1 [ms]	0.5	1	3	5	7	10
0.5	0.89	0.54	0.50	0.47	0.54	0.51
1	0.54	0.92	0.54	0.53	0.53	0.50
3	0.50	0.54	0.35	0.15	0.18	0.17
5	0.47	0.53	0.15	0.11	0.07	0.08
7	0.54	0.53	0.18	0.07	0.09	0.08
10	0.51	0.50	0.17	0.08	0.08	0.13

(g) P4QoS. Average percentage of packets that violated their deadline.).

d_2 [ms] \ d_1 [ms]	0.5	1	3	5	7	10
0.5	99.28	84.77	52.84	50.50	49.77	49.43
1	84.77	99.00	53.60	51.22	49.42	49.90
3	52.84	53.60	88.60	46.49	45.22	44.53
5	50.50	51.22	45.49	66.04	34.15	32.95
7	49.77	49.42	44.22	34.15	46.36	24.33
10	49.43	49.90	45.53	32.95	24.33	28.72

(h) P4QoS+CoDel. Average percentage of packets that violated their deadline.

d_2 [ms] \ d_1 [ms]	0.5	1	3	5	7	10
0.5	1.87	1.11	0.83	1.01	0.82	0.94
1	1.11	1.97	0.99	0.98	0.99	0.92
3	0.83	0.99	0.21	0.10	0.09	0.10
5	1.01	0.98	0.10	0.15	0.07	0.05
7	0.82	0.99	0.09	0.07	0.05	0.02
10	0.94	0.92	0.10	0.05	0.02	0.05

(i) Interval P4QoS. Average percentage of packets that violated their deadline.

Fig. 3: Evaluation using the Tofino switch (topology 2a). Confidence intervals 95%.

(i.e., if a packet is marked the interval is reduced, otherwise the interval is reset to its default value ($100ms$)). Furthermore, to limit the number of target update packets, we tested two back-off mechanisms: (1) fixed P4QoS, which sends a target update only once each N packets (seconds), and (3) adaptive P4QoS, which adapts the interval between each target update based on current traffic conditions, i.e., bottleneck counts.

A. Single-bottleneck topology

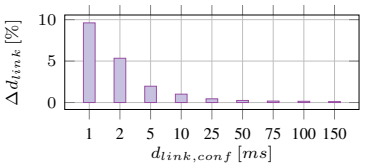
LLEP accuracy. In our hardware experiments, we estimated the middle link to be between $59 - 67ns$.

Utilization. Due to its very aggressive marking policy, P4QoS experienced a drop in throughput, especially for lower values of $trgt$. In contrast, by only marking one packet each interval, and thus only punishing one flow at a time, P4-CoDel, P4QoS+Codel, and Interval P4QoS maintained a high utilization independently of $trgt$ (Fig. 3a).

Fairness. All evaluated solutions, except priority queuing, were able to maintain a high fairness index ($\geq$ 0.80 on average, Fig. 3b). Due to its strict priority approach, priority queueing always favored the lower deadline packets. Thus, in all scenarios in which the flows had different d, those flows (with the lower d) claimed all the link resources, resulting in low fairness. In scenarios where flows had the same d, priority queueing had the same performance as NoAQM. Further, due to its very aggressive marking policy and consequent drops in utilization, P4QoS has a slightly lower fairness index on lower values of $trgt$.

Delay. Taking flow properties into account (using P4QoS) reduces the maximum end-to-end delay experienced by the flows (Fig. 3c). P4QoS and Interval P4QoS maintained the maximum end-to-end delay around the lowest deadline. P4-CoDel (and P4QoS+P4-CoDel) only dropped a packet when the queueing delay of all packets processed in an interval was higher than $trgt$. Consequently, they allowed the queues to form far beyond d and had a higher maximum delay (Fig. 3c). Finally, NoAQM and priority queueing formed static queues, and all packets experienced the maximum queueing delay.

Percentage of packets violating the deadline. As Fig. 3d-3h illustrate, P4QoS and Interval P4QoS have the best performance and were able to maintain a low percentage of packets that violate the deadline ($\leq 2\%$). In contrast, NoAQM, Priority Queueing, and P4-CoDel had the worst performance. Consequently, even when they had higher utilization compared to P4QoS, most of the processed packets were outdated at the destination host and, thus, no longer useful (e.g., 70.30% utilization with only 0.89% outdated packets on average for P4QoS compared to 93.11% utilization with 99.99% outdated packets on average for NoAQM). Similarly, P4QoS-Codel, in which $trgt$ specifies the minimum queueing delay all packets

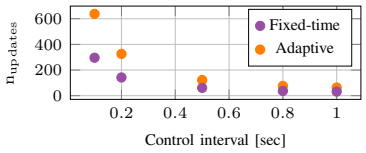


(a) Average error in the LLEP estimation (Δd_{link}) for different configured link latencies ($d_{link,conf}$) with 90% confidence.

Time interval [sec]

n_{btl}	0-10	10-20	20-30	30-40	40-50	50-60
0	17.74	27.42	15.96	22.18	27.82	14.52
1	80.31	62.70	61.01	57.26	59.68	83.67
2	1.75	9.68	22.22	20.16	12.30	1.81
3	0.20	0.20	0.81	0.40	0.20	0.00

(b) Average percentage of bottleneck counts per time interval each representing a specific traffic scenario in the simulation.



(c) Comparison of fixed-time and adaptive-time interval P4QoS schemes based on the average frequency of target updates.

Fig. 4: Simulation results using Mininet (topology 2b).

processed in an interval need to have for it to kick-in, violated almost all latency constraints for the flow with the lower deadline, leading to $\approx$ 50% of packet violations. Interval P4QoS, that kicks-in when the queueing delay of any packet exceeds $trgt$ (but then idles for the duration of an interval), combined the best of both P4QoS and P4-Codel. First, using the auto-tuned interval approach and only marking one packet per interval achieved high utilization. Second, setting $trgt$ based on the packets' current deadlines, maintained a low percentage of packets violating their deadline.

B. Multiple Bottleneck Scenario

LLEP accuracy. First, we evaluated the accuracy of LLEP by varying the configured link latency $d_{link,conf}$ (Fig. 4a). In all the scenarios, the difference between the configured and estimated link latency was less than 200μs (Fig. 4a). As this value was nearly constant in our experiments, we conclude that this overhead is the processing overhead of the egress block (i.e., processing after setting the timestamp in the LLEP packet), the parser, the deparser, and the additional overhead while processing the packet in the kernel. Especially on higher $d_{link,conf}$, this inaccuracy was negligible ($\leq 0.5\%$ for $d_{link,conf} = 25ms$).

Bottleneck count By generating different patterns of cross-traffic, we evaluated the ability of our solution to detect bottlenecks and to adjust the dropping target adequately. As illustrated in Fig. 4b, without cross-traffic, the number of detected bottlenecks is relatively low. As the cross-traffic flows commence at 10s, a greater number of bottlenecks is detected. This trend increases further once both cross-flows are present and decreases upon their completion ($\geq 50s$).

Target update overhead. Depending on the traffic pattern, P4QoS can generate a lot of updates ($\approx 34107 \pm 1323$ packets over the 60s). Fixed P4QoS - that sends a single update each control interval - produced far fewer update packets ($\approx 296 \pm 11$ over 60s) with only a slight drop in throughput in some experiments. Finally, adaptive P4QoS had no perceptible difference in performance compared to P4QoS, but sent a higher amount of updates than fixed P4QoS ($\approx 638 \pm 23$ over 60s, Fig. 4c).

IV. Conclusion

This paper leveraged the advanced monitoring possibilities of programmable data-planes to enable the network to process packets based on their per-packet specified QoS, by enabling the switches to keep track of the latency the packet experienced while being processed in the network. Next, this paper introduced a queue management system with low processing and memory overhead that works entirely on a per-packet basis without keeping track of flow states. Furthermore, we showed that our solution *P4QoS*, especially when combined with the interval approach (Interval P4QoS), can significantly improve the performance in the network, especially for low-latency traffic, by significantly reducing the amount of marked (outdated) packets without causing a drop in throughput. Finally, we evaluated our solution in a more complex topology, and included two mechanisms that can improve the stability by reducing the amount of sent target updates.

References

[1] P4 behavioral model. https://github.com/p4lang/behavioral-model. Accessed: 19-03-2018.

[2] Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., and Walker, D. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev. 44*, 3 (July 2014), 87–95.

[3] Claypool, M., and Claypool, K. Latency and player actions in online games. *Communications of the ACM 49*, 11 (2006), 40–45.

[4] Intel® Tofino™: P4-programmable Ethernet switch ASIC that delivers better performance at lower power. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html. [Online; accessed 03-November-2020].

[5] Karakus, M., and Durresi, A. Quality of service (qos) in software defined networking (sdn): A survey. *Journal of Network and Computer Applications 80* (2017), 200–218.

[6] Kim, C., Bhide, P., Doe, E., Holbrook, H., Ghanwani, A., Daly, D., and Hira, Mukesh amd Davie, B. In-band network telemetry (int), 2016. https://p4.org/assets/INT-current-spec.pdf, Last accessed on 10-06-2020.

[7] Kim, C., Sivaraman, A., Katta, N., Bas, A., Dixit, A., and Wobker, L. J. In-band network telemetry via programmable data-planes. In *ACM SIGCOMM* (2015).

[8] Kundel, R., Blendin, J., Viernickel, T., Koldehofe, B., and Steinmetz, R. P4-codel: Active queue management in programmable data planes. In *2018 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)* (2018), IEEE, pp. 1–4.

[9] Nichols, K., Jacobson, V., McGregor, A., and Iyengar, J. Controlled delay active queue management. *RFC 8289 1* (2018).

[10] Polachan, K., Turkovic, B., Prabhakar, T., Singh, C., and Kuipers, F. A. Dynamic network slicing for the tactile internet. In *2020 ACM/IEEE 11th International Conference on Cyber-Physical Systems (ICCPS)* (2020), IEEE, pp. 129–140.

[11] Turkovic, B., Kuipers, F., van Adrichem, N., and Langendoen, K. Fast network congestion detection and avoidance using p4. In *Proceedings of the 2018 Workshop on Networking for Emerging Applications and Technologies* (2018), pp. 45–51.