Alleviating Congestion via Switch Design for Fair Buffer Allocation in Datacenters

Ahmed M. Abdelmoniem, *Member, IEEE*, and Brahim Bensaou, *Senior Member, IEEE*

***Abstract*—In data-centers, the composite origin and bursty nature of traffic, the small bandwidth-delay product and the tiny switch buffers lead to unusual congestion patterns that are not handled well by traditional end-to-end congestion control mechanisms such as those deployed in TCP. Existing works address the problem by modifying TCP to adapt it to the idiosyncrasies of data-centers. While this is feasible in private environments, it remains almost impossible to achieve practically in public multi-tenant clouds where a multitude of operating systems and thus congestion control protocols co-exist. In this work, we design a simple switch-based active queue management scheme to deal with such congestion issues adequately. Our approach requires no modification to TCP which enables seamless deployment in public data-centers via switch firmware updates. We present a simple analysis to show the stability and effectiveness of our approach, then discuss the real implementations in software and hardware on the NetFPGA platform. Numerical results from ns-2 simulation and experimental results from a small testbed cluster demonstrate the effectiveness of our approach in achieving high overall throughput, good fairness, smaller flow completion times (FCT) for short-lived flows, and a significant reduction in the tail of the FCT distribution.**

***Index Terms*—Data center, congestion control, TCP, NetFPGA.**

I. Introduction

CLOUD computing has dramatically risen in popularity in the past two decades. The special nature of datacenter network traffic brought back to the surface the old tug-of-war problem between short-lived flows' completion times and long-lived flows' throughput [1], [2]. In a nutshell, like recent works [3], [4], [5], our goal in this work is to study, design and implement the means to reconcile the flow delivery timeliness requirements of the former and the high throughput requirements of the latter, under the stringent conditions imposed by datacenter network characteristics. Numerous congestion controllers were proposed for the Internet, high-speed WANs, lossy wireless networks and datacenters, and the interested reader may refer to the following surveys for a broad coverage [6], [7], [8], [9]. Each algorithm aims to improve the way TCP reacts to congestion in a particular network setting.

The co-existence of various flows with different performance requirements, ranging from synchronous short-lived flows to bulky long-lived flows, poses another challenge to TCP in datacenter networks. This can be attributed to several characteristics of datacenter networks, including the high bandwidth-low latency and the use of switches with small buffers instead of routers. These characteristics are at odds with the original design philosophy of the TCP congestion controller. As a result, many complex, non-conventional congestion events (incast congestion [10], [11] and buffer-bloating [3], [12]), which are typically not observed on (or do not cause dramatic effects in) the Internet, appeared in datacenter networks. Also, Ssuch events cannot simply be inferred from packet losses or duplicate ACKs, and hence they require special treatment. For example: 1) *Incast congestion* occurs frequently in datacenters where many loosely time-correlated short-lived flows converge onto the same congested output port of a switch over a short period of time. With the small switch buffers, many of these flows would experience synchronized losses; 2) *Buffer-Bloating* occurs as a result of the normal stabilization process of loss-based TCP congestion control, and as a result, ends up filling up the small switch buffer (even though it is not necessary) [4], [12], [13], [14], [15]. Long-lived flows occupy the small buffer space persistently while short-lived flows fail to grab their share of buffer and experience repeated losses, which ultimately results in significantly long flow completion times. These two problems may coincide due to the co-existence of flows of different natures competing for the same buffer at the bottleneck link.

In this work, we advocate a flow-aware approach similar to traditional flow-based networks (e.g., the available bit rate service in ATM networks (ATM-ABR) [16] or its Internet alter-ego XCP [17]) and its variation RCP [18]). This is also supported by prior work showing that flow control can not be optimally decentralized and requires the assistance of network elements [19]. The challenge that arises however is how to deploy such *flow-awareness in the flow-agnostic and flow-averse IP environment* without modifying the TCP sender and receiver algorithms or code. XCP (or RCP) would have been a good solution, however, the latter requirement immediately disqualifies it, as it is a clean-slate redesign of congestion control that requires not only changes to the routers but also to the sender and receiver. Prior attempts [4], [13] addressed more or less the problem by adopting similar switch-based solutions.

To achieve our goal, we need to update the switch with minimal added complexity and without changing the TCP sender/receiver behaviour. In essence, for the switch to be able

Manuscript received 5 April 2023; revised 6 December 2023; accepted 27 December 2023. Date of publication 23 January 2024; date of current version 8 March 2024. This work was supported by Hong Kong RGC under Grant GRF16209922. Recommended for acceptance by X. Tang. *(Corresponding author: Ahmed M. Abdelmoniem.)*

Ahmed M. Abdelmoniem is with the School of EECS, Queen Mary University of London, E1 4NS London, U.K., and also with CS Department, FCI, Assiut University, Asyut 71515, Egypt (e-mail: ahmed.sayed@qmul.ac.uk).

Brahim Bensaou is with CSE Department, The Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong (e-mail: brahim@cse.ust.hk).

Digital Object Identifier 10.1109/TCC.2024.3357595

© 2024 The Authors. This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

to help in controlling congestion while ensuring fairness, the switch's algorithm must only be able to: i) track a per queue counter of the number of active TCP flows (instead of monitoring the full TCP state); ii) calculate a fair share for each flow that traverses the output link; iii) convey this fair share back to the traffic sources, without the need for maintaining per-flow state information.

To achieve this, in this work, we propose *FairQ* a switch-based algorithm for a fair queue allocation among competing TCP flows. *FairQ* leverages TCP flow control to achieve its goal of maintaining a target buffer occupancy (such that sufficient room is left for incast traffic). Since TCP flow control is an integral part of all TCP variants, *FairQ* is a switch mechanism that requires no change to the network stack at the endpoints (e.g., private virtual machines (VMs) or containers) in datacenters. The algorithm maintains each queue's fair-share window values and conveys them via the receiver window field in the TCP ACK headers. Our contributions are as follows:

- We propose *FairQ*, a novel TCP-independent switch scheme, to achieve fair allocation in datacenters.
- We mathematically model *FairQ* dynamics and show its fast convergence to the operating point as well as its stability.
- We discuss a hardware prototype implementation of *FairQ* on the NetFPGA platform and show that the design imposes minimal increase on FPGA resource usage compared to the reference vanilla-switch design.[1]
- We evaluate *FairQ* via experiments in a small testbed using its hardware prototype switch and show that it significantly improves the FCT performance by up to two order-of-magnitude over the state-of-the-art methods.

The rest of this paper is organized as follows: we give a brief overview of the problem in Section II, then, we discuss the proposed method and present the proposed queue management scheme *FairQ* in Section III. We further develop a simple analytical model of *FairQ* to evaluate its convergence and stability analytically and via simulations in Section IV. We present the hardware prototype design and the testbed implementation and experiments in Sections VI. We discuss some related works in Section VII and summarize our contributions in Section VIII.

II. Background

A. Intra-Protocol Unfairness in Datacenters

By design, for most TCP variants, the coexistence of TCP flows from the same TCP variant should ideally result in a fair competition where each flow can grab an equal share of the bottleneck link capacity [21]. This fairness is also known as the Round-Trip Time (RTT) fairness because it is conditional on the fairness of the RTT: that is, the TCP throughput is inversely proportional to the RTT, and two flows that share a bottleneck can nominally achieve the same throughput if they last long enough and experience the same RTT [22]. In addition, since the Internet-centric design of TCP targets long-term fairness among competing flows, TCP's deployments in datacenter networks have inherited this goal. As a result, short-lived TCP flows, that abound in datacenter networks and do not last long enough to reach the steady state, cannot obtain their nominal fair share and often experience losses leading to long waiting for the retransmission timers to Timeouts (RTO). Short flows would benefit if short-term fairness is achieved. However, it is hard to achieve this for short-delay high-speed networks (e.g., datacenters) with the current TCP design which caters for long-delay low-speed networks (e.g., the Internet) [1], [3].

B. The TCP Flow Control Mechanism

To implement a flow-controlled byte stream reliable data transfer service on top of the Segmented TCP transmission, each TCP end-point reserves, during connection establishment, a buffer for receiving incoming data from its peer. The main goal of this buffer is to simplify the implementation of a distributed flow control by simply ensuring that the sender never overflows the buffer space of the receiver. As such, the outgoing segments awaiting transmission or the arriving segments waiting to be consumed by the application are stored in the send buffer or receive buffer, respectively. To prevent receive buffers from overflowing, TCP provides the means for the receiver to pace the sender rate by controlling the extra amount of data bytes that can be sent by the sender. This is achieved by returning a permissible "window" of bytes with every ACK in a field named "Receiver Window Size" [23]. This field has a width of 16 bits thus allowing barely an increment of 64 KB of data, which was sufficient in the early days of the Internet. Today, TCP includes an option called "window scaling option" [24] that semantically expands the field to 30 bits allowing increments of up to 1 GB.

C. Relationship Between Congestion Control and Flow Control

In addition, to the flow control window, TCP congestion control also uses a window to enforce a calculated limit on the source sending rate based on the currently perceived congestion level. Even though the two mechanisms are considered to be different and target different purposes, functionally they are interrelated as they both are used to limit the TCP source sending rate. Generally speaking: 1) *Flow Control:* adjusts the sending rate of the source to match the available buffer and processing speed of its peer; 2) *Congestion Control:* adapts the sending rate to the congestion state perceived from the network's implicit or explicit feedback. At any instant in time, any TCP flow in the network is limited by either the remaining buffer space of its peer or the current value of its unused congestion window. TCP sets the relation between the congestion window $Cwnd$ and the receiver window $Rwnd$ as follows: $Swnd = \min(Cwnd, Rwnd)$, where $Swnd$ is the sender window.

[1]This work extends and builds on our prior work in [20]. To help interested readers reproduce our results and for openness, we make the code and scripts of our implementations, simulations, and experiments available online (as is) at the following link: **https://github.com/ahmedcs/FairSwitch**.

D. The Role of Active Queue Management (AQM)

AQM algorithms are deployed in switching devices to help in controlling congestion by continuously monitoring the state of the output queues and taking an active role in relieving congestion if necessary. Typically the instantaneous (or average) queue size, arrival rate and/or departure rate are estimated and whenever their value exceeds a certain threshold the algorithm infers (impending) congestion on the link. When this happens, either, the algorithm proactively drops packets as a form of implicit congestion notification to loss-based TCP sources or it sends explicit congestion notification signals to the sources to adjust their sending rates accordingly. A typical example of this is the so-called Random Early Drop with Explicit Congestion Notification (RED-ECN) [25].

III. The Proposed Methodology

TCP is a full-duplex protocol that implements flow control via the so-called Receiver window $Rwnd$ in the ACK headers, in addition to the receiver window scaling option of TCP. The two end-points "agree" *a priori* on an exponent number n, between 0 and 14, that defines a multiplicative factor 2^n applied to the received $Rwnd$ value. Upon receiving such information, the sender calculates the actual receiver window value as $Rwnd << n$. RFC7323 [24] states that the window scaling option can be set during the connection establishment phase in the SYN packets. In our approach, we propose to migrate the congestion controller to the switch and piggyback it on the flow control window. That is, to achieve the same expected effects as those achieved normally by end-to-end flow control and congestion control via the sender window setting: $Swnd = \min(Cwnd, Rwnd)$, and without maintaining per-flow state information at the switches, it is sufficient for the switch to overwrite the $Rwnd$ field in the TCP ACK headers to indicate the bottleneck fair share of the bandwidth and buffer available for each flow sharing the same output port. As the ACKs traverse the switches in the reverse path towards the sender, each switch examines the ACK and modifies the $Rwnd$ value after adjusting it, using the window scaling which can be carried in the four reserved bits in the TCP segment header to avoid maintaining flow state information at the switch.

A. System Design and Algorithm

The main variables and parameters used in *FairQ* algorithm are described in Table I. Note that T, M and α are parameters of the algorithm that can be chosen by the administrator. Algorithm 1 is shown as a set of event handler functions in an event-driven environment. It runs on the switch and responds to two major events: packet departures, and timer-based local window update events.

Upon a Packet P Departure: the algorithm updates the local maximum packet size ($LMSS$) seen so far (Line 2). If this is the only flow (i.e., $\beta \leq 0$) to just arrive at this port, then the current window $LRwnd$ is initially set to the target queue worth of bytes (Lines 5 and 14) then *FairQ* enters the slow-start phase to start probing for the effective window size (Lines 6 and 15). This is because initially the end-to-end bandwidth-delay product (BDP) is unknown to the switch and hence the available bandwidth has to be probed. Subsequently, for each new flow, the current value of $LRwnd$ is re-scaled to share the bandwidth equally among all the flows (Lines 8 and 12). If the ACK bit is set, the receive window field in TCP header $Rwnd(P)$ is re-scaled by the scale factor $Reserved(P)$, then compared to the current local window value $LRwnd$ of the corresponding forward path queue (Line 16). If $LRwnd < (Rwnd(P) << Reserved(P))$, then this packet is updated with the current local window $LRwnd$ after scaling by the scale factor (i.e., $Rwnd(P) \leftarrow LRwnd >> Reserved(P)$) (Line 17). Note that, in Section V-A, we show how the scale factor can be encoded into the reserved bits of the TCP header (i.e., $Reserved(P)$) by a hypervisor-level shim-layer running on the end-host.

TABLE I

Variables and Parameters of *FairQ* Algorithm 1

Parameter name	Description
T	Timeout value for each window increment interval
M	Number of increment intervals to wait for an update
B	Buffer size of the bottleneck link on the data path
α	Target level (or threshold) of the queue occupancy
Variable name	Description
$LRwnd$ ($LMSS$)	Local receive window (maximum segment size) values maintained by the switch for each queue
β	Number of current ongoing flows
γ	Window increments of one update interval
Γ	Counter of the number of increments
Q	Current output queue length in bytes
κ	The drift of Q from the target αB
P	A packet
$Rwnd(P)$	The value of receive window in the TCP header
$Reserved(P)$	The value of reserved bits in TCP header
$slow_start$	The current state flag

Upon Window Update Timer Expiry: κ is calculated to track the deviation of the current queue length from the target (Line 19). This is used to control the fraction of MSS added or subtracted from the current value of γ (Line 20). After a number M of updates to γ (Line 22), the current value of the local per-queue window ($LRwnd$) is updated (Line 23-24). Specifically, if slow start is active, *FairQ* adds two MSS to the window to ramp up $LRwnd$ fast and reach the per-flow fair share, otherwise, it adds the current value of γ to the per-queue local window $LRwnd$. Specifically, for the first flow, the receive window is set to the target queue occupancy (Line 6) then the slow start is activated (Line 7). Then, the window is increased by 2MSS gradually to account for the bandwidth-delay product. The slow start phase ends when the queue hits the target (i.e., $\alpha \times B$) (Line 25). Notice that the value of the window increment is updated M times before these updates are reflected in $LRwnd$ and thus in the actual value of the TCP receive window $Rwnd(P)$ that is conveyed to the TCP sender.

The design of *FairQ* enables it to maintain a low loss rate by keeping enough buffer space available to absorb sudden traffic bursts (e.g., incast) while keeping the links fully utilized. It embodies two principles, i) a congestion controller that adopts a proportional increase, proportional decrease approach where the window is expanded/shrunk in proportion to the level of

Algorithm 1: Fair-Share AQM (FairQ) Algorithm.

1 **Function** *Packet Departure Event Handler* (P)
2 **if** $LMSS \leq TCPSize(P)$ **then**
  $LMSS \leftarrow TCP_Size(P)$;
3 **if** $SYN - ACK(P)$ **then**
4 //Update for both incoming and outgoing queue;
5 **if** $\beta \leq 0$ **then**
6 $LRwnd \leftarrow \alpha \times B$;
7 $slow_start \leftarrow True$;
8 **else**
9 $LRwnd \leftarrow LRwnd \times \frac{\beta}{\beta+1}$;
10 $\beta \leftarrow \beta + 1$;
11 **if** $FIN(P)$ **then**
12 //Update variables for only the outgoing queue;
13 $\beta \leftarrow \beta - 1$;
14 **if** $\beta \geq 0$ **then** $LRwnd \leftarrow LRwnd \times \frac{\beta+1}{\beta}$;
15 **else**
16 $LRwnd \leftarrow \alpha \times B$;
17 $slow_start \leftarrow True$;
18 **if** $ACK(P)$ && $(LRwnd \leq Rwnd(P) << Reserved(P))$ **then**
19 $Rwnd(P) \leftarrow LRwnd >> Reserved(P)$;
20 **Function** *Window Update Timer*
21 $\kappa \leftarrow 1 - \frac{Q}{B \times \alpha}$;
22 $\gamma \leftarrow \gamma + \frac{\kappa \times LMSS}{M}$;
23 $\Gamma \leftarrow \Gamma + 1$;
24 **if** $\Gamma == M$ **then**
25 **if** $slow_start == True$ **then**
  $LRwnd \leftarrow LRwnd + 2 \times LMSS$;
26 **else** $LRwnd \leftarrow LRwnd + \frac{\gamma}{\beta}$;
27 **if** $Q \geq \alpha * B$ **then** $slowstart \leftarrow False$;
28 $\gamma \leftarrow 0; \Gamma \leftarrow 0$;

deficit/excess buffer occupancy with respect to a buffer threshold, and thus congestion; ii) a fairness controller that divides the amount of increase or decrease equally among all ongoing flows. This makes it appropriate to handle well the co-existence of short-lived and long-lived flows. Each switch port is associated with a nominal window variable $LRwnd$. Initially and whenever the number of ongoing flows drops to zero, the algorithm goes into the slow start mode, where this window is incremented by two MSS after the end of each update period. When the queue exceeds the target occupancy, the algorithm goes into congestion avoidance and this window is decremented in proportion to the backlog in excess of the target queue occupancy.

B. Practical Aspects of the System

Flow Tracking: In principle *FairQ* is very effective in solving the problem of congestion, and actually avoiding it outright. However, to enable its successful practical deployment, the following requirements need to be met: i) the ACKs must travel back along the reverse path taken by the corresponding data packets, as they are used for switch explicit rate signals, ii) the switch must be able to track the number of ongoing flows to enable fair sharing; and, iii) as the ***Rwnd*** field of the TCP header is used for signalling, the algorithm must take into account the possible use of the window scaling option for each ongoing flow to avoid semantic mismatches between the receiver and the switch in interpreting the $Rwnd$ values. To achieve the first requirement (i), two approaches are possible: either implement flow-aware routing in the open source network OS of the bare-metal switch or, more likely, since SDN-based switches are more common nowadays in datacenters, one can rely on the functions already provided by SDN to setup flow-based routing. To fulfill the second requirement (ii), one can implement a SYN/FIN-based counting using hardware registers in the switch, or again relying on an SDN controller to track the number of active flows via a special OpenFlow rule on SYN/FIN packets; we implemented and tested the first approach in a NetFPGA platform and deployed the second in an SDN-enabled testbed.[2] To meet the third requirement (iii), we can rely on assistance from the end hosts as described next.

TCP Window Scaling: The TCP window scaling option remains an important issue. In practice, this option is supposed to be activated to deal with long-fat pipes by increasing the receiver window from 64 KB per flow to up to 1 GB per flow. However, even in current low-latency DCNs with 10-100 Gbps interfaces, the scaling remains a necessary element and the receive window still needs to be scaled to maintain full link utilization. Even though, one could argue that the chances of having a single flow active on a given port are close to nil (considering the average number of flows per server in a private DCN measurement is about 36 [3]), a robust technique should provide the ability to rescale receive window values. According to the RFC [24], the window scaling option is negotiated between the sender and the receiver and to enable it, both sender and receiver must send their window scaling option in the SYN segment and its corresponding SYN-ACK. However, in practice, the scaling value is not negotiated as different TCP implementations adopt different default values for the scaling factor. For example, by default in MacOS the scaling exponent is set to three while Linux calculates it according to the allocated receiver buffer size. Furthermore, these values can be reconfigured by the application to be from 0 (i.e., up to 64KBytes for no-scaling) to 14 (i.e., up to 1 GBytes with full scaling). To avoid any cognitive mismatch between the values set by *FairQ* in the receive window field and those interpreted at the receiver, and to operate regardless of the link speed used in modern datacenters, the following are the possible ways to solve this issue without modifying TCP in the VM/container:

- if the scaling option is negotiated then we propose to simply unify the value supported in the DCN by rewriting it in the SYN and its corresponding SYN-ACK during the phase

[2]We note that other recent works have implemented various AQMs on P4 Tofino switch [15], [26] and therefore we believe, owning to *FairQ*'s simplicity, it could be easily implemented on programmable switches and NICs.

of TCP connection establishment via an SDN rule in the switches or directly by *FairQ*;

- if the TCP implementation informs the peers of the scaling value during connection setup, then we propose to have a lightweight shim layer at the end-hosts. This shim-layer tracks the per-flow scaling factor recomputes the receive window of outgoing ACKs and resets the value to a pre-set network-wide scaling factor which is already configured on all the switches; and

We adopt in our prototype (see Section V) a module (or shim-layer) in the server (or hypervisor) to store the scaling factors of the flows. This factor is communicated to the other side of the TCP connection within the SYN packet. The module leverages four of the reserved bits in the TCP header to encode the scaling factor on the outgoing ACKs. The switches use the encoded value to adjust the window value before it is updated in the TCP header. Therefore, this approach does not require any changes to the network stack and is transparent to the VM's guest OS.

Role of SDN: In the SDN approach, SDN's capability to track flows, flow statistics and the scaling value sent in the SYN segments can be easily invoked to address the three requirements above. In contrast, if SDN is not available and the network elements are not prone to upgrade, additional knowledge of the DCN architecture and routing can enable the DCN operator to easily deploy *FairQ*. For example, if single-path routing is used, the learning ability of the switches can be invoked to implicitly assume that forward and reverse paths are already the same. If Equal-Cost Multi-Path (ECMP) routing is used a simple modification to the *FairQ* algorithm to equally divide the flow fair share among the multiple routing paths is easily applied. In addition, to track the number of active TCP flows, we can simply implement an efficient header filter to track SYN/FIN flags for connection establishment or tear-down without per-flow state by using per-port registers.

Processing Complexity: In terms of processing complexity, *FairQ* is a simple algorithm with very low complexity because it requires $O(1)$ computation per packet. This makes it ideal for integration into new switches or routers. For example, it can be implemented in Linux-based routers as a module using the NetFilter framework. Netfilter [27] allows for modifications to the packet headers prior to their forwarding by the IP layer. *FairQ* can also cope with Internet checksum recalculation efficiently after header modification, by applying a straightforward one's-complement add and subtract operations on the following three 16-bit words [23]: $CSum_{new} = CSum_{old} + Rwnd_{new}(P) - Rwnd_{old}(P)$. In addition, since *FairQ* is designed to deal with TCP traffic, tracking the number of flows can be achieved in a scalable manner by monitoring SYN/SYN-ACK and FIN/FIN-ACK bits which is $O(1)$.[3] All in all, the operations of *FairQ* are $O(1)$.

[3] This approach may result in unfair allocation with silent flows. Alternatively, the switch can estimate the number of active flows via flow-state information. Both solutions have drawbacks but maintaining flow-state information at the switch makes the solution less practical [4], [13], [17]. To handle silent flows, our shim-layer could maintain a per-flow timer and generate dummy FIN (and SYN) to reduce (and increase) the number of active connections at the switches, dummy FIN and SYN being then discarded by the receiving end host before reaching TCP. A similar approach was adopted in [28], [29].

Effect on Internet-Facing TCP Connections: It is worth mentioning that, the WAN connections of datacenters to the Internet in most cases are facing intra-datacenter load-balancers and proxies that split the TCP connection. Hence, TCP connections inside the datacenter are effectively separated from the TCP traffic from outside the Internet. This avoids the possible issues of updating the receive window of the Internet-facing TCP connections which run in a high Bandwidth-Delay Product (BDP) environment like the Internet.

IV. Convergence and Stability Analysis

Since *FairQ* adopts a proportional increase and proportional decrease approach to adjust its window, it is important to study its convergence and stability. In the following, we model *FairQ* behavior by considering the three parts that make up the system: the switch queue behavior, the switch's per-queue local window updates, and the source's window adjustments in response to the switch feedback updates.

A. Mathematical Modelling

Similar to [30], [31], we adopt a fluid approach to model how *FairQ* reacts proportionally to the extent of congestion and how it updates the local $LRwnd$ value at a predetermined constant interval in the switch before conveying its value in the incoming ACKs to the sources. This leads us to a model that is centered around the switch where all calculations are based on the advances in time. Recall that T is the interval duration of the increments. We let the target queue occupancy be $\alpha \times B$ where B is the buffer size and α is the target threshold as defined in Table I. At the start of *FairQ*'s operation, the local window size denoted $w(t)$ in the switch is initially set to $\alpha \times B$ bytes. We further model the window dynamics using a discrete-time model with respect to T, then the window dynamics, $w(t)$, can be expressed as

$$w(t) = \begin{cases} w(t-T) + \gamma(t) & \text{if } t = kMT, \\ w(t-T) & \text{otherwise,} \end{cases} \tag{1}$$

where k is a positive integer which tracks the window update epochs and $\gamma(t)$ is the average value of $\kappa(t)$ over the different increment intervals expired during one update interval. Simply put, $\gamma(t)$ is the number of MSS by which the window should be increased or decreased in the update interval ending at t. Note that, $\gamma(t)$ is reset at the end of each update epoch (i.e., at time $t = kMT$). Hence, $\gamma(t)$ can be expressed as (2) shown at the bottom of the next page.

Notice that instead of reducing the queue dynamics in the update interval to the final value only, our calculation of $\gamma(t)$ takes into account the past queue fluctuations from the start of the interval by averaging all M sampled values. Let the link speed (or capacity) be C, then the queue dynamics can be described as

$$q(t) = \left[q(t-T) + \frac{T}{RTT} w\left(t - \frac{RTT}{2}\right) - CT \right]^+ ; \tag{3}$$

that is, the queue at time t receives a window-worth of bytes that were calculated half an RTT earlier (to account for the

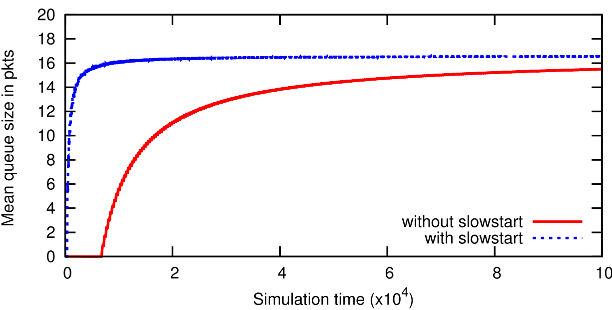


Fig. 1. Stability and convergence speed of the system described by (1), (2) and (3) where the x-axis is in RTT units (i.e., 100 μs).

propagation time of the ACK from the switch to the source and the propagation of the data from the source to the switch, arriving at time t). For simplicity, we assume there is no congestion and the RTT fluctuates slowly, hence the ACKs do not queue up in the reverse direction.

Then, it is expected that the persistent queue converges to $\alpha \times B$ as t goes to infinity. To support this claim, we use the above model and run numerical experiments with the system model described by (1), (2) and (3) in Matlab. In the experiments, we assume traffic sources are connected to one switch with a buffer size of 83 packets and the target queue occupancy is set to $\alpha = 20\% = 16.6$ packets. The capacity of the output link is 10 Gbps and the RTT is 100 μs. We run two scenarios, one with the slow start enabled and one without.

Fig. 1 shows the mean queue size over time, which is displayed in increments of RTT (i.e., 100 μs), for the two scenarios (i.e., with and without slow start) as obtained from the Matlab simulations. The graph supports our intuition that, in the beginning, the mean queue occupancy remains at the zero level until the pipe is filled. Then, it increases steadily until it converges to the target queue occupancy as time goes to infinity. In addition, a slow start seems to improve the speed of convergence dramatically. A slow start leads the queue occupancy to the target very fast, and then the proportional increase proportional decrease mechanism maintains the window around the target queue occupancy while reacting with agility to congestion. Hence, *FairQ* enjoys a fast convergence speed and is expected to operate the queue around the target queue level. The system is also expected to be stable as it can reject any external transient disturbance (e.g., new flow or incast event) and return quickly back to the steady state operating point (i.e., the target queue level α).

B. Simulation Analysis

We then study the convergence properties of *FairQ* via ns2 simulation in network scenarios with low BDP (i.e., datacenter networks). We compare *FairQ* to the state-of-the-art TCP variant for datacenters (i.e., DCTCP). The simulation results demonstrate *FairQ* can outperform DCTCP over convergence speed to steady state and fairness among competing flows.

For *FairQ*, the values of α, T and M are chosen based only on the target level of congestion that can be tolerated regardless of the capacity, delay, and the number of sources. In the simulation experiments, we set α to 20% (of the buffer size), T to 50 μs and M to 10 intervals leading to an update epoch every 500 μs. DCTCP parameters are set according to their recommended settings with K (the target queue occupancy) of DCTCP set to 17% of the buffer size.

We use a single rooted-tree (i.e., Dumbbell) topology and run the experiments for a period of 1 sec. The buffer size of the bottleneck link is set to the value of the BDP in all cases (e.g., 83 Packets or 125 KBytes), and the IP data packet size is 1,500 bytes. We use high-speed links of 11 Gbps for the sending stations in our simulation experiments, a bottleneck link speed of 10 Gbps, a low RTT of 100 μs and a RTO_{min} of 2 ms as opposed to the default 200 ms in real implementations.

We simulate a scenario with 5 long-lived flows that start and stop each in a predetermined order. Fig. 2(a) and (b) show the goodput achieved by the 5 long-lived flows. The results show that compared to DCTCP, *FairQ* is able to converge faster to the fair-share for each newly active flow and achieves better short-term (or instantaneous) fairness with a lower variance.

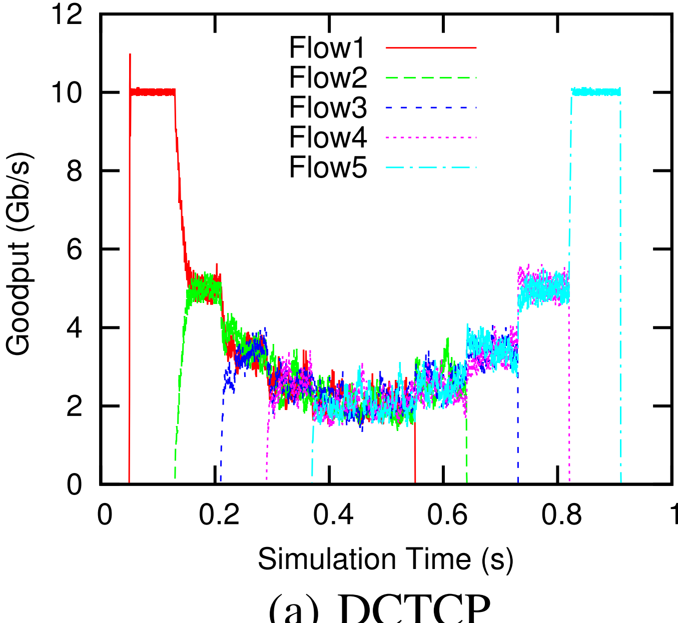
V. System Implementation

DCN operators use commodity Ethernet switches with small buffers for interconnecting the servers mainly because of their low cost and ease of deployment. Typically, the in-network switches are under the control of the datacenter operator. These switches have a certain lifetime (and even worse they frequently fail, long before that) after which they need to be replaced just like any other hardware equipment. Hence, we believe that solutions involving very simple modifications to the switching chips or firmware are feasible. Switching chip manufacturers would be motivated by marketing efficient datacenter-tailored chips that involve minimal hardware/software updates in favor of performance gains.

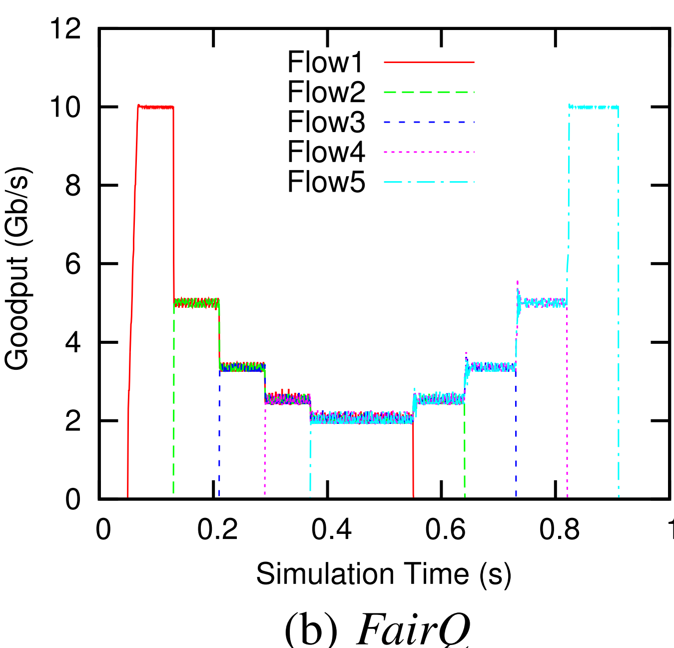
In the following, we present the details of software and hardware involved in the prototyping of *FairQ*. The goal is to fulfill the following: (F1) Improving latency-sensitive applications Flow Completion Time (FCT) and mitigate incast; (F2) Maintaining high sustained throughput for long-lived flows in a work-conservative manner; (F3) Achieving this without imposing modifications on the existing TCP protocol stack or anything that is controlled by the tenant;[4] we then evaluate *FairQ* via a real deployment in a small testbed.

[4]If changes to existing systems are needed (i.e., inevitable) then they must be in network devices and/or hypervisors that are fully under the control of the DCN operator

$$\gamma(t) = \begin{cases} 0 & \text{if } t = kMT, \\ \frac{MSS}{M}\left(1 - \sum_{j=1}^{M} \kappa(t - jT)\right) = \frac{MSS}{M}\left(1 - \sum_{j=1}^{M} \frac{q(t-jT)}{\alpha B}\right) & \text{otherwise.} \end{cases} \quad (2)$$



(a) DCTCP



(b) *FairQ*

Fig. 2. Goodput of 5 flows that start/stop in a predetermined order showing the convergence speed to the fair-share.

The principles discussed above in the design of *FairQ* can be implemented in many different ways (e.g., in a virtual software switch such as OpenvSwitch (OvS) using virtualization or in firmware on bare metal switches, and so on). In this paper, we discuss the implementation of *FairQ* as a two components software/hardware implementation which we name hereafter FairSwitch (or FS in short) as shown in Fig. 3: the first component in FS (the software part) is implemented under the hypervisor at the physical servers and consists in a shim-layer whose role is to maintain the consistency of the receiver window fields with respect to their window scaling factors. The second component

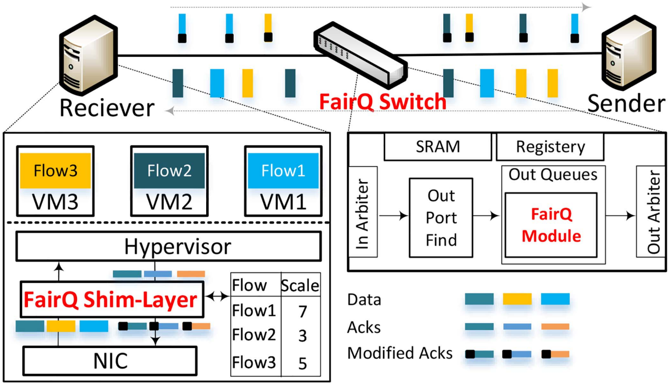


Fig. 3. FairQ switch: a software/hardware incarnation of *FairQ*.

is implemented in the switch and deploys the logic that estimates the equal fair share, and modifies the receiver window field in the outgoing ACKs to enforce rate adjustments in a fair manner to regulate the queue at a certain target occupancy by leveraging the TCP flow control mechanism to control TCP flow rates without interfering with their current "congestion" window update function.

Fig. 3 shows how the FairQ switch could be deployed in a datacenter. The *FairQ* switch module computes the fair share associated with the output port buffer and updates the receiver window of all outgoing ACKs to assign that fair share to all flows that share this buffer. To take into account the scaling factor when updating the corresponding window values yet avoid maintaining per-flow state information, the scale factor is written in the reserved bits of the outgoing TCP ACK packet headers by the FairQ shim-layer in the host and is applied to the new window value by *FairQ* switch module on the fly so that the assigned fair share is interpreted correctly by the ACK receiving end-point (i.e., the TCP sender). The switch module monitors the queue occupancy of the per-port output queue and the occurrence of special packets such as SYN-ACK/FIN-ACK. The switch actively calculates a local per-port window value based on the deviation from the target queue occupancy. The local window value is re-calculated each time a new flow is observed or an existing flow leaves the queue.

A. An End-Host Helper Module for Window Scaling

The switch requires the window scaling factor to rescale the local receive window values before updating the TCP header of the ACKs. RFC7323 [24] states that the three-byte scale option may be sent in an SYN segment by each TCP end-point, otherwise it is ignored. Its purpose is to inform the peer of the value of the exponent it is using to scale the offered receive window. This option is unnecessary for networks with a BDP of 31.25 Kbyte (i.e., a bandwidth of 1 Gbps and RTT of 250 μs). However, with the introduction of high-speed links of 40 Gbps (i.e., BDP=1.25 Mbyte) and 100 Gbps (i.e., BDP=3.125 Mbyte), the scaling factor becomes necessary to utilize the bandwidth when the number of flows is small.

To avoid scalability issues with flow-level state tracking, we propose the adoption of a lightweight end-host/hypervisor shim-layer to explicitly append the scaling factor to all outgoing ACKs. The shim-layer extracts from outgoing SYN and SYN-ACK packets the advertised scaling factor (i.e., within the scaling option) for each established TCP flow at connection setup, hashes the flows via the 4-tuple (including IP addresses and port numbers) into a hash-table in which it stores the scaling exponent. The entries in the hash table are cleared when a connection is explicitly closed (i.e., FIN is sent out).

The module writes the exponent of the scale factor for all outgoing ACK packets in the 4-bit reserved field of TCP headers.[5] Regardless of the method used to carry the exponent bits, these modified TCP bits are cleared by the shim-layer once used, to avoid the packet being dropped by the destination due to an invalid TCP checksum value. This saves the recalculation of the TCP checksum by the switches along the path.

Typically, this module resides either above the NIC driver for a non-virtualized setup, or below the hypervisor to support VMs/containers in cloud datacenters or finally could be implemented in the data-path of the NIC (e.g., FPGA or Smart NICs). Hence, this module does not affect any network protocol stack implementation of the guest OS in the VM/container, making it readily deployable in production datacenters.

B. Hardware Prototype

We discuss hereafter the hardware design aspects of FairQ switch (called FairSwitch in the sequel) on the NetFPGA platform. FairSwitch is a hardware realization of a simple AQM mechanism based on the ideas of the *FairQ* algorithm discussed above. As such, 1) when congested, it has to share the throughput among competing (distributed) TCP flows equally; 2) it must only rely on universally standardized mechanisms for congestion or flow control to throttle the senders; and 3) it must not require heavy computational overhead to intervene at the line speed. FairSwitch achieves *FairQ*'s objective of maintaining a target Q_Target occupancy to reduce the latency for short-lived TCP flows with minimal impact on the throughput of long-lived flows. This is achieved by implementing a module in the switch which tracks the per-queue active flows counters by observing the flags in TCP headers and uses built-in implementations of the division operator to calculate the local per-queue receive window. These two functionalities can be offloaded to an SDN controller that uses readily available OpenFlow per-queue switch statistics and a rule to duplicate SYN/FIN packets for active flow tracking. The controller can calculate the per-queue window and set a rule in the switch to update ACKs with the calculated window.

[5]Since the exponent is limited to a maximum value of 14 these 4 bits are sufficient. Alternatively, to avoid any collision with other protocols that might be using the reserved 4 bits, the 16-bit window field can be used: with 4-bits for the exponent and the remaining 12 bits for window values which allows values of 256 MB which is orders of magnitude larger than the BDP in existing datacenter networks. It is also possible to resort to using the checksum field of IP header which in modern networks is ignored.

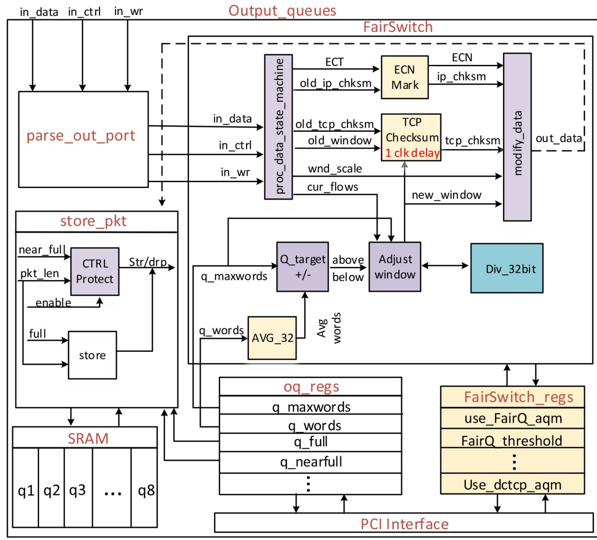


Fig. 4. FairSwitch NetFPGA-based system design.

C. NetFPGA Design and Implementation

The hardware implementation of FairSwitch was developed on the NetFPGA-1 G platform using "Verliog" hardware description language and is based on the reference switch project [32]. NetFPGA processes data in words consisting of 64 bits (i.e., 8 bytes) and spends 1 clock cycle on each word.

1) Fair_Switch Module: As shown in Fig 4, the "Fair_Switch" module is part of the top "output_queues" module. The main inputs to the module are the incoming packet on the in_data line, the control information on the in_ctrl line and the data_ready signal on the in_wr line. These are fed in from the output_queues module and are allocated SRAM space for the destination queue (i.e., q_maxwords) and the number of words currently used by stored packets (i.e., q_words which are fed by oq_regs module). The main outputs are the (un)modified packet information (i.e., out_data, out_ctrl and out_wr) which is written into the FIFO queue of the destination port. The module uses helper modules (i.e., "avg_32", "div_32", "TCP_checksum" and "ECN_mark").

The module tracks separate per-output-queue counters for SYN/SYN-ACK, FIN/FIN-ACK and RST packets. The counters are updated via a state machine to process the headers coming in the in_data input. The module identifies a word as part of the packet header or NetFPGA metadata when in_wr is active and in_ctrl is disabled. Then, the following cases could occur:

- whenever a SYN-ACK is received, the active flow counter is incremented for both the incoming and outgoing queues.
- whenever a FIN-ACK or RST packet is received, the active flow counter is decremented for the outgoing queue.

Note, the counter is reset whenever no packets are received from any of the active flows for a long period of time (i.e., 1 sec which is suitable time-out for intra-DC traffic).

The module monitors whether the safe threshold (i.e., target queue occupancy) has been exceeded for any output queue. This can be evaluated via the average number of occupied words, and

the maximum number of words in the queue right-shifted by the target threshold.[6] If the queue occupancy is above or below the threshold, the local window of the queue is readjusted following Algorithm 1. Notably, whenever the variable cur_flows is updated (i.e., the number of active connections has changed), a new window is produced. The new window value is calculated as the target occupancy (i.e., q_maxwords $\geq$ *FairQ* _threshold) is divided by cur_flows.

Finally, before the departure of each ACK, the receive window field of the TCP header is updated before the packet is forwarded to its output queue (port).[7] This happens only if the window is less than the receive window in the TCP header. Note that the threshold should be set to maintain a nearly empty queue to absorb bursts of short-lived flows or incast traffic while keeping the link fully busy with long-lived flows.

2) Helper Modules: The FairSwitch module relies on the following helper modules:

1) *AVG_32*: calculates a running average of the number of words used for each output-queue. It averages the last 32 samples of the q_words every avg_queue_time. The avg_queue_time is a configurable timer. The default value is set to 48 μs which corresponds to the transmission time of four packets of 1500 Bytes onto a link of 1 Gbps.
2) *TCP_checksum*: calculates a new TCP checksum using the incremental update [33] method shown in line 13 of Algorithm 1. It takes the old and new windows, and the current checksum as input and outputs the new checksum value in the next clock cycle. Note that, because of the one clock cycle delay, the in_data is also delayed by one clock cycle for writing the new checksum into the correct word of the out_data.
3) *ECN_mark*: implements the DCTCP AQM marking. Whenever the maximum queue occupancy limit (by default set at 25%, which is close to the advised 20% value) is exceeded, the packets are marked. Additionally, the incremental update [33] method is used to generate a new IP checksum using ECT (01 or 10), ECN (11), and the previous checksum value. In this case, no delay is added because the IP checksum is one word after the TOS field.
4) *DIV_32 b*: implements the division for window calculation. It divides the left-shifted q_maxwords by *FairQ* _threshold over cur_flows (i.e., fair allocating the buffer among concurrent flows). This module is generated via the Xilinx IP cores library. The division process takes 18 clock cycles to complete but there is no need to add an extra 18-cycle delay to in_data. This is because the new window is updated with SYN/FIN occurrences but is not used until the ACKs arrive in the backward path after approximately half an RTT from the connection setup.
5) *FairSwitch_regs*: maintains FairSwitch parameters (stored in registers) and is responsible for communicating with the Peripheral Component Interconnect (PCI) interface.

[6]Right shifting by x is equivalent to multiplying by 2^{-x}

[7]The new window is right shifted by the "window scale" value which can be extracted from the reserved-bits field [23] of the ACK header

TABLE II
RESOURCE USE BY THE REFERENCE SWITCH VERSUS FAIRSWITCH

	Reference Switch		FairSwitch		NetFPGA Total
	Used	Percentage	Used	Percentage	Available
Slices	9605	40%	10851	45%	23616
Slice Flip Flops	8364	17%	9419	19%	47323
4 input LUTs	14067	29%	15985	33%	47232
BRAMs	22	9%	22	9%	232
Bonded IOBs	360	52%	360	52%	692
CGLKs	8	50%	8	50%	16
DCM	6	75%	6	75%	8

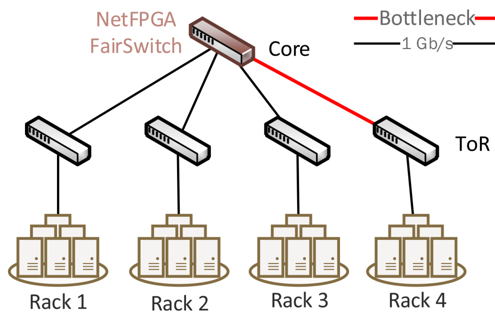


Fig. 5. Testbed setup for FairSwitch evaluation

Fig. 4 also shows that the FairSwitch module interacts with the other modules. The destination port information is first parsed and extracted by parse_out_port module from the control headers which is added by output_port_lookup. Then the packet is processed by FairSwitch and its output (i.e., out_data) is sent to store_pkt module for storing in the SRAM for transmission later or dropping the packet if it can not be stored.

Finally, to assess the complexity of our module in terms of resource usage, Table II highlights the increase in the resources between the reference switch design and the modified one that includes the FairSwitch module.

VI. EVALUATION

Testbed Setup: In the next set of experiments, we deploy the NetFPGA-based FairSwitch in a small-scale real testbed. The testbed consists of 28 virtual servers, each server is associated with a physical dedicated 1 Gbps Network Interface Card (NIC). The serves are a set of high-performance Dell PowerEdge R320 machines. The machines are equipped with an Intel Xenon E5-2430 6-cores CPU, 32 GBytes of RAM and Intel I350 server-grade 1 Gbps quad-port NIC. As shown in Fig. 5, the servers are organized into 4 racks (each rack is a subnet) and connected via 4 non-blocking Top-of-Rack (ToR) switches and the NetFPGA FairSwitch serves as the core switch of the network. The 4 racks are divided into (racks 1, 2 and 3) which are designated as senders and rack 4 which is designated as the receiver. Each 7 out of the 28 ports belonging to the same subnet is connected to one of the non-blocking ToR switches through 1 Gbps Ethernet links. The base RTT in the network is $\approx$200-300 μs. The servers run Linux Operating System (i.e., Ubuntu distribution) whose Kernel has by default the implementations of DCTCP [34], Cubic and New-Reno (abbreviated to Reno) congestion control mechanisms. We

also run on all end-hosts the shim-layer which is implemented as a NetFilter-based loadable Linux kernel module [27].

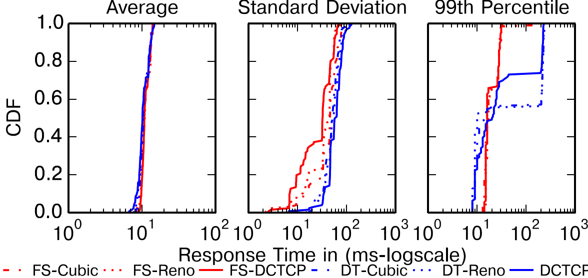
For experimentation purposes, the machines are installed with the *iperf* [35] for creating long-lived flows and Apache benchmark [36] for creating short-lived flows. We setup different scenarios to reproduce both "incast" and "incast with buffer-bloating" situations on the bottleneck link connected to the receiving rack 4. To emulate virtual guests and increase the number of senders dramatically, senders are attached to multiple virtual ports at the end-hosts on OpenvSwitch (OvS) [37]. In this case, each iperf or Apache client/server process is directly associated with one of the virtual ports of OvS. This allows us to emulate traffic originating from any number of VMs and simplifies the creation of scenarios with a large number of flows in the network. The objectives of the experiments are to verify the ability of FairSwitch: i) to support more TCP connections and with high link utilization; ii) to equally and fairly allocate bandwidth among conflicting short-lived and long-lived TCP flows; iii) to improve the FCT of the time-critical short-lived flows and by how much.

A. Incast Without Background Traffic

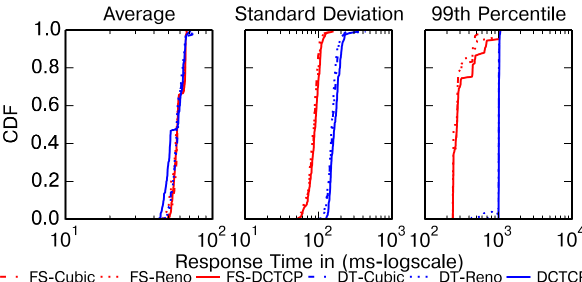
We run two scenarios with mildly and heavily loaded incast where a large number of short-lived flows request a large content divided into 11.5 KB chunks.

Experimental Setup: In both scenarios, each of the 7 servers in rack 4, issues web requests via the Apache benchmark for a *"index.html"* page of size 11.5 KB from each of the 21 (7×3) servers in racks 1, 2 and 3. Hence a total of 126 synchronized requests are issued (i.e., each server in rack 4 makes 21 requests to the servers in racks 1, 2, and 3). The requests within the same subnet are avoided so the number of transfers is 3 racks $\times$ 7 src $\times$ 6 dest (not in the same subnet) $=$ 126. In the mildly loaded scenario, each request is repeated a thousand consecutive times by Apache benchmark which is equivalent to the transfer of 11.5 MByte file for each requester (11.5 MB $\times$ 126 $\approx$ 1.5 GBytes total transfer through the bottleneck link). In the heavily loaded case, we repeat the same experiment with a thousand consecutive requests however, we use now five parallel TCP connections instead of just one. This is the same amount of transfer through the bottleneck link within the same period as in the mild load but with more concurrent connections. Note that Apache benchmark, at the 0thsec, starts requesting the web page 1,000 times then it reports different statistics over all the requests.

Experimental Results: Fig. 6 shows that, under heavy load, FairSwitch achieves a significantly improved performance for TCP flows in terms of the FCT variation from average and the FCT of the tail-end flows. Even though *FairQ* still improves both metrics in the mild case, the performance gains are not significant compared to the heavy load scenario. The competing short-lived flows benefit under FairSwitch in the mild case by achieving almost the same FCT on average but with an order-of-magnitude, smaller standard deviation compared to TCP (Cubic, Reno) with DropTail and DCTCP. In addition, the FCT of the tail-end (i.e., the maximum FCT) is improved by two orders of magnitude suggesting that almost all flows (including tails) can meet their deadlines. Moreover, Fig. 6 shows that more gains are obtained in the heavy load experiment thanks to the agility and fast convergence of *FairQ* (or its FairSwitch incarnation). Finally, the results suggest that FairSwitch can help incast traffic grab their fair-share quickly thanks to the drop rate which is significantly reduced. Fig. 7 shows that the number of drops during the incast events is reduced by $\approx 55\%$ and $\approx 73\%$ in the mild and heavy scenarios, respectively.

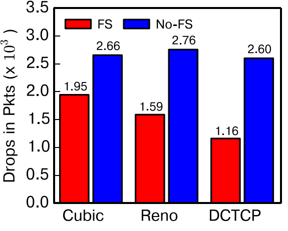


(a) Medium incast scenario: 126 concurrent short-lived flows

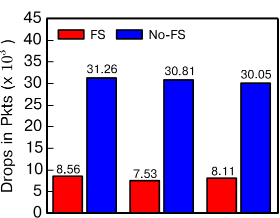


(b) Heavy incast scenario: 630 concurrent short-lived flows

Fig. 6. Incast Scenario: Average, Standard Deviation and Max FCT for TCP with FairSwitch versus DropTail versus DCTCP. Each flow requests 11.5 MB file (divided into 1000 11.5 KB blocks).



(a) The medium load case

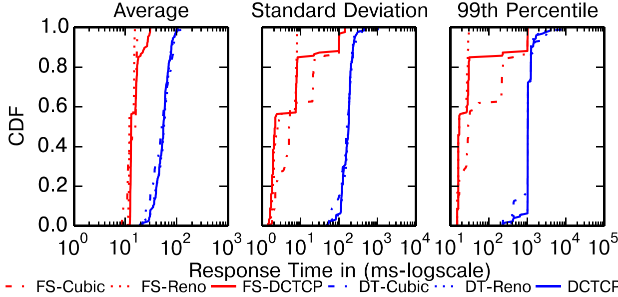


(b) The heavy load case

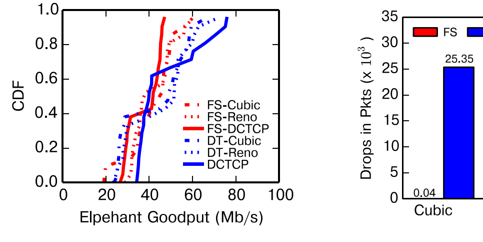
Fig. 7. Incast Scenario: Total Packet Drops during the experiment for TCP with FairSwitch versus DropTail versus DCTCP.

B. Low-Frequency Incast With Background Traffic

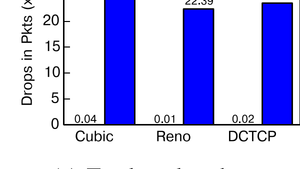
In the following experiments, we run a low-frequency scenario where short-lived flows compete with long-lived flows. Our goal is to see if FairSwitch can help short-lived flows grab some bandwidth from long-lived flows and to see the effects on the throughput of long-lived flows.



(a) FCT metrics for one epoch of 126 concurrent short-lived flows

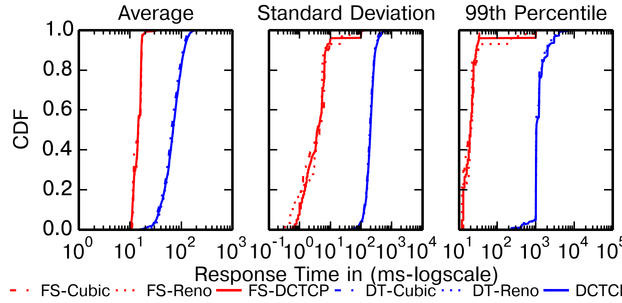


(b) Avg. long flows goodput

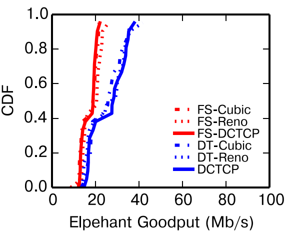


(c) Total packet drops

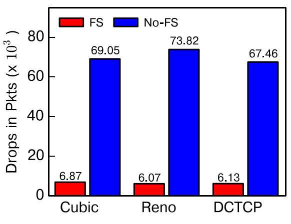
Fig. 8. Incast with background traffic: TCP with FairSwitch versus DropTail versus DCTCP. Each of the 126 short-lived flow requests once a 1.15 MB file (divided into 100 11.5 KB blocks) while competing with 21 long-lived flows.



(a) FCT metrics for 9 epochs of 126 concurrent short-lived flows



(b) Avg. long flow goodput



(c) Total packet drops

Fig. 9. Incast with background traffic: TCP with FairSwitch versus DropTail versus DCTCP. Each of the 126 short-lived flows requests 9 times a 1.15 MB file (divided into 100 11.5 KB blocks) while competing with 21 long-lived flows.

Experimental Setup: With *iperf*, we first generate 21 synchronized long-lived flows from each server in racks 1, 2 and 3 towards rack 4 through the bottleneck link continuously sending traffic for 20 seconds. Then, we again invoke Apache benchmark on the servers of rack 4 to request *"index.html"* from each of the web servers running on the servers in racks 1, 2 and 3. Hence, these web requests must compete for the bottleneck bandwidth with each other and most importantly with the iperf long-lived flows. After long-lived flows have reached a steady state (i.e., at the 10th second), we trigger a single incast epoch consisting of 126 flows issuing 100 consecutive web requests each (i.e., each client requests a 1.15 MB file partitioned into 100 11.5 KB chunks).

Experimental Results: Fig. 8(a) shows that, in medium load, FairSwitch achieves FCT improvements for short-lived flows while nearly not affecting the performance of the long-lived flows. Short-lived flows benefit from FairSwitch by improving the FCT on average and with one order-of-magnitude reduction in FCT variation compared to TCP (Cubic, Reno) with DropTail and DCTCP. Also, in terms of the tail-end (i.e., the 99%), FairSwitch reduces the tail FCT by two order-of-magnitude almost close to the average, and the FCT values are within 10's of ms. The improvement means short-lived flows finish quickly within their stipulated deadlines. Fig 8(b) shows that the long-lived flows are almost not affected by FairSwitch intervention and the throttling of their rates during the incast epochs. Fig. 8(c) shows that the packet drops under FairSwitch are reduced by up to $\approx 99\%$ due to its efficient rate control during incast which explains why short-lived flows can avoid long waiting for timeouts.

C. High-Frequency Incast With Background Traffic

We repeat the previous experiment, increasing the frequency of incast epochs to nine times within the 20-second period (i.e., at the 2nd, 4th,..., and 18th sec). In each epoch, each server requests the web page 100 times. The total transfer is $\approx$ 145 MBytes per epoch and $\approx$ 1.3 GBytes for all 9 epochs.

Experimental Results: Fig. 9(a) shows that even with the higher incast frequency, FairSwitch can keep up even when short-lived flows are competing against each other and long-lived flows. The average and variation of FCT for short-lived flow show similar improvement as the previous experiment. This can be attributed to the lower packet drop rate improvement of up to $\approx 92\%$ with the help of FairSwitch and hence lower chances of experiencing timeouts as shown in Fig. 9(c). Compared to the previous experiment, Fig. 9(b) shows that long-lived flows' throughput is slightly reduced because of the equal rate allocation for a surge of incast flows and the few long-lived flows during the incast epochs. However, we believe that the fair utilization of the bandwidth by short-lived and long-lived flows during incast events is necessary for short-lived flows to finish quickly which explains the marginally lower achieved goodput by long-lived flows. To summarize, the results highlight *FairQ*'s benefits: 1) it reduces the FCT variance and tail-end FCT for short-lived flows by up to two orders of magnitude; 2) it can maintain the same improvement even if the long-lived flows are hogging the network; 3) it efficiently handles incast of various frequencies by fairly allocating the resources even in the presence of bandwidth-hungry long-lived flows; and 4) it fulfils its requirements with no more than the default assumptions about the network stack and without imposing any modifications to VMs protocol stack.

VII. Related Work

Due to the impact and severity of the congestion symptoms, much recent work has been devoted to addressing such shortcomings of TCP in DCNs. Some approaches explored flow path scheduling schemes to isolate short-lived and long-lived flows [2], [38]. However, they require path delay estimation and solving optimization problems which is inadequate for low-latency datacenters. Instead, most of the proposed solutions fall into two categories: window-based (e.g., [3], [4], [11], [39], [40], [41], [42]) or fast loss recovery (e.g., [28], [29], [43], [44], [45]), receiver-driven (e.g., [11], [46], [47], [48],or proactive and token-based (e.g., [5], [13], [14], [15], [49], [50]) schemes. In the window-based category, For instance, DCTCP [3] is a window-based method which proposes a modification to both TCP and RED AQM to adapt the congestion window for stabilizing the queue length at a predefined threshold, guaranteeing thus short delays for incast traffic, without degrading the link utilization.

Fast loss recovery schemes try to improve the agility of TCP in recovering from congestion events by shortening the reaction time. For instance, [28], [43] reduces TCP's minimum retransmission timeout RTO_{min} to reduce the unnecessarily long waiting times after packet losses [44] cleverly tries to deploy a fast congestion-detection mechanism by truncating the packet payload of congestion-causing packets, only conveying the header to the receiver.

Receiver-driven approaches [11], [46], [47], [48] estimate the available bandwidth and convey it to the sender proactively to avoid congestion at the receiver. They mainly handle congestion at the receiver and do not address buffer buildup in the switches. Other work consider the co-flow abstraction to collectively optimize the performance of flows that share the same task [51], [52].

Recently, there has been a line of work proposing proactive congestion controls with in-network assistance [14], [15] or receiver-driven [49], [50]. HPCC [15] is designed for RDMA networks requiring support at the end-host and switch. Proactive transports [14], [49], [50] that use a form of "credit" for packet transmission protocol are promising approaches towards transport protocols that emerged recently but are still under investigation and have not been seen practical deployments.

Alternatively, we have explored an end-to-end flow-aware approach leveraging explicit feedback from in-network devices similar to traditional flow-based systems like ATM-ABR [16], XCP [17] or RCP [18]. More recently, SAB [4], TFC [13], and PowerTCP [5] were proposed and similar to *FairQ* in trying to maintain the occupancy of the buffers low while utilizing the network bandwidth. RCP and TFC measure an estimate of the number of concurrent flows using packet dynamics. PowerTCP in essence is similar as it relies on the exchange of the network telemetry information between the switch and end-hosts to convey the transmitted volume and queue occupancy in the TCP header of in-flight packets. Both SAB and *FairQ* rely on-switch SYN/FIN counting for ongoing flow measurement. This potentially achieves a more accurate estimation of the active flows compared to RCP, TFC and PowerTCP. Also, *FairQ* gives operators the flexibility of controlling the buffer allocation via hyper-parameters which is beneficial in all scenarios compared to SAB which uses a preset buffer capacity and has a problem with the flow counting logic via using SYN packets.

VIII. Conclusion

In this work, we explore a non-intrusive way of reconciling the performance of long-lived with that of short-lived flows in datacenters. To achieve this, the persistent switch queue sizes should operate at low levels to make room for the bursts of incast traffic which helps avoid packet-losses. We proposed *FairQ*, a switch-assisted flow-aware rate matching algorithm that only relies on the existing flow-control mechanism of TCP to feedback queue occupancy levels to TCP senders. To show the practicality of *FairQ*, we implemented several prototypes as Linux-Kernel module for bare-metal switches, as an OpenvSwitch patch for virtual networks which were omitted in the paper due to space constraints and finally as hardware NetFPGA-based switches for programmable hardware switches in datacenters. A number of detailed simulations and real test-bed experiments showed that *FairQ* can achieve its goals efficiently while outperforming the most prominent alternative approaches. Last but not least, knowing that in public datacenters the TCP sender and/or receiver are outside the control of the datacenter operator, *FairQ* ensures that it does not modify the TCP congestion control which enables its true deployment potential in public datacenter networks.

References

[1] L. Guo and I. Matta, "The war between mice and elephants," in *Proc. Int. Conf. Netw. Protoc.*, 2001, pp. 180–188.

[2] W. Wang, Y. Sun, K. Salamatian, and Z. Li, "Adaptive path isolation for elephant and mice flows by exploiting path diversity in datacenters," *IEEE Trans. Netw. Service Manag.*, vol. 13, no. 1, pp. 5–18, Mar. 2016.

[3] M. Alizadeh et al., "Data center TCP (DCTCP)," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 40, pp. 63–74, 2010.

[4] J. Zhang, F. Ren, X. Yue, R. Shu, and C. Lin, "Sharing bandwidth by allocating switch buffer in data center networks," *IEEE J. Sel. Areas Commun.*, vol. 32, no. 1, pp. 39–51, Jan. 2014.

[5] V. Addanki, O. Michel, and S. Schmid, "PowerTCP: Pushing the performance limits of datacenter networks," in *Proc. 19th USENIX Symp. Netw. Syst. Des. Implementation*, 2022, pp. 51–70.

[6] S. Fahmy and T. P. Karwa, "TCP congestion control: Overview and survey of ongoing research," Purdue Univ., West Lafayette, IN, Tech. Rep. 01-016, 2001. [Online]. Available: http://docs.lib.purdue.edu/cstech/1513/

[7] J. Widmer, R. Denda, and M. Mauve, "A survey on TCP-friendly congestion control," *IEEE Netw.*, vol. 15, no. 3, pp. 28–37, May/Jun. 2001.

[8] D. Liu and W. Baptiste, "On approaches to congestion control over wireless networks," *Int. J. Commun. Netw. Syst. Sci.*, vol. 2, no. 3, pp. 222–228, 2009.

[9] R. P. Tahiliani, M. P. Tahiliani, and K. C. Sekaran, "TCP variants for data center networks: A comparative study," in *Proc. Int. Symp. Cloud Serv. Comput.*, 2012, pp. 57–62.

[10] Y. Chen, R. Griffith, J. Liu, R. H. Katz, and A. D. Joseph, "Understanding TCP incast throughput collapse in datacenter networks," in *Proc. 1st ACM Workshop Res. Enterprise Netw.*, 2009, pp. 73–82.

[11] H. Wu, Z. Feng, C. Guo, and Y. Zhang, "ICTCP: Incast congestion control for TCP in data-center networks," *IEEE/ACM Trans. Netw.*, vol. 21, no. 2, pp. 345–358, Apr. 2013.

[12] M. Alizadeh, A. Kabbani, T. Edsall, and B. Prabhakar, "Less is more: Trading a little bandwidth for ultra-low latency in the data center," in *Proc. 9th USENIX Symp. Netw. Syst. Des. Implementation*, 2012, Art. no. 19.

[13] J. Zhang, F. Ren, R. Shu, and P. Cheng, "TFC: Token flow control in data center networks," in *Proc. 11th Eur. Conf. Comput. Syst.*, 2016, Art. no. 23.

[14] S. Hu, W. Bai, B. Qiao, K. Chen, and K. Tan, "Augmenting proactive congestion control with aeolus," in *Proc. 2nd Asia-Pacific Workshop Netw.*, 2018, pp. 22–28.
[15] Y. Li et al., "HPCC: High precision congestion control," in *Proc. ACM Special Int. Group Data Commun.*, 2019, pp. 44–58.
[16] CISCO inc, "Understanding the available bit rate (ABR) service category for ATM VCs," 2005. [Online]. Available: http://www.cisco.com/c/en/us/support/docs/asynchronous-transfer-mode-atm/atm-traffic-management/10415-atmabr.html
[17] D. Katabi, M. Handley, C. Rohrs, D. Katabi, M. Handley, and C. Rohrs, "Congestion control for high bandwidth-delay product networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 32, no. 4, pp. 89–102, 2002.
[18] N. Dukkipati and N. McKeown, "Why flow-completion time is the right metric for congestion control," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 36, pp. 59–62, 2006.
[19] J. Jaffe, "Flow control power is nondecentralizable," *IEEE Trans. Commun.*, vol. 29, no. 9, pp. 1301–1306, Sep. 1981.
[20] A. M. Abdelmoniem and B. Bensaou, "Reconciling mice and elephants in data center networks," in *Proc. IEEE Int. Conf. Cloud Netw.*, 2015, pp. 119–124.
[21] S. Molnár, B. Sonkoly, and T. A. Trinh, "A comprehensive TCP fairness analysis in high speed networks," *Comput. Commun.*, vol. 32, pp. 1460–1484, 2009.
[22] G. Marfia, C. Palazzi, G. Pau, M. Gerla, M. Y. Sanadidi, and M. Roccetti, "TCP libra: Exploring RTT-fairness for TCP," in *Proc. Int. Conf. Res. Netw.*, 2007, pp. 1005–1013.
[23] J. Postel, "RFC 793 - Transmission control protocol," pp. 1–85, 1981. [Online]. Available: http://www.ietf.org/rfc/rfc793.txt
[24] D. Borman, R. Braden, V. Jackbson, and R. Scheffenegger, "TCP extensions for high performance," 2014. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc7323
[25] S. Floyd and V. Jacobson, "Random early detection gateways for congestion avoidance," *IEEE/ACM Trans. Netw.*, vol. 1, no. 4, pp. 397–413, Aug. 1993.
[26] I. Kunze, M. Gunz, D. Saam, K. Wehrle, and J. Rüth, "Tofino p4: A strong compound for AQM on high-speed networks?," in *Proc. IFIP/IEEE Int. Symp. Integr. Netw. Manage.*, 2021, pp. 72–80.
[27] NetFilter.org, "NetFilter packet filtering framework for linux," Jan. 2023. [Online]. Available: http://www.netfilter.org/
[28] A. M. Abdelmoniem and B. Bensaou, "Curbing timeouts for TCP-Incast in data centers via a cross-layer faster recovery mechanism," in *Proc. IEEE Conf. Comput. Commun.*, 2018, pp. 675–683.
[29] A. M. Abdelmoniem and B. Bensaou, "T-RACKs: A faster recovery mechanism for TCP in data center networks," *IEEE/ACM Trans. Netw.*, vol. 29, no. 3, pp. 1074–1087, Jun. 2021.
[30] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose, "Modeling TCP throughput: A simple model and its empirical validation," *SIGCOMM Comput. Commun. Rev.*, vol. 28, no. 4, pp. 303–314, Oct. 1998.
[31] V. Misra, W.-B. Gong, D. Towsley, V. Misra, W.-B. Gong, and D. Towsley, "Fluid-based analysis of a network of AQM routers supporting TCP flows with an application to RED," in *Proc. Conf. Appl. Technol. Architectures Protoc. Comput. Commun.*, 2000, pp. 151–160.
[32] netfpga.org, "The NetFPGA 1G specification documents," Jan. 2023. [Online]. Available: https://netfpga.org/NetFPGA-1G.html
[33] A. Rijsinghani, "RFC 1624—Computation of the internet checksum via incremental update," 1994. [Online]. Available: https://tools.ietf.org/html/rfc1624
[34] K. D. Community, "Data CenterTCP (DCTCP)," Jan. 2023. [Online]. Available: https://docs.kernel.org/networking/dctcp.html
[35] iperf, "The TCP/UDP bandwidth measurement tool," Jan. 2023. [Online]. Available: https://iperf.fr/
[36] Apache.org, "Apache HTTP server benchmarking tool," Jan. 2023. [Online]. Available: http://httpd.apache.org/docs/2.2/programs/ab.html
[37] OpenvSwitch.org, "Open virtual switch project," Jan. 2023. [Online]. Available: http://openvswitch.org/
[38] W. Cheng, F. Ren, W. Jiang, K. Qian, T. Zhang, and R. Shu, "Isolating mice and elephant in data centers," 2016, *arXiv:1605.07732*.
[39] A. M. Abdelmoniem, B. Bensaou, and V. Barsoum, "IncastGuard: An efficient TCP-incast mitigation mechanism for cloud networks," in *Proc. IEEE Int. Conf. Glob. Commun.*, 2018, pp. 1–6.
[40] A. M. Abdelmoniem and B. Bensaou, "Incast-aware switch-assisted TCP congestion control for data centers," in *Proc. IEEE Glob. Commun. Conf.*, 2015, pp. 1–6.
[41] A. M. Abdelmoniem and B. Bensaou, "Hysteresis-based active queue management for TCP traffic in data centers," in *Proc. IEEE Int. Conf. Comput. Commun.*, 2019, pp. 1621–1629.
[42] A. M. Abdelmoniem and B. Bensaou, "Enhancing TCP via hysteresis switching: Theoretical analysis and empirical evaluation," *IEEE/ACM Trans. Netw.*, vol. 31, no. 6, pp. 2614–2623, Dec. 2023.
[43] V. Vasudevan et al., "Safe and effective fine-grained TCP retransmissions for datacenter communication," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 39, pp. 303–314, 2009.
[44] P. Cheng, F. Ren, R. Shu, and C. Lin, "Catch the whole lot in an action: Rapid precise packet loss notification in data center," in *Proc. 11th USENIX Symp. Netw. Syst. Des. Implementation*, 2014, pp. 17–28.
[45] A. M. Abdelmoniem, H. Susanto, and B. Bensaou, "Taming latency in data centers via active congestion-probing," in *Proc. IEEE Int. Conf. Distrib. Comput. Syst.*, 2019, pp. 101–110.
[46] J. Hwang, J. Yoo, and N. Choi, "Deadline and incast aware TCP for cloud data center networks," *Comput. Netw.*, vol. 68, pp. 20–34, 2014.
[47] K. Nagaraj, D. Bharadia, H. Mao, S. Chinchali, M. Alizadeh, and S. Katti, "NUMFabric: Fast and flexible bandwidth allocation in datacenters," in *Proc. ACM SIGCOMM Conf.*, 2016, pp. 188–201.
[48] J. Xue, M. U. Chaudhry, B. Vamanan, T. N. Vijaykumar, and M. Thottethodi, "Dart: Divide and specialize for fast response to congestion in RDMA-based datacenter networks," *IEEE/ACM Trans. Netw.*, vol. 28, no. 1, pp. 322–335, Feb. 2020.
[49] B. Montazeri, Y. Li, M. Alizadeh, and J. Ousterhout, "Homa: A receiver-driven low-latency transport protocol using network priorities," in *Proc. ACM Special Int. Group Data Commun.*, 2018, pp. 221–235.
[50] Q. Cai, M. T. Arashloo, and R. Agarwal, "DcPIM: Near-optimal proactive datacenter transport," in *Proc. ACM SIGCOMM Conf.*, 2022, pp. 53–65.
[51] L. Shi, Y. Liu, J. Zhang, and T. Robertazzi, "Coflow scheduling in data centers: Routing and bandwidth allocation," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 11, pp. 2661–2675, Nov. 2021.
[52] H. Susanto, B. L. Ahmed, M. Abdelmoniem, H. Zhang, and D. Towsley, "A near optimal multi-faced job scheduler for datacenter workloads," in *Proc. IEEE 39th Int. Conf. Distrib. Comput. Syst.*, 2019, pp. 2026–2036.

Ahmed M. Abdelmoniem (Member, IEEE) received the PhD degree in computer science and engineering from the Hong Kong University of Science and Technology, Hong Kong, in 2017. He is an Assistant Professor with the Queen Mary University of London, U.K. and Assuit University, Egypt. Formerly, he was a research scientist with KAUST, Saudi Arabia, and a senior researcher with Huawei's Future Networks Lab, Hong Kong. He is a principal and co-investigator on projects totalling USD 1.5mil in funding. His research interests lie in the intersection of distributed systems, networks and machine learning. His work appears in top-tier conferences and journals including NeurIPS, AAAI, MLSys, ACM EuroSys, IEEE INFOCOM and ICDCS, *IEEE/ACM Transactions on Networking*, *IEEE Internet of Things Journal*, and *Elsevier Computer Networks*. He is a member of ACM and USENIX.

Brahim Bensaou (Senior Member, IEEE) received the PhD degree in computer science from the University Paris VI, in 1993. He is a faculty member with the CSE Department of HKUST. Formerly, he held positions of research assistant with France Telecom Research Labs, research associate with HKUST, and senior staff with the National R&D Centre for Wireless Communications in Singapore where he led the strategic research group on wireless networking. His research is in general centered around Internet Communication, Wireless communications and Mobile Networks, and their performance (e.g., Congestion Control, Information-centric Networks, Energy efficiency, and Performance evaluation). He published more than 130 research papers in prominent conferences and journals, received numerous research grants, supervised and graduated more than 20 postgraduate research theses including both PhD and master's and holds 3 granted US patents, one of which is licensed. He is a member of ACM.