Enabling TCP Pacing using Programmable Data Plane Switches

Elie F. Kfoury*, Jorge Crichigno*, Elias Bou-Harb†, David Khoury‡, and Gautam Srivastava§
*Integrated Information Technology, University of South Carolina, Columbia, U.S.A.
†Cyber Threat Intelligence Lab, Florida Atlantic University (FAU), Florida, U.S.A.
‡Computer Science Department, American University of Science and Technology (AUST), Lebanon.
§Department of Mathematics and Computer Science, Brandon University, Canada.
Email: ekfoury@email.sc.edu, jcrichigno@cec.sc.edu, ebouharb@fau.edu, dkhoury@aust.edu.lb, srivastavag@brandonu.ca

Abstract*—Previous studies have observed that TCP pacing - evenly spacing out packets- minimizes traffic burstiness, reduces packet losses, and increases throughput. However, the main drawback of pacing is that the number of flows and the bottleneck link capacity must be known in advance. With this information, pacing is achieved by manually tuning sender nodes to send at rates that aggregate to the bottleneck capacity. This paper proposes a scheme based on programmable switches by which rates are dynamically adjusted. These switches store the network's state in the data plane and notify sender nodes to update their pacing rates when the network's state changes, e.g., a new flow joins or leaves the network. The scheme uses a custom protocol that is encapsulated inside the IP Options header field and thus is compatible with legacy switches (i.e., the scheme does not require all switches to be programmable). Furthermore, the processing overhead at programmable switches is minimal, as custom packets are only generated when a flow joins or leaves the network. Simulation results conducted in Mininet demonstrate that the proposed scheme is capable of dynamically notifying hosts to adapt the pacing rate with a minimum delay, increasing throughput, mitigating the TCP sawtooth behavior, and achieving better fairness among concurrent flows. The proposed scheme and preliminary results are particularly attractive to applications such as Science DMZ, where typically a small number of large flows must share the bandwidth capacity.

***Keywords*—TCP pacing; programmable switches; P4 language; fairness**

I. Introduction

Bandwidth is a constrained resource that measures the transmission capacity over a path in a given amount of time. Carefully provisioning and consuming bandwidth affect network's performance and end-hosts' communications. Carefully provisioning and consuming bandwidth affect network's performance and end-hosts' communications. Research centers, governments, and educational institutions are facing difficulties transferring tremendous amounts of data [1] via existing network protocols.

The majority of network applications rely on the Transmission Control Protocol (TCP) as the TCP/IP protocol stack dominates current communication systems [1]. Overall TCP throughput is tainted by its bursty transmissions which increase packet losses. Previous studies observed that *pacing*, which is a technique used to evenly space packets into the network over an entire RTT, can minimize the packet burstiness, especially in high-speed networks [2].

This work was supported by the U.S. National Science Foundation, award 1829698.

Notably, the first TCP congestion control algorithm based upon pacing has been recently proposed, namely, the Bottleneck Bandwidth and Round-Trip Time (BBR) algorithm [3]. Furthermore, the Energy Science Network (ESnet), which connects the U.S. national laboratories and hundreds of research institutions across the world, has recently demonstrated [4] that pacing produces more stable throughput in networks such as Science Demilitarized Zones (Science DMZs). The Science DMZ is a network designed to facilitate the transfer of big science data. In these environments, typically there is a small number of large concurrent flows. This scenario contrasts with regular enterprise networks, where there is a large number of small flows [5].

While pacing reduces burstiness and improves performance, it is still being manually configured on end-hosts as there is no effective mechanism to adaptively determine the desired pacing rate that aggregates to the overall bandwidth.

In this paper, we propose a scheme for adapting the transmission rate of end-hosts with the help of programmable data planes. The switches preserve the network's state by maintaining the number of active connections. When new hosts join the network, the switches automatically inform all hosts about the new bottleneck-link consumption. Consequently, end-hosts adjust their transmission rate according to the values specified by the switches.

Specifically, we frame the paper's contributions as follows:

- Storing the network state in a custom header inserted in the IP options field of a packet.
- Demonstrating the capability of storing the network state in the switches' registers.
- Defining a new mechanism for notifying previously connected hosts about network changes.
- Validating the proposed mechanism by illustrating the simulation results which compare the observed throughput via pacing against regular TCP.

The proposed scheme is particularly attractive to applications such as Science DMZ, where a handful of large flows must share the network capacity. Using programmable switches provides an automated mechanism to dynamically share the available bandwidth.

The rest of the paper is divided as follows: Section II provides a short background on programmable data planes and TCP pacing. Section III describes the proposed system. Section IV demonstrates and compares the simulation results against regular TCP. Finally, section V concludes the outlined system with the intended future work.

II. Background

A. Programmable Data Plane and P4

Although Software Defined Networking (SDN) offers numerous advantages compared to traditional network architectures [6], it still lacks full programmability as the forwarding plane is embedded on a fixed Application-Specific Integrated Circuit (ASIC) chip [7]. Once the ASIC is designed, it cannot be changed; this approach raised several limitations in terms of time, cost, and feasibility. Protocol Independent Switch Architecture (PISA) is a single pipeline forwarding architecture (Figure 1) that allows full programmability in the data plane. The architecture consists of a parser, an ingress pipeline, a scheduler (buffer), egress pipeline, and deparser. Programming Protocol-independent Packet Processors (P4) is a programming language that defines the processing logic of packets on data planes [8]. It allows programmers to describe packets' headers (fields and sizes) and program a parser which defines the sequence of allowed headers in packets. P4 introduced `tables`, which can be used for storing routing entries, access-control lists (ACLs), flow lookup, etc. follow a match-action strategy, where keys are matched against table entries, and their corresponding actions are executed. Most P4 constructs are stateless by design: information is not retained across packets. As stateless constructs require less packet processing and storage, they are attractive for high-performance devices operating at very high rates (e.g., 10/100 Gbps). However, *extern* objects like `counters` and `registers` can be used by the switches to store values and states [9]. Specifically, registers declared in a P4 program can be modified or read during run-time by both data and control planes. Other useful features in P4 and programmable switches include *packet cloning* and *packet recirculating*. Packet cloning is a mechanism where replicas of an incoming packet can be redirected to specified ports after ingress and/or egress pipelines, while packet recirculating restarts the ingress processing of a packet after finishing the egress pipeline.

B. TCP Pacing

TCP pacing is a technique performed by the network scheduler to evenly space, or "pace", TCP packets transmitted to a network. It is known from the literature that pacing renders TCP traffic less bursty. In Linux-based systems, network traffic can be controlled by several Queueing Disciplines (`qdisc`) used in conjunction with the `tc` (Traffic Control) tool. In this work we focus on the most two commonly used disciplines: Fair Queueing (FQ) and Hierarchical Token Bucket (HTB). In these queueing disciplines, aggregate queues are used to associate token buckets in order to limit the transmission rate.

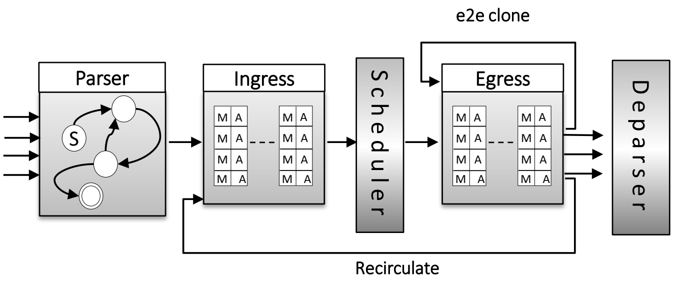


Fig. 1: Protocol Independent Switch Architecture (PISA).

1) Fair Queueing (FQ): FQ performs flow separation to achieve pacing; it is designed to follow the requirements set by the TCP stack [10]. Generally, a flow is considered all packets pertaining to a particular socket. FQ uses the red-black tree data structure [11] to index and track the state of single flows. A red-black tree is a binary search tree that ensures that no path in the tree is more than twice as long as any other. This property ensures that tree operations have a logarithmic complexity. FQ achieves fairness through the Deficit Round Robin (DRR) algorithm [12] illustrated in Figure 2. The DRR is an algorithm that allows each flow passing through a network device to have a nearly perfect fairness and requires only $O(1)$ operations per packet.

FQ uses the leaky bucket queue where transmitting timestamps (indexed on the RB tree) are derived from the pacing rate specified by the user and the packet size. FQ is a non-work conserving scheduler, therefore, it can have idle scheduled resources idle even if there are jobs ready to be scheduled.

2) Hierarchy Token Bucket (HTB): HTB is a classful qdic that follows a tree-based data structure (Figure 3) to represent traffic shapers [13]. Incoming packets to an HTB-enabled network scheduler are classified into classes, where each class contains a token bucket shaper in the leaf node. HTB follows a top-down approach to determine which class should be used for shaping traffic, while child nodes can *borrow* from parent nodes. According to [14], CPU usage of HTB grows linearly

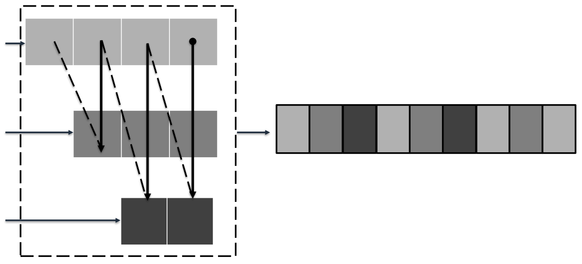


Fig. 2: Deficit Round Robin (DRR) algorithm.

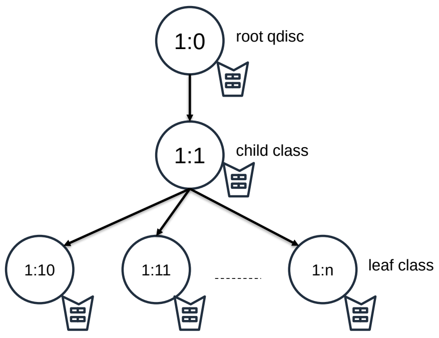


Fig. 3: Hierarchy Token Bucket (HTB) organization.

with the packets per second (PPS) rate. FQ and HTB pacing provide good rate conformance (i.e., how close the measured throughput deviates from the paced rate), deviating at most 5% to 6% from the target rate, but at the expense of CPU cost.

III. Proposed System and Design Principles

Our work began with the observation that *pacing*, which requires transmitting nodes to send at a fixed rate, minimizes the burstiness of transmission rate. Pacing is achieved by *manually* tuning end hosts to send at constant rates that aggregate to the maximum available bandwidth (bottleneck). A core challenge is to determine the number of active flows in order to evenly split the bottleneck bandwidth. Another major challenge is to instantly adapt the transmission rate of previously connected hosts whenever new hosts join the network.

Our proposed system aims at solving the aforementioned challenges by storing the network's current state in programmable switches. These switches instantly notify end hosts to update their transmission rates when the network's state changes, i.e., a new host joined the network. As our focus in this work is on TCP as the transport-layer protocol, we rely on the flags (SYN, SYN-ACK, and ACK) used in its 3-way handshake to determine the state of the connection. Figure 4 demonstrates a high-level architecture of the system. To initiate a TCP connection (1), an end-host inserts a custom header in the IP-Options field [15] of the IP header, indicating its will to establish a new connection. This custom header is inserted once during the 3-way handshake, particularly, when sending the TCP-SYN flag. The switches on the other hand parse the headers of the received packets, and check if a new flow is requested by the end-host. In such case, each switch stores the new state ***S'*** of the network (2), and inserts its bottleneck link capacity and the current total number of hosts in the SYN-ACK message. Afterwards, the switches broadcast ***S'*** to all previously connected hosts. Finally, when the hosts receive the SYN-ACK message (i.e., the second message in the handshake), they adjust their transmission rates according to the values present in the IP options header (4). Upon receiving a TCP packet on the port used for initiating its TCP connection, the end-host parses the custom header, and adjusts its transmission rate by applying pacing with a rate derived from the values of the custom header fields. The rate is adjusted according to Equation (1).

$$R' = \frac{1}{m}\Big[\min_{0\le i\le n-1} S_i[C]\Big] - l \tag{1}$$

Where R' is the newly adjusted rate, S is a variable-sized array that holds the list of all switch traces. Each trace describes a link capacity C and the number of hosts m currently connected to the switch. The amount of bandwidth to be subtracted from the total bandwidth is described as l; preserving this small portion of the bandwidth is advantageous for maintaining the stability of the paced flows.

IV. Simulation Results

In this section, we demonstrate and compare the simulation results of the custom header against regular TCP in Mininet [16]. We use the P4 programming language [8] to define the proposed header on the *bmv2* switch [17] (software switch), and *iperf* tool [18] to measure the throughput of the network. Moreover, we evaluate the approach using the commonly used queuing disciplines explained in Section II-B (FQ and HTB). Traffic Control (*tc*) Linux command [19] is used to configure the queueing disciplines on the hosts' packet schedulers, while the *netem* tool [20] is used to configure bottleneck bandwidth and inject delay.

The simulated scenario consists of four hosts connected to a server through a P4 programmable switch as shown in Figure 5. We ran the simulation 10 times and averaged the results in two different scenarios: 1) the link connecting the server to the switch is pre-configured to a maximum bandwidth of 300 Mbps with no injected delay, and 2) The same link is pre-configured to a maximum bandwidth of 100 Mbps with

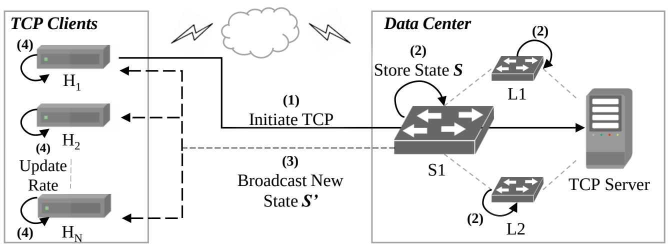


Fig. 4: High-level network architecture.

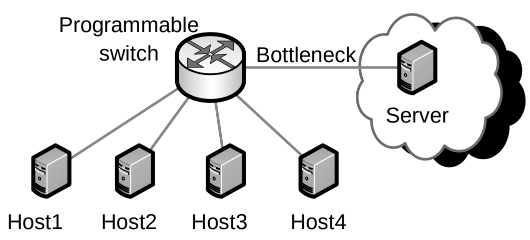
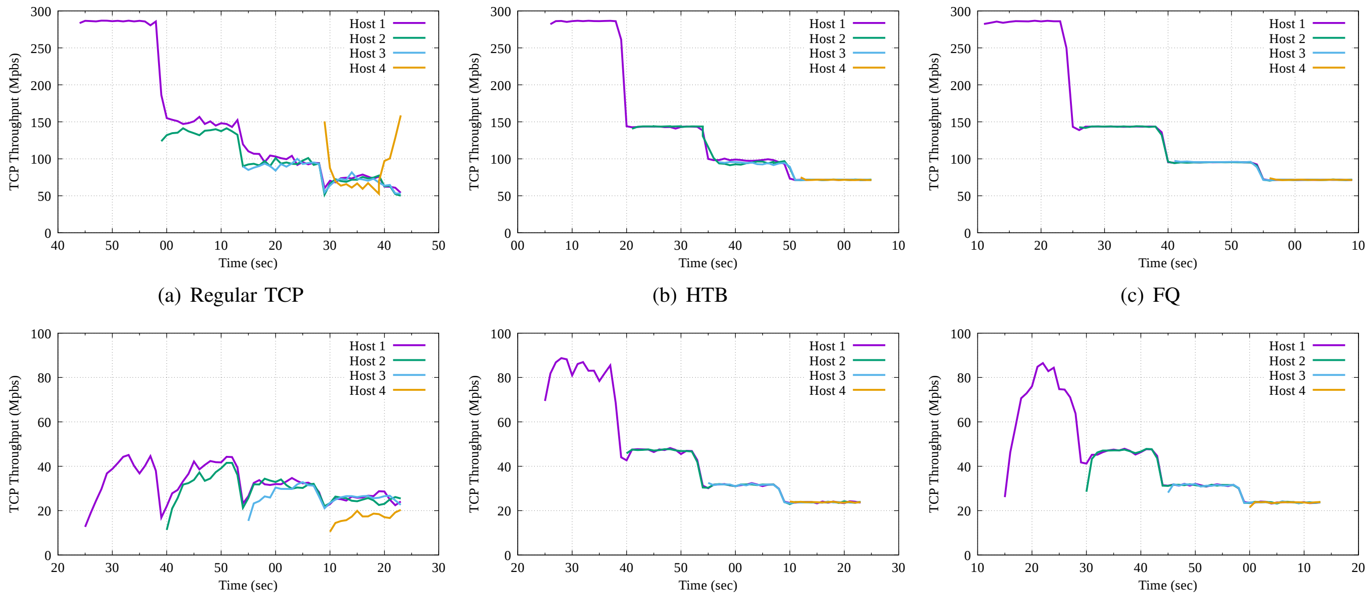


Fig. 5: Simulation topology.



(a) Regular TCP (b) HTB (c) FQ

(d) Regular TCP, 20 ms RTT scenario (e) HTB, 20 ms RTT scenario (f) FQ, 20 ms RTT scenario

Fig. 6: TCP throughput.

a 20 ms delay. The second scenario is to emulate a WAN where variable delay, loss, duplication and re-ordering occur. To verify that the system notifies previously connected hosts about network changes, we configured hosts to join 15 seconds after each other.

The above scenario reflects networks such as Science DMZs, where a small number of end devices (data-transfer nodes) evenly shares the bandwidth capacity [4]. In such network, the operator controls the end hosts and switches, thus allowing the cooperation between these devices.

To verify that the system notifies previously connected hosts about network changes, we configured hosts to join 15 seconds after each other. Figure 6 illustrates the throughput of the transmitting hosts in regular TCP, HTB and FQ modes in both scenarios. The upper sub-figures correspond to the first scenario (300 Mbps, less than 1 ms RTT) while the lower ones correspond to the second scenario (100 Mbps, 20 ms RTT). Table I lists the mean throughput of all hosts in all modes grouped by time periods (P_1: 01 - 15 sec, P_2: 16-30 sec, P_3: 31-45 sec, and P_4 46-60 sec) in scenario 2.

Consider Figure 6(a), which shows the throughput of regular TCP, and Figures 6(b) and 6(c), which show the throughput of TCP pacing using HTB and FQ respectively. While the throughput performance is similar, the fairness and reduced variation achieved with pacing can be noted graphically as the number of active flows increases. The performance difference between non-paced and paced flows is exacerbated as the RTT increases to 20 milliseconds, as seen in Figures 6(d)-(f) and summarized in Tables I and II. With one active flow only in the network, HTB pacing produces the highest throughput (see Table I) (81.25 Mbps), followed by FQ pacing (66.59 Mbps) and regular TCP (33.62 Mbps). When two flows are active between times 16 - 30, pacing continues to produce higher throughput (HTB: 93.1 Mbps, FQ: 89.81 Mbps, regular TCP: 67.27 Mbps). When more flows join the network, the performance of regular TCP becomes more competitive.

Consider now Table II, which lists measurements of coefficient of variation (CV) of each flow and fairness index (F). The CV is calculated as the standard deviation of the host's average throughput divided by the average throughput, and multiplied by 100. Thus, the CV is given in percentage. The fairness index for a set of flows is calculated using Jain's equation [21]:

$$F = \frac{\left(\sum\limits_{i=1}^{n} T_i\right)^2}{n \cdot \sum\limits_{i=1}^{n} (T_i)^2}, \tag{2}$$

where T_i is the throughput of flow i (in our simulated scenario, $n = 4$ and $i = 1, 2, ..., 4$).

The coefficient of variation reflects how much the throughput of a flow varies with respect to its average. For many applications (e.g., large flows observed in Science DMZs), a minimum variation is highly desirable. One of the main results of the proposed scheme is achieved with HTB pacing. Note the minimal variation, in particular when multiple flows are active. E.g., with HTB, when Hosts 1 and 2 are transmitting, their corresponding CVs are 3.773 and 2.998. On the other hand, with regular TCP, the CVs of Hosts 1 and 2 are 22.63 and 30.08 respectively. When all hosts are active, HTB produces CVs of less than 2% (1.168, 1.138, 0.755, and 0.684 for Hosts 1, 2, 3, and 4 respectively). In contrast, regular TCP produces CVs of 7.806, 5.260, 6.447, and 17.27 for Hosts 1, 2, 3, and 4 respectively. Thus, the throughput variation is substantially reduced.

TABLE I: Throughput (T_i) in Mbps (average over time) for Hosts i = 1, ..., 4 (see Figure 5). Hosts start transmitting at times t = 1 (Host 1), t = 16 (Host 2), t = 31 (Host 3), and t = 46 (Host 4) seconds. The RTT between the sender Hosts 1-4 and the receiver server is 20 milliseconds.

	Regular TCP					**HTB**					**FQ**				
Period	$\sum T_i$	**T$_1$**	**T$_2$**	**T$_3$**	**T$_4$**	$\sum T_i$	**T$_1$**	**T$_2$**	**T$_3$**	**T$_4$**	$\sum T_i$	**T$_1$**	**T$_2$**	**T$_3$**	**T$_4$**
P$_1$ (01-15 sec)	**33.62**	33.62	N/A	N/A	N/A	**81.25**	81.25	N/A	N/A	N/A	**66.59**	66.59	N/A	N/A	N/A
P$_2$ (16-30 sec)	**67.27**	36.06	31.21	N/A	N/A	**93.1**	46.40	46.70	N/A	N/A	**89.91**	45.85	44.06	N/A	N/A
P$_3$ (31-45 sec)	**88.83**	31.27	30.61	26.95	N/A	**94.42**	31.40	31.37	31.65	N/A	**93.72**	31.40	31.36	30.96	N/A
P$_4$ (46-60 sec)	**91.86**	25.32	24.63	25.32	16.59	**95.12**	23.78	23.75	23.73	23.86	**94.52**	23.71	23.71	23.67	23.43

TABLE II: Coefficient of Variation (CV_i) in percentage and fairness index F, for Hosts i = 1, ..., 4 (see Figure 5). The ideal fairness index value is 1. Hosts start transmitting at times t = 1 (Host 1), t = 16 (Host 2), t = 31 (Host 3), and t = 46 (Host 4) seconds. The RTT between the sender Hosts 1-4 and the receiver server is 20 milliseconds.

	Regular TCP					**HTB**					**FQ**				
Period	**F**	**CV$_1$**	**CV$_2$**	**CV$_3$**	**CV$_4$**	**F**	**CV$_1$**	**CV$_2$**	**CV$_3$**	**CV$_4$**	**F**	**CV$_1$**	**CV$_2$**	**CV$_3$**	**CV$_4$**
P$_1$ (01-15 sec)	1.00	32.32	N/A	N/A	N/A	1.0000	8.188	N/A	N/A	N/A	1.0000	28.427	N/A	N/A	N/A
P$_2$ (16-30 sec)	.994	22.63	30.08	N/A	N/A	.99998	3.773	2.998	N/A	N/A	.99960	4.351	14.142	N/A	N/A
P$_3$ (31-45 sec)	.994	9.349	10.90	19.69	N/A	.99998	2.065	2.081	1.985	N/A	.99960	1.618	1.317	3.879	N/A
P$_4$ (46-60 sec)	.974	7.806	5.260	6.447	17.27	.99999	1.168	1.138	.755	.684	.99997	1.022	1.020	.996	3.336

V. Conclusion

In this paper, we have proposed a mechanism for instantly adapting the transmission rate of TCP flows using a custom header processed by programmable data switches. Our aim is to automate end hosts TCP pacing to enhance throughput and minimize variations. The solution offers a novel way to store the network state in a P4 switch and to notify previously connected hosts about a change in the network, particularly, when a new host joins.

Our implementation on Mininet showed that the solution has indeed achieved the desired goals while increasing the overall mean throughput. For future work, we intend to modify the custom header to support more complex topologies and interwork with dynamic reroutes performed by the switches whenever congestion occurs. We also plan to extend the sharing bandwidth scheme for scenarios where an uneven allocation is desirable (priorities).

References

[1] B. A. Forouzan and S. C. Fegan, *TCP/IP protocol suite*. McGraw-Hill Higher Education, 2002.

[2] N. Hanford, B. Tierney, and D. Ghosal, "Optimizing data transfer nodes using packet pacing," in *Proceedings of the Second Workshop on Innovating the Network for Data-Intensive Science*, p. 4, ACM, 2015.

[3] N. Cardwell, Y. Cheng, S. Yeganeh, and V. Jacobson, "Bbr congestion control," *Working Draft, IETF Secretariat, Internet-Draft draft-cardwell-iccrg-bbr-congestion-control-00*, 2017.

[4] E. Dart, L. Rotman, B. Tierney, M. Hester, and J. Zurawski, "The science dmz: A network design pattern for data-intensive science," *Scientific Programming*, vol. 22, no. 2, pp. 173–185, 2014.

[5] J. Crichigno, E. Bou-Harb, and N. Ghani, "A comprehensive tutorial on science dmz," *IEEE Communications Surveys & Tutorials*, 2018.

[6] H. Kim and N. Feamster, "Improving network management with software defined networking," *IEEE Communications Magazine*, vol. 51, no. 2, pp. 114–119, 2013.

[7] M. J. S. Smith, *Application-specific integrated circuits*, vol. 7. Addison-Wesley Reading, MA, 1997.

[8] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[9] M. Budiu and C. Dodd, "The p416 programming language," *ACM SIGOPS Operating Systems Review*, vol. 51, no. 1, pp. 5–14, 2017.

[10] A. Demers, S. Keshav, and S. Shenker, "Analysis and simulation of a fair queueing algorithm," in *ACM SIGCOMM Computer Communication Review*, vol. 19, pp. 1–12, ACM, 1989.

[11] S. Hanke, "The performance of concurrent red-black tree algorithms," in *International Workshop on Algorithm Engineering*, pp. 286–300, Springer, 1999.

[12] M. Shreedhar and G. Varghese, "Efficient fair queuing using deficit round-robin," *IEEE/ACM Transactions on networking*, no. 3, pp. 375–385, 1996.

[13] D. G. Balan and D. A. Potorac, "Linux htb queuing discipline implementations," in *Networked Digital Technologies, 2009. NDT'09. First International Conference on*, pp. 122–126, IEEE, 2009.

[14] A. Saeed, N. Dukkipati, V. Valancius, C. Contavalli, A. Vahdat, *et al.*, "Carousel: Scalable traffic shaping at end hosts," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pp. 404–417, ACM, 2017.

[15] R. Fonseca, G. Porter, R. Katz, S. Shenker, and I. Stoica, "Ip options are not an option," *University of California at Berkeley, Technical Report UCB/EECS-2005-24*, 2005.

[16] R. L. S. De Oliveira, C. M. Schweitzer, A. A. Shinoda, and L. R. Prete, "Using mininet for emulation and prototyping software-defined networks," in *2014 IEEE Colombian Conference on Communications and Computing (COLCOM)*, pp. 1–6, IEEE, 2014.

[17] P. L. Consortium *et al.*, "Behavioral model (bmv2)," 2014.

[18] A. Tirumala, "Iperf: The tcp/udp bandwidth measurement tool," *http://dast. nlanr. net/Projects/Iperf/*, 1999.

[19] B. Hubert, T. Graf, G. Maxwell, R. van Mook, M. van Oosterhout, P. Schroeder, J. Spaans, and P. Larroy, "Linux advanced routing & traffic control," in *Ottawa Linux Symposium*, vol. 213, 2002.

[20] S. Hemminger *et al.*, "Network emulation with netem," in *Linux conf au*, pp. 18–23, 2005.

[21] R. Jain, A. Durresi, and G. Babic, "Throughput fairness index: An explanation," 1999.