Augmented Queue: A Scalable In-Network Abstraction for Data Center Network Sharing

Xinyu Crystal Wu[*]
Rice University

Zhuang Wang[*]
Rice University

Weitao Wang
Rice University

T. S. Eugene Ng
Rice University

ABSTRACT

Traffic aggregates in cloud data center networks are by and large buffered and transmitted by simple physical FIFO queues. Despite the crucial role they play, a well-known problem of physical FIFO queues is that they are unable to provide precise bandwidth guarantees. This leads to a range of negative impacts spanning the application layer, the transport layer, and the data link layer.

In this paper, we address this problem with Augmented Queue (AQ), a scalable in-network abstraction that provides precise bandwidth guarantees for traffic constituents. AQ serves multiple valuable use cases in data center networks. For example, AQ facilitates the isolation of traffic from different applications; ensures that different congestion control algorithms can properly co-exist; and enforces inbound and outbound bandwidth for virtual machines. We demonstrate via testbed and simulation experiments that AQ can provide precise bandwidth guarantees and scale to millions of traffic constituents.

CCS CONCEPTS

• **Networks** → **In-network processing**; **Network dynamics**;

KEYWORDS

Cloud Computing; Network Sharing; In-network Abstraction

ACM Reference Format:
Xinyu Crystal Wu, Zhuang Wang, Weitao Wang, and T. S. Eugene Ng. 2023. Augmented Queue: A Scalable In-Network Abstraction for Data Center Network Sharing. In *ACM SIGCOMM 2023 Conference (ACM SIGCOMM '23), September 10–14, 2023, New York, NY, USA*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3603269.3604858

1 INTRODUCTION

Large cloud infrastructures such as Azure, AWS, and Google Cloud handle the network traffic of millions of customers every day. Despite the extreme number of customers and applications sharing their networks' bandwidth, at the lowest level, aggregated network traffic traversing each switch port is buffered and transmitted by a small number of – and sometimes even just one – physical FIFO queue (or "physical queue" for short). To govern how physical queues are shared, cloud providers rely on end-to-end congestion control (CC) algorithms as well as traffic rate limiters at end hosts.

However, a well-known problem of physical queues is that they are unable to provide precise bandwidth guarantees for different traffic constituents, which negatively impacts multiple network layers. At the application layer, traffic from aggressive and gentle applications alike sharing a physical queue can interfere with each other, leading to unpredictable performance that can vary by an order of magnitude [8, 37, 49, 62]. At the transport layer, the physical queue cannot ensure the proper co-existence of different CC algorithms that greatly differ by their optimization objectives (*e.g.*, low delay vs. full utilization), their feedback signals (*e.g.*, packet loss signal vs. ECN signal vs. delay signal), and their ramp-up ramp-down aggressiveness [58]. At the link layer, a virtual machine (VM) may want precise specifications for both its inbound and outbound bandwidth (*i.e.*, no more, no less) [1, 14, 16, 33]. However, not only is the physical queue unable to guarantee a bandwidth minimum, but it can also release traffic that exceeds the specified bandwidth when the specified bandwidth is less than the link capacity.

Many cloud providers have confirmed that these problems are common in production environments [27, 32, 55]. Nevertheless, shared physical queues are still extensively employed because no practical and superior alternative exists. Specifically, per-flow queue [26] seems like a plausible solution that works by isolating traffic in dedicated queues, but in reality, it does not scale well because the number of tenants in a data center is several orders of magnitude greater than the number of per-flow queues available in today's switches [3, 64]. Furthermore, per-flow queues can also release traffic that exceeds the specified VM bandwidth at the link layer. CC algorithms and rate limiters at end hosts help lessen the problems with physical queues but they are far from sufficient. First, although a well-designed CC algorithm allows fair sharing of the network, it cannot support flexible and precise bandwidth guarantees [63]. Second, although rate limiters at end hosts can prevent overly aggressive injection of traffic, they cannot address the aforementioned problems in the transport layer when different CC algorithms share the physical queue. Furthermore, depending on how the limiters are configured, the inbound bandwidth specification of a VM can be under-utilized or violated depending on the traffic pattern among VMs [59].

In this paper, we propose Augmented Queue (AQ), an in-network abstraction to provide precise bandwidth guarantees for different

[*]Both authors contributed equally.

ACM SIGCOMM '23, September 10–14, 2023, New York, NY, USA
© 2023 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 979-8-4007-0236-5/23/09…$15.00
https://doi.org/10.1145/3603269.3604858

traffic constituents. AQ can isolate traffic from different applications and mitigate performance interference. It also provides different types of network feedback, such as packet loss, ECN, and delay, for different CC algorithms simultaneously. In addition, it can control traffic rates independent of the physical queue length.

The core idea of AQ to provide traffic constituents with precise bandwidth guarantees is controlling their rate based solely on their own traffic. There are two challenges in the design of AQ. The first challenge is determining what measure can be suitably used for traffic rate control. We observe that the physical queue length is not a suitable option because it leads to inflexibility in reflecting the relation between the traffic rate and the allocated rate. Instead, AQ controls the rate with a deliberately-designed measure function that can provably allow a traffic constituent to achieve the precise bandwidth guarantee in a shared network. The second challenge lies in devising a scalable and efficient approach to control the traffic rate for different traffic constituents separately. AQ converts the measure function from a continuous domain to a discrete domain, allowing the control of traffic rate at the packet level. It then differentiates and guarantees bandwidth for traffic from different constituents based on the measure function, generates different network CC feedback, and limits the rates of different traffic constituents simultaneously. AQ can support millions of traffic constituents for precise bandwidth guarantees, regardless of the number of physical queues in switches.

In summary, we make the following contributions:

- We analyze the limitations of physical queues and discuss the feasibility to relieve the data center's reliance on physical queues for data center network sharing.
- We propose Augmented Queue (AQ), a scalable in-network abstraction to provide precise bandwidth guarantees for millions of traffic constituents simultaneously.
- We prototype AQ in both NS3 simulation and a Tofino testbed. The evaluations show that AQ can achieve precise bandwidth guarantees for the application layer, transport layer, and link layer. For example, when two applications expect to fairly share a network, AQ can bound the difference of their shared bandwidth between 0.99 and 1.01; but the difference with physical queues and rate-limiting solutions can be arbitrary.

2 MOTIVATION

In today's data center networks, a switch is typically equipped with *physical queues* in which packets are buffered and transmitted. These physical queues can absorb traffic burstiness to reduce packet drops. In addition, many congestion control (CC) algorithms rely on the congestion information provided by physical queues, such as the physical queue length and queuing delay, to optimize for low latency and high network utilization. Physical queues possess two essential properties: 1) they are shared by all the traffic passing through them, regardless of their applications and the applied CC algorithms; and 2) they require built-up queues to generate congestion signals, *i.e.*, the incoming traffic rate should surpass the line rate. These two properties, we contend, impose fundamental limitations on physical queues to provide precise bandwidth guarantees.

2.1 Traffic from Different Applications Can Interfere in Physical Queues

Today's data centers can isolate the compute, memory, and disk resources for different applications [19, 27]. However, the network is still shared by different applications and their traffic can interfere with each other. They can experience unpredictable network performance and their throughput can vary by an order of magnitude [8, 36, 37, 49, 53, 60, 62]. In order to achieve predictable network performance, we argue that data centers should provide isolation for multiple applications to better share the network.

Goal 1: Provide network isolation for different applications. Each application has a specific bandwidth requirement and we can regard an application as an entity. The sharing of the network for an application on a bottleneck link is only determined by its own allocated bandwidth, regardless of the traffic protocols in use (*i.e.*, TCP or UDP) and the number of flows [45, 46, 53].

Example 1: Suppose multiple distributed applications are sharing a network link. Each involves multiple VMs and can generate an arbitrary number of flows. They require an equal amount of bandwidth on the network bottleneck link. Unfortunately, a physical queue alone cannot satisfy this requirement. For example, applications with UDP traffic have the potential to monopolize the bandwidth and starve applications with TCP traffic. Furthermore, when all applications generate TCP traffic, the bandwidth sharing among these applications is determined by the number of flows generated by each of them due to the fairness property of the CC algorithms [4, 22, 34]. Because the number of flows an application can generate is arbitrary, the bandwidth sharing is also arbitrary. Although a switch may have several physical queues, the number is limited and it is unlikely that the switch can allocate a dedicated physical queue to each application. Therefore, some applications have to share a physical queue, resulting in arbitrary bandwidth sharing among them.

2.2 Different CC Algorithms Are Hard to Coexist in Physical Queues

Many CC algorithms [4, 22, 34, 39, 68] have emerged in recent years, targeting different optimization objectives (*e.g.*, high bandwidth, network stability, and ultra-low latency), using different types of congestion signals (*e.g.*, loss, ECN, and delay), and having different ramp-up ramp-down aggressiveness. Different tenants or even different applications of a tenant may prefer different CC algorithms to meet their performance requirements. For example, latency-sensitive applications, such as high frequency trading [20], demand extremely low latency and thus may prefer CC algorithms like Swift [34] and HPCC [39]. On the other hand, throughput-intensive applications, such as distributed training [30, 38, 50], prioritize high throughput and hence prefer CC algorithms like DCTCP [4] and DCQCN [68]. In addition, when developing and upgrading CC algorithms in data centers, multiple CC algorithms can be expected to co-exist under the same cluster to avoid physical partitioning and application migration. Therefore, we argue that data centers should support multiple CC algorithms sharing the network simultaneously.

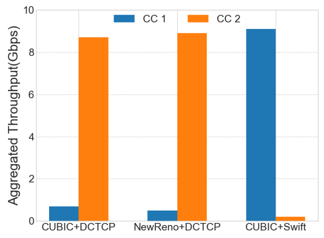


Figure 1: Traffic interference between different CC algorithms when they share a physical queue.

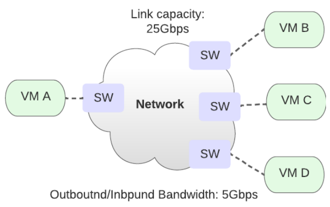


Figure 2: An example of bi-directional bandwidth guarantees for VMs.

Goal 2: Provide network isolation for different CC algorithms. A tenant can use different CC algorithms for its different applications based on different optimization objectives and performance requirements. We can regard a CC algorithm in a tenant as an entity that can involve one or multiple applications. Each entity can specify its expected bandwidth for sharing the bottleneck link in data center networks.

Example 2: Suppose that multiple entities are sharing a network with different CC algorithms and they require to fairly share the network. A physical queue alone cannot evenly distribute network bandwidth among entities. Because traffic from different CC algorithms shares the same physical queue, they have to suffer the same degree of congestion. However, different algorithms react differently under the same congestion, leading to different behaviors and bandwidth sharing.

To illustrate the interference of different CC algorithms, we measure the throughput of different CC algorithms using a shared dumbbell topology with 10Gbps link capacity. We evaluate three types of CC, *i.e.*, drop-based: CUBIC [22] and NewReno [17]; ECN-based: DCTCP [4]; and delay-based: Swift [34]. We use two CC algorithms at a time, with each supporting 10 traffic flows. As shown in Figure 1, different CC algorithms cannot coexist gracefully and fairly share the network fabric. For example, the DCTCP traffic aggressively captures a significantly larger bandwidth compared to those drop-based CC algorithms. Specifically, when sharing the network bottleneck link with CUBIC, the throughput of DCTCP is 8.7Gbps while the throughput of CUBIC is only 0.7Gbps. Similarly, Swift traffic struggles when sharing the network fabric with other CC algorithms, as its throughput falls below 0.2Gbps, leaving it nearly starved.

Even with multiple physical queues, it is still difficult to fairly share the network bandwidth among different entities. Due to the limited number of physical queues in a switch, we cannot allocate a dedicated queue to each entity or each tenant. Traffic from some entities or tenants has to share the same queue. One way to share the queues is to allocate a physical queue to each CC algorithm, and traffic with the same CC from different tenants needs to share a physical queue. It is equivalent to the case that multiple distributed applications using the same CC are sharing a physical queue, as discussed in Section 2.1, and it is unable to satisfy the requirement in Example 1.

2.3 Physical Queues Cannot Achieve Inbound and Outbound Bandwidth Guarantees for VMs

A tenant in a data center typically consists of multiple virtual machines (VMs) that communicate with each other. Because the network is shared in a best-effort manner, it is common for tenants to require reserved network bandwidth for their VMs to avoid traffic interference and to achieve predictable network performance [1, 8]. Analogous to physical machines with inbound and outbound bandwidth regardless of the traffic patterns, we argue that data centers should reserve exact network bandwidth for VMs [14, 16, 33].

Goal 3: Provide bi-directional bandwidth guarantees for different VMs. A tenant can specify the expected network bandwidth reservations for each of its VMs. We can take a VM as an entity. Each VM has an illusion that it occupies a network cloud exclusively with the reserved inbound and outbound bandwidth for any traffic patterns.

Example 3: Suppose multiple VMs are sharing a network and they can communicate with each other. Each VM can have a *traffic profile* for its bi-directional bandwidth, *i.e.*, the network allocates an exact bandwidth to both directions. Each VM expects to send traffic to and receive traffic from all the other VMs simultaneously with the guaranteed bandwidth.

Physical queues alone cannot satisfy this traffic profile. They can generate congestion signals, such as ECN and packet loss, to inform end hosts to adjust their sending rates when they have built-up queues. However, they cannot generate congestion signals when the ingress traffic line is always lower than the line rate. For example, in Figure 2, four VMs are connected through a network with 25Gbps link capacity. Each VM has a traffic profile with a 5Gbps outbound bandwidth and a 5Gbps inbound bandwidth. Their traffic patterns among each other are arbitrary. Because the profiled bandwidth of each VM is lower than the link capacity, there are no queues built-up in physical queues and they cannot generate congestion signals for VMs to control their rates.

It is possible for rate limiters at end hosts to achieve outbound bandwidth limits for VMs, but they cannot satisfy bi-directional guarantees. For example, it is common to limit the outbound bandwidth of VMs, *i.e.*, their sending rates, but it can violate the inbound bandwidth required by the traffic profile. After deploying a rate limiter of 5Gbps at each VM, a VM can receive at most 15Gbps when three VMs send traffic to the same one simultaneously, much higher than the specified 5Gbps inbound bandwidth.

In order to achieve the required inbound bandwidth guarantee, one needs to assume the traffic pattern among VMs to allocate bandwidth to each VM pair [37, 53]. However, because the traffic pattern can be arbitrary, any allocation strategy can lead to underutilized bandwidth. Take VM A as an example. To guarantee its 5Gbps inbound bandwidth, one strategy is to equally split this bandwidth among the other three VMs, *i.e.*, each of them can send traffic to VM A at the rate of 5/3Gbps. It is likely that VM B and VM C have no traffic to send to VM A during a period and only VM D communicates with VM A. Due to the mismatch between the traffic pattern and the specified strategy, only one-third of the inbound bandwidth of VM A is utilized.

2.4 Relieve the Reliance on Physical Queues

Based on the aforementioned discussion, we can conclude that physical queues are incapable of supporting two fundamental and critical functionalities: 1) to provide different congestion signals for different CC algorithms; and 2) to limit the rates of distributed applications, CC algorithms, and VMs. We observe that *achieving these two functionalities does not necessarily depend on the physical queues, but the discrepancy between the allocated rate and the traffic rate* (refer to Section 3.2). However, the key challenge is how to capture the discrepancy for different applications, CC algorithms, and VMs, respectively. The recent trend of programmable switches [6, 10, 23, 41] in data center networks provides a new opportunity. Since programmable switches can support customized packet processing and provide stateful memory [11], it is feasible to differentiate traffic, monitor and limit their rates separately, and generate different congestion signals, regardless of the number of physical queues. Hence, we propose an in-network augmented queue (AQ) abstraction to capture the discrepancy and provide these two functionalities for tenants.

3 AUGMENTED QUEUE ABSTRACTION

In this section, we first introduce the design requirements for traffic rate control of different entities. We then analyze the feasibility to relieve the reliance on physical queues and explore alternative options to achieve rate limiting and generate different network feedback. Finally, we provide a detailed design for AQ with a mathematical model and a framework that allows for better traffic rate control of different entities.

3.1 Design Requirements

Before presenting the AQ abstraction, we first discuss three requirements that are necessary for effective traffic rate control among different entities.

R1: Provide rate limiting for entities. Different entities sharing the network can specify their required bandwidth. For example, a distributed application or a CC algorithm can expect an aggregated bandwidth of all its traffic on a network bottleneck; a VM can also expect that both its inbound and outbound rates conform to a specific traffic profile. Each entity should be provided with the illusion that its traffic could exclusively use a network with the allocated bandwidth.

R2: Provide different network feedback for different entities. The traffic of multiple entities can share a single physical queue in the network. In case of network congestion, these entities can make different contributions to physical queuing. The network should differentiate their different contributions and provide different network feedback to regulate their traffic accordingly. Because different entities can choose their preferred CC algorithms, the network should simultaneously provide different types of network congestion signals, such as packet loss, ECN marking, and queuing delay, based on different configurations, such as the maximum limit to drop packets, ECN thresholds, and target queuing delay. In addition, the network feedback of an entity should only depend on its own traffic and the generation of the feedback should be triggered independently at different times for different entities.

R3: Scale to a large number of entities. Entities of different granularities can be formed based on different requirements and network scenarios, such as different applications, different CC algorithms, and VMs sharing a network. Because of the increasing number of applications, CC algorithms, and VMs in today's data centers, there could be a large number of entities, *e.g.*, hundreds of thousands of entities, sharing the same physical link. The number of physical queues in commodity switches cannot keep up with this scale in cloud networks. A practical AQ abstraction must be scalable to accommodate a large number of sharing entities in data center networks.

3.2 Rethink What to Use for Traffic Rate Control

Suppose an entity has traffic flows from different *sources* and these flows share a network bottleneck link. We define the *allocated rate* R of an entity as the bandwidth allocated to its traffic in the bottleneck link. The allocated rate can be equal to or less than the link capacity. We also define the *arrival rate* $r(t)$ of an entity as the aggregated throughput of its traffic entering the bottleneck link at time t.

Analogous to the analysis in Generalized Processor Sharing (GPS) [31, 44, 65], we define a source is *backlogged* when it has packets to be transmitted. Therefore, a source has two types of time periods: backlogged periods and empty periods, and it has no packets to send during the empty periods. Similarly, we define an entity is backlogged when it has any backlogged sources and an entity is empty when it has no backlogged sources.

During backlogged periods of an entity, the goal of the traffic control is to ensure that

$$|r(t) - R| < \epsilon, \forall t \text{ in backlogged periods.} \quad (1)$$

where ϵ is an arbitrary small positive number. However, it is very challenging to satisfy Expression (1) for every time point in backlogged periods. $r(t)$ is contributed by all the backlogged sources and the set of backlogged sources is changing over time. Suppose $r(t) = R$ at time t and some sources turn to empty from backlogged after time t. The traffic rates of backlogged sources need to be adjusted; otherwise, the arrival rate will be smaller than R. It takes time for rate adjustment and Expression (1) cannot be satisfied during the adjustment.

Therefore, a practical goal of traffic control during backlogged periods is to ensure that the average of $r(t)$ over a given time interval δ approximates R. In other words, the traffic volume entering the bottleneck link during δ approximates δR, *i.e.*,

$$|\int_t^{t+\delta} r(t)\,dt - \delta R| < \epsilon, \ (t, t+\delta] \text{ in a backlogged period.} \quad (2)$$

We can see that Expression (1) focuses on the rate difference for every time point, but Expression (2) focuses on the byte difference during a time interval.

Because we only care about the arrival rate in backlogged periods, we split the time into intervals $(0, t_1]$, $(t_1, t_2]$, ..., $(t_m, t_{m+1}]$ according to the two types of time periods, which alternate in these intervals. For example, suppose $(t_i, t_{i+1}]$ is a backlogged period, then $(t_{i+1}, t_{i+2}]$ is an empty period.

In the following, we will discuss what can be used to control the traffic rate to satisfy Expression (2) for each entity. There are two mechanisms for traffic rate control: 1) the traffic rate is adjusted by a CC algorithm according to the network feedback; and 2) the traffic rate is limited by a rate limiter. We will discuss these two mechanisms separately.

3.2.1 *Traffic rate control with CC.*

In today's data center, the network feedback for different CC algorithms is heavily coupled with physical queues. For example, packets are dropped when the queue length reaches a limit; ECN is marked on packets when the queue length exceeds a threshold; and queuing delay is determined by the physical queue length.

However, this coupling leads to inflexibility that makes certain traffic rate control outcomes unachievable. In Example 3 in Section 2.3, the allocated rate R of each VM is $1/5$ of the link capacity. VMs are expected to receive feedback from the network to inform them to decrease their traffic rates when $r(t)$ is greater than 5Gbps. However, there is no queue build-up in the physical queue. Another example is that multiple entities can share the network. Suppose the arrival rate of one entity is much lower than its allocated rate, but the network is still saturated by traffic from other entities and the physical queue length exceeds the threshold to generate network feedback. However, this entity should not be one of them to receive feedback to reduce the traffic rate.

Our proposal. In this paper, we decouple the network feedback from the physical queue length and propose to adjust the traffic rate of an entity according to the *discrepancy* between its allocated rate and arrival rate. In other words, the network feedback is determined by the discrepancy, and the traffic control of an entity is only related to its own traffic and independent of the traffic from other entities.

Next, we will discuss how to define the function for the discrepancy between the allocated rate and the arrival rate to generate network feedback and achieve Expression (2).

A strawman function for the discrepancy. Consider taking the integrated difference between $r(t)$ and R in a time interval $[t, t+\delta]$ as the discrepancy, defined as:

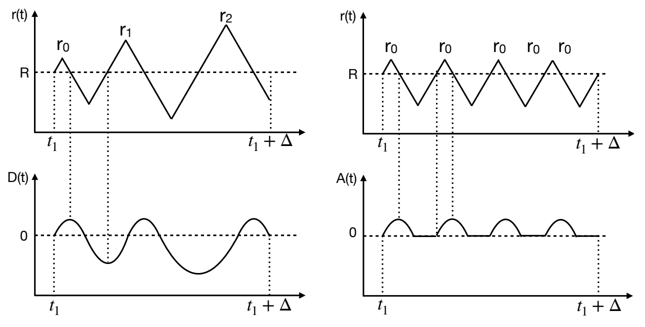
$$d(t, t+\delta) = \int_t^{t+\delta} r(t)\,dt - \delta R. \qquad (3)$$

When the integrated difference is negative, it implies increasing the traffic rate, and vice versa. The integrated difference is a continuous function with respect to time. It can be positive at the end of a backlogged period. Though the integrated difference keeps decreasing during the next empty period, it might be still greater than zero at the beginning of the next backlogged period. Therefore, we define the strawman function as:

$$D(t) = \begin{cases} D(t_i) + d(t_i, t), \text{ if } t \in (t_i, t_{i+1}] \text{ that is backlogged,} & (4) \\ \max\{0, D(t_i) - R(t - t_i)\}, \text{ if } t \in (t_i, t_{i+1}] \text{ that is empty.} & (5) \end{cases}$$

$D(t)$ is initialized as zero. One CC algorithm can use $D(t)$ as the discrepancy to generate the network feedback to adjust $r(t)$. To achieve Expression (2), it is equivalent to achieving the following expression:

$$|D(t+\delta) - D(t)| < \epsilon, \ (t, t+\delta] \text{ in a backlogged period.} \qquad (6)$$



(a) $D(t)$ is used. The average of $r(t)$ approximates R, but $r(t)$ can be much higher than R.

(b) $A(t)$ is used. $r(t)$ is close to R because the surplus is not allowed. $A(t)$ can minimize traffic bursts.

Figure 3: The varying arrival rate of an entity with different functions for the discrepancy. The applied CC overly reduces the traffic rate.

The strawman has limitations. Unfortunately, there are disadvantages when using $D(t)$ as the function for the discrepancy because it allows an entity to use the *surplus*, i.e., the value of $|D(t)|$ when $D(t) < 0$ in backlogged periods. Figure 3(a) illustrates the arrival rate $r(t)$ of an entity and it uses $D(t)$ as the discrepancy to generate network feedback with $D(t_1) = 0$. The applied CC algorithm aims for zero queuing delay; it overly reduces the traffic rate and leads to a negative $D(t)$ in backlogged periods. When $r(t)$ exceeds the allocated rate R, $D(t)$ becomes positive and the CC algorithm aggressively decreases $r(t)$ from r_0. Then $D(t)$ becomes negative and the CC increases $r(t)$ until it reaches r_1 and $D(t) > 0$ again. Because the surplus is used, r_1 is higher than r_0. Similarly, the arrival rate r_2 in the next cycle can be even higher than r_1. Although $D(t_1 + \Delta) = D(t_1) = 0$ and it achieves Expression 2, the arrival rate can be either much higher or lower than R. It can lead to severe queuing delays and even packet loss. For example, when R is equal to the link capacity and $r(t)$ is approaching r_2, the network congestion keeps worsening because the traffic rate is increasingly higher than the link capacity.

The A-Gap function for the discrepancy. We argue that the usage of surplus to generate network feedback should not be allowed. We refine the strawman function to calculate the discrepancy of each entity with the A-Gap as

$$A(t+\epsilon) = \max\{0, A(t) + d(t, t+\epsilon)\}, \qquad (7)$$

where $A(0) = 0$. The A-Gap function is set to zero when $A(t) + d(t, t+\delta) < 0$, regardless of the period is backlogged or empty.

The A-Gap equals the physical queue length when the allocated rate R is the link capacity, but they are different when R is smaller than the link capacity. For example, the allocated rate R for an entity is 5Gbps and the link capacity is 25Gbps. When the arrival rate is 6Gbps, its $A(t) > 0$, but the physical queue length is always zero.

Using $A(t)$ as the function for the discrepancy can address the disadvantages resulted from using $D(t)$ as the function. Figure 3(b) illustrates the arrival rate $r(t)$ of an entity with $A(t)$ as the discrepancy to generate network feedback. The arrival rate begins to decrease when it reaches r_0 and $A(t)$ bottoms out at zero. When the

CC algorithm increases $r(t)$ again, the arrival rate can only achieve r_0, rather than r_1, because the surplus is not allowed.

The guideline to use the A-Gap. Using $A(t)$ as the function can isolate the traffic from different entities. It also penalizes entities that apply ill-designed CC algorithms. Here we provide a condition for how to achieve Expression (2) when using the A-Gap as the function for the discrepancy to generate network feedback.

Lemma 3.1. *To achieve Expression (2), one CC algorithm should ensure*

$$\begin{cases} |A(t+\delta) - A(t)| < \epsilon,\ (t, t+\delta] \text{ in a backlogged period,} & (8) \\ A(t) > 0,\ t \text{ in a backlogged period.} & (9) \end{cases}$$

According to the definition of $A(t)$ in Expression (7), we have $A(t+\delta) = A(t+\delta) + d(t, t+\delta)$ when $A(t)$ is always positive. Therefore, $|A(t+\delta) - A(t)| = |d(t, t+\delta)|$. If Expression (8) holds, Expression (2) also holds. Lemma 3.1 implies that a CC algorithm should keep $A(t)$ positive to make the traffic rate conform to the allocated rate. When R is the link capacity, it implies that a CC algorithm should keep the physical queue length greater than zero to fully utilize the bottleneck link bandwidth if it uses the network feedback to adjust the traffic rate.

3.2.2 Traffic rate control with rate limiters. A rate limiter can also control the traffic rate without any assumptions on the transport protocols (*i.e.*, TCP or UDP) and applied CC algorithms. As aforementioned, when R is less than the link capacity, the physical queue length is unrelated to the arrival rate and it alone cannot limit the traffic rate to R.

We can control the traffic rate with a rate limiter based on the A-Gap function. The A-Gap keeps increasing when the arrival rate exceeds the allocated rate. If there is no constraint on the A-Gap for an entity, $r(t)$ can be arbitrarily greater than R, contradicting the purpose of a rate limiter. We can address this issue by limiting an entity's maximum A-Gap, which is denoted as $limit$, to bound the difference between $r(t)$ and R in backlogged periods. According to Expression (7), we have $A(t+\epsilon) \geq A(t) + d(t, t+\epsilon)$ given a time interval $[t, t+\epsilon]$ in a backlogged period. Because $A(t+\epsilon) \leq limit$ and $A(t) \geq 0$, we have $d(t, t+\epsilon) \leq limit$. According to the definition of $d(t, t+\epsilon)$ in Expression (3), the difference between the average of $r(t)$ and R in the time interval $[t, t+\epsilon]$ is bounded by $limit/\delta$.

Takeaways. We can conclude that realizing the two mechanisms for traffic rate control does not depend on the physical queue length. The rate control of an entity should be related to its own traffic only, and independent of traffic from other entities. In addition, we can use the A-Gap function as the discrepancy for traffic rate control of different entities.

3.3 Augmented Queue Design

An Augmented Queue (AQ) is defined by an allocated rate R and the A-Gap function. It can calculate the A-Gap for each entity and allow entities to control their traffic rate according to the A-Gap. In this section, we will first develop a streaming algorithm to accurately compute the A-Gap. We then devise a framework that uses the A-Gap to provide traffic rate control.

3.3.1 A mathematical model to compute the A-Gap. According to Expression (7), the definition of the A-Gap is in a continuous domain and it is a function of R, $r(t)$ and time t. However, it is very challenging to derive the varying arrival rate $r(t)$ and to calculate the A-Gap for every time point.

In practice, the arrival time of packets is discrete and there are time gaps between the arrivals of any two packets. These time gaps can be viewed as empty periods, allowing us to only care about the A-Gap of the time each packet arrives. Given a sequence of packets and p_k is the k^{th} packet. The packet index starts from 1. $p_k.time$ is the arrival time of p_k and $p_k.size$ is its packet size. We also denote $\Delta(k) = p_k.time - p_{k-1}.time$.

Theorem 3.2. *The A-Gap can be calculated with the following recurrence relation:*

$$\begin{cases} A(0) = 0, & (10) \\ A(p_k.time) = \max\{0, A(p_{k-1}.time) - \Delta(k)R\} + p_k.size. & (11) \end{cases}$$

Proof. According to the definition of the A-Gap in Expression (7), we have $p_{k-1}.time < p_k.time - \delta < p_k.time$ where δ is any small time interval. We first compute $A(p_k.time - \delta)$.

$$\begin{aligned} & A(p_k.time - \delta) \\ &= \max\{0, A(p_{k-1}.time) + d(p_{k-1}.time^+, p_k.time - \delta)\} \\ &= \max\{0, A(p_{k-1}.time) + \int_{p_{k-1}.time^+}^{p_k.time-\delta} (r(t) - R)\, dt\} \\ &= \max\{0, A(p_{k-1}.time) - \Delta(k)R + \delta R\}, \end{aligned}$$

where $p_{k-1}.time^+$ is the time point right after p_{k-1} arrives. Note that $\int_{p_{k-1}.time^+}^{p_k.time-\delta} r(t)\, dt = 0$ because no packet arrives in $(p_{k-1}.time$, $p_k.time - \delta]$. We then compute $A(p_k.time^-)$, where $p_k.time^-$ is the time point right before p_k arrives.

$$\begin{aligned} & A(p_k.time^-) \\ &= \max\{0, A(p_k.time - \delta) + d(p_k.time^+ - \delta, p_k.time^-)\} \\ &= \max\{0, A(p_k.time - \delta) - \int_{p_k.time^+-\delta}^{p_k.time^-} (r(t) - R)\, dt\} \\ &= \max\{0, A(p_k.time - \delta) - \delta R\}. \end{aligned}$$

Note that $\int_{p_k.time^+-\delta}^{p_k.time^-} r(t)\, dt = 0$ because no packets arrive during $(p_k.time - \delta, p_k.time)$. When $A(p_k.time - \delta) > 0$, we have

$$\begin{aligned} A(p_k.time^-) &= \max\{0, A(p_k.time - \delta) - \delta R\} \\ &= \max\{0, A(p_{k-1}.time) - \Delta(k)R + \delta R - \delta R\} \\ &= \max\{0, A(p_{k-1}.time) - \Delta(k)R\}. \end{aligned} \quad (12)$$

When $A(p_k.time - \delta) = 0$, we have

$$\begin{aligned} A(p_k.time^-) &= \max\{0, A(p_k.time - \delta) - \delta R\} \\ &= \max\{0, -\delta R\} = 0. \end{aligned} \quad (13)$$

Combining Expression (12) and Expression (13), we can get

$$A(p_k.time^-) = \max\{0, A(p_{k-1}.time) - \Delta(k)R\}.$$

Packet p_k arrives at time $p_k.time$ so that we have

$$A(p_k.time) = \max\{0, A(p_{k-1}.time) - \Delta(k)R\} + p_k.size.$$

Since $A(0) = 0$, we have the recurrence relation in Theorem 3.2. □

Algorithm 1: The streaming algorithm

```
Input: $aq$ is an augmented queue. $aq.rate$ is its allocated
       rate and $aq.gap$ is its A-Gap.
Function A_Gap($aq$, $pkt$):
    $\Delta = pkt.time - aq.last\_time$
    $aq.gap = \max(0, aq.gap - \Delta * aq.rate) + pkt.size$
    $aq.last\_time = pkt.time$
    return $aq.gap$
```

Algorithm 1 illustrates a streaming algorithm to calculate the A-Gap of an AQ for the arrival of every packet according to Theorem 3.2, regardless of whether the packets are in backlogged periods or empty periods.

3.3.2 A framework to control the traffic rate. We provide a framework with the AQ abstraction to limit the traffic rates and generate network feedback for different entities based on the discrepancy calculated with the A-Gap function. The pseudocode is shown in Algorithm 2.

Rate limiters. The AQ abstraction can provide rate-limiting for entities, regardless of underlying protocols (*i.e.*, TCP or UDP) and CC algorithms. As discussed in Section 3.2, each AQ has a maximum A-Gap $limit$, which guarantees that the arrival rate can converge to the allocated bandwidth R. For any incoming packet p_k, the A-Gap $A(p_k.time)$ is calculated with Theorem 3.2 when the packet arrives. If $A(p_k.time) > limit$, the AQ drops the packet, preventing it from entering the network (Lines 2-4 in Algorithm 2). Note that this limit is an intrinsic property of AQ, similar to the queue limit that restricts the number of packets or bytes in a physical queue.

Network feedback generation. The AQ abstraction can also generate congestion signals as the network feedback for different entities to adjust the traffic rates. Entities are allowed to specify their own CC algorithms and configurations based on their requirements. Different types of CC algorithms can be supported.

- Drop-based CC algorithms [17, 22, 40]. They are naturally supported by the AQ abstraction because packets are dropped when the A-Gap is greater than the AQ limit and no additional actions are needed.
- ECN-based CC algorithms [4, 18, 67]. They require ECN marked in packets as the network feedback to adjust the rate. An AQ can generate ECN signals based on the A-Gap and support different trigger conditions and marking actions (Lines 6-7). Take DCTCP as an example. With a physical queue, it triggers ECN marking operations when the physical queue length exceeds a threshold. An AQ can set a virtual threshold based on the A-Gap for ECN marking. When the A-Gap after a packet arrives exceeds the threshold, this packet is marked with ECN.
- Delay-based CC algorithms [34, 43]. They use queuing delay as the network feedback. For example, Swift measures the network delay and uses it to adjust traffic rates. However, the physical queuing delay is not a good indicator to provide network feedback when the network is shared by multiple entities, as discussed in Section 3.2. Analogous to a physical queue, given the A-Gap $A(k)$ for the k^{th} packet, the time it takes for an AQ to "drain" the A-Gap is $A(k)/R$, which is denoted as the *virtual queuing delay*. When a packet passes through a switch, the virtual queuing delay is calculated and piggybacked onto the packet header. The AQ abstraction accumulates the virtual queuing delay along the network path for a packet and uses it as the network feedback for delay-based CC algorithms to adjust the traffic rates.

Algorithm 2: The framework for traffic control

```
Function Generate_NFB($aq$, $pkt$):
    if $aq.gap > aq.limit$ then
        $aq.gap = aq.gap - pkt.size$
        Drop($pkt$)
    else
        if $aq.CC = ECN\_type$ then
            Apply_ECN_Marking_Actions($aq$, $pkt$)
        else if $aq.CC = delay\_type$ then
            Apply_Update_Delay_Actions($aq$, $pkt$)
        end
    end
```

4 APPLYING AUGMENTED QUEUES IN PRACTICE

Applying AQs in data center networks requires interactions among tenants, cloud operators, and switches. Figure 4 shows an architecture for how to deploy AQs on switches to provide network services for entities.

4.1 The Control Plane

The AQ Controller is managed by the cloud operator and it receives AQ requests from tenants. After an AQ request is granted, the AQ Controller configures the AQ and deploys it on the switch data plane accordingly.

AQ requests. There are three types of information in a request for an AQ: rate-related, CC-related, and position-related. Note that the AQ Controller allows an entity to have multiple AQs deployed in the network with multiple AQ requests.

- The rate-related information is to request the network bandwidth for an AQ. There are two modes for bandwidth allocation. In absolute mode, tenants specify their bandwidth requirements as absolute guarantees. In weighted mode, tenants specify their bandwidth requirements as network weights. Each AQ has a network weight and different AQs proportionally share the bandwidth of the network bottleneck link based on the corresponding weights.
- The CC-related information specifies the congestion control policy for an AQ. For example, if the applied congestion control is DCTCP [4], the tenant needs to inform the AQ Controller when the packets from this entity should be marked with ECN.
- The position-related information specifies the *position profile*, which indicates where an entity expects its traffic rate to be controlled. There are two positions in switches to deploy an

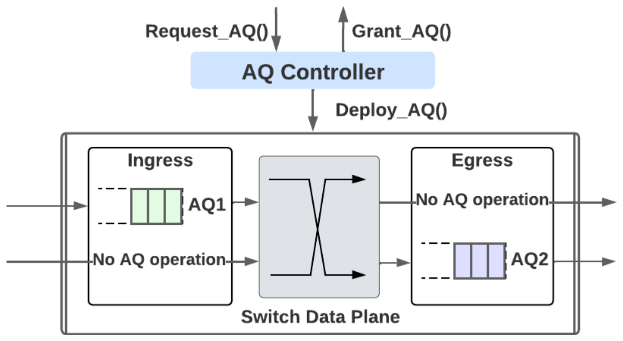


Figure 4: Applying AQ abstraction to networks.

AQ: ingress pipeline and egress pipeline. An AQ deployed at the ingress pipeline can control the rate of the traffic sent to all the egress ports; an AQ deployed at the egress pipeline can control the rate of the traffic from all the ingress ports. For example, if a VM requires a bi-directional bandwidth guarantee, it needs to send two AQ requests to the AQ Controller. One request specifies the position profile at the ingress to guarantee its outbound bandwidth; the other specifies the position profile at the egress to guarantee its inbound bandwidth.

We assume that tenants are aware of how to set these three types of information in an AQ request. In addition, the placement of AQ is out of the scope of this paper and the positions of AQ in switches are determined by the tenants.

AQ grants. After receiving an AQ request from a tenant, the AQ controller determines whether to grant or decline it based on the request information. If the bandwidth is allocated in absolute mode, the AQ Controller grants the request when the network link has sufficient bandwidth resources. Otherwise, it just declines the request. If the bandwidth is allocated in weighted mode, the AQ Controller determines (or updates) the specific bandwidth for each AQ based on their weights sharing the network link. The AQ Controller generates a unique ID for an AQ when it is granted and it also returns this ID to the tenant. The tenant needs to tag the AQ ID into the header of packets. It is possible that an entity has AQs deployed at both ingress and egress. Therefore, the tenant needs two fields in the packet header for the two AQ IDs, respectively. The field is set to a default value if there is no AQ deployed at either position. Either the VM hypervisor in each end host or applications of tenants can perform this tagging operation.

AQ deployments. The AQ Controller also generates an AQ configuration for its deployment in switches based on a granted AQ request. The fields of an AQ configuration are listed in Table 1. CC fields in the AQ request can be directly used for an AQ. AQ ID is uniquely generated for each AQ. AQ rate is the allocated rate R and it can be either directly specified in absolute mode or calculated with the network weights of entities. In order to compute the A-Gap according to Algorithm 1 and control traffic rates according to Algorithm 2, an AQ also needs extra fields, such as AQ limit, AQ gap, and AQ last_time. Deploying AQs at ingress or egress pipeline in switches can be either achieved through runtime operations provided by the control plane CLI [2], or through the recently proposed dynamic register memory-sharing mechanisms in the programmable data plane [66].

	AQ request	AQ configuration
Bandwidth demand	✓	
CC fields	✓	✓
Position profile	✓	
AQ ID		✓
AQ rate		✓
AQ limit		✓
AQ gap		✓
AQ last_time		✓

Table 1: Fields in AQ request and AQ configuration.

4.2 The Data Plane

An AQ can be deployed at either the ingress or the egress pipeline of the switch data planes, as shown in Figure 4. When a packet arrives at a switch, the switch needs to match the AQ at both ingress and egress because its AQ can be deployed at either position or both.

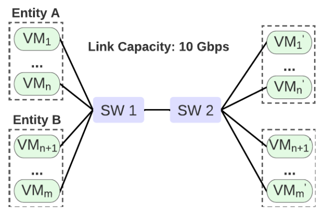
The switch first checks the AQ ID in the header of a packet for the ingress pipeline. If the ID is a default value, it indicates that the entity has no AQ deployed at the ingress and there is no additional AQ operation needed for this packet. Otherwise, the switch looks up the corresponding AQ based on the unique AQ ID. Each AQ updates its A-Gap based on the AQ rate, the time interval between the last and current packet, and the packet size on the arrival of a packet, according to Algorithm 1. If the conditions for limiting traffic rates or generating network feedback are triggered with the A-Gap, the switch takes corresponding actions based on the applied CC algorithms to inform the end hosts to adjust their traffic rates, according to Algorithm 2. When the packet is not dropped at the ingress, it is sent to the egress pipeline. The switch will then check the AQ ID in the packet header for the egress pipeline again and take the same steps as it did at the ingress.

5 EVALUATION

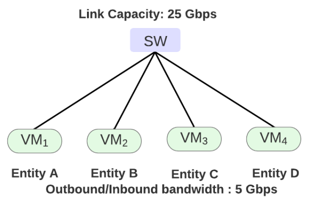
5.1 Experimental Setup

Platforms. We evaluate the augmented queue (AQ) abstraction with a prototype implemented in both a software simulator and a hardware testbed. The CloudLab platform [15] is used with 32GB RAM and eight-core 2.0GHz CPUs servers. The simulation environment is established using the BMv2 software switches in the NS3 simulator with 10Gbps network link bandwidth and 10us propagation delay. The testbed environment is established with a Tofino switch that has 32× 100Gbps ports. We can also configure it with 25 Gbps link speed. The topologies and divisions of the entities used for simulation and testbed are illustrated in Figure 5(a) and Figure 5(b), respectively.

Workloads. Our evaluation uses a web search workload trace that consists of a diverse mix of small and large TCP flows arriving at different times [4, 25]. The *traffic pattern*, *i.e.*, the traffic matrix among all hosts, is arbitrary because each host can communicate with any other hosts with arbitrary traffic volume at any different time, and each flow in the workload trace can be randomly generated by any source host and its traffic can be sent to any destination host. In addition, entities can also generate long-lived TCP and



(a) Topology and entity division on simulation.



(b) Topology and entity division on testbed.

Figure 5: Network topologies and entity divisions.

UDP flows. We apply five congestion control algorithms, including CUBIC, New Reno, Illinois, DCTCP, and Swift, to adjust the rates of TCP flows, respectively.

Baselines. We compare our AQ abstraction against both physical queues and rate-limiting solutions. A physical queue (PQ) is inherently equipped on the switch. There are two types of rate-limiting solutions: 1) pre-determined rate limiter (PRL), *i.e.*, the sending rates of VMs and applications are pre-determined; 2) dynamic rate limiter (DRL), *i.e.*, the rates are dynamically adjusted based on the traffic pattern. We use HTB [7] at end hosts as an example of the first type; we use ElasticSwitch [46] as an example of the second type and the rate adjustment interval is 15ms.

5.2 Network Performance of Applications

We will demonstrate that the AQ abstraction can allow distributed applications to fully utilize their allocated bandwidth and it can provide network isolation for different applications sharing a bottleneck link with the NS3 simulator. Each application is regarded as an entity.

AQ can fully utilize the allocated bandwidth. We evaluate the performance of one distributed application under different approaches. This application can involve different numbers of VMs to run the web search trace. All the generated flows share a network bottleneck link. In Figure 6, the workload completion time with different approaches is normalized to PQ under which the network is fully utilized. We observe that AQ can achieve a very close workload completion time as PQ, indicating that AQ can also fully utilize the network bandwidth. However, the completion time with both PRL and DRL keeps increasing with the number of VMs. PRL evenly allocates the bottleneck bandwidth among VMs and the allocation is fixed during the lifecycle of an entity. However, because the traffic pattern is arbitrary and it changes over time, *i.e.*, any source and destination pair can transmit flows with different traffic volumes at different times, the bandwidth demand at runtime inevitably mismatches the allocated bandwidth. Some VMs do not have enough traffic to saturate the allocated bandwidth, but other VMs still cannot use their spare bandwidth due to the fixed allocation, leading to an underutilized network with PRL. Although DRL can dynamically adjust the bandwidth allocation, it requires a time interval, *e.g.*, 15 millisecond [46], for the adjustment. Because the runtime bandwidth demand of each VM can change dramatically, the allocated bandwidth cannot match its demand changes, leading to a sub-optimal workload completion time.

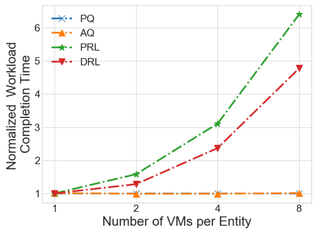


Figure 6: The normalized workload completion time of an entity under different numbers of VMs.

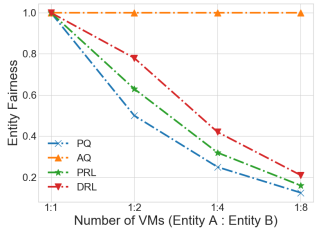


Figure 7: The entity fairness under different mechanisms when different entities have different numbers of VMs.

AQ can isolate different applications. We evaluate the performance of multiple applications sharing a bottleneck under different approaches. Each application expects to share the network bandwidth according to the allocated network weights, regardless of the number of VMs, the number of TCP flows, and the protocol types (*i.e.*, TCP or UDP).

- **Different numbers of VMs.** In this experiment, we have two entities, entity A and entity B. Entity A has one VM and entity B can have one or multiple VMs. They both run the web search trace and all the generated flows share a network bottleneck link. The two entities have the same network weight and they expect to fairly share the network bandwidth. PRL evenly distributes the allocated bandwidth of an entity among its VMs. We define the *entity fairness* as the ratio of the shorter workload completion time between entity A and entity B to the longer one. The expected entity fairness is 1. Figure 7 displays the entity fairness under different approaches. We observe that the entity fairness of all the compared baselines decreases with the increasing number of VMs in entity B. When entity B has eight VMs, entity A's completion time is only 0.16× and 0.21× entity B's completion time using the PRL and DRL, respectively. This is because the allocated bandwidth of entity B cannot be always fully utilized when there are multiple VMs in an entity. When using PQ, entity A's completion time is around 7.2× entity B's completion time due to the flow-level fair share. In contrast, the entity fairness with AQ is always around 1, regardless of the number of VMs.

- **Different numbers of TCP flows.** In this experiment, we have two entities, entity A and entity B, and both of them have one VM. Entity A generates one TCP flow and entity B generates one or multiple TCP flows. Figure 8 shows the throughput of the two entities using PQ and AQ, respectively. With PQ, the network bandwidth shared between the two entities corresponds to their number of flows. They can evenly share the network when they both have one TCP flow. However, when entity B increases its number of flows to 64, it can grab most of the bandwidth and starve entity A. It implies that an entity can arbitrarily seize the network bandwidth by generating more flows with PQ. In contrast, AQ guarantees that the two entities can share the bandwidth according to their network weights, regardless of their different number of flows. For example, the two entities evenly share the bandwidth when they have the same weight and their throughput is 1:2 when their weights are 1:2.

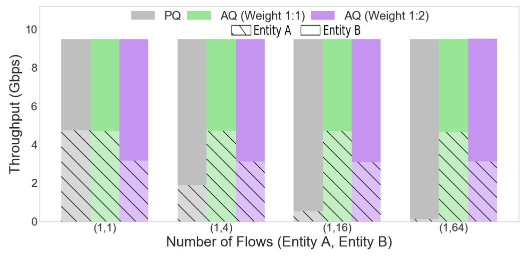
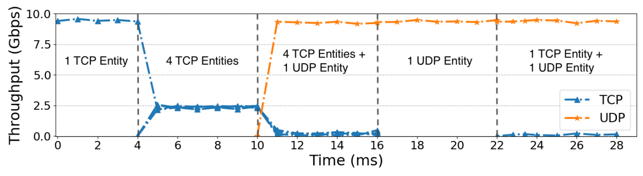
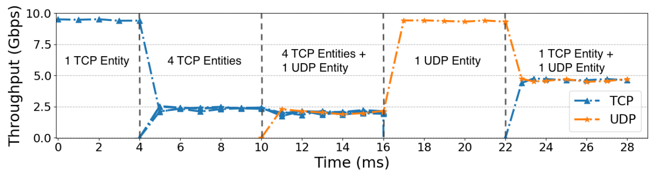


Figure 8: Throughput of different entities with different numbers of flows.



(a) Using PQ



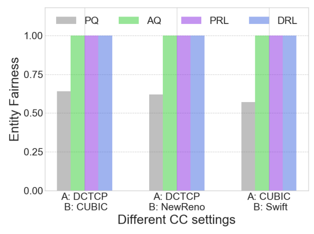
(b) Using AQ

Figure 9: Throughput of TCP and UDP entities.

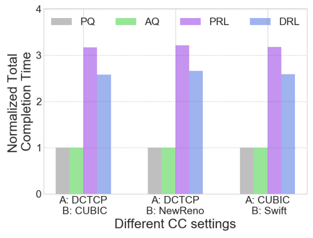
• **Different protocol types (UDP vs. TCP).** In this experiment, we run several entities with the same network weights and they expect to equally share the bottleneck link. Each entity has one VM that generates either a TCP or a UDP flow. The sending rates of each entity with UDP flows equal the link capacity. Figure 9(a) displays the throughput of UDP and TCP entities with PQ. When there are only TCP entities, they can fairly share the network bandwidth. However, when a UDP entity begins injecting traffic into the network, it almost grabs all the bandwidth and starves TCP entities. In contrast, AQ can provide network bandwidth isolation to entities. As shown in Figure 9(b), the throughput of each entity is always around $1/n$ of the link capacity with n active entities sharing the bottleneck link regardless of UDP or TCP entities. Specifically, when there are four TCP entities and one UDP entity, the allocated bandwidth of each entity is 2Gbps and their achieved bandwidth saturation is over 95%. It indicates that AQ can provide network isolation for entities and fully utilize the network bandwidth regardless of protocol types.

5.3 Network Performance of CCs

We will demonstrate that the AQ abstraction can provide network isolation for entities with different CC algorithms when their traffic shares a bottleneck link using the NS3 simulator. All the entities have the same network weights and they expect to fairly share the network bandwidth.



(a) The entity fairness.



(b) The total completion time.

Figure 10: Entity fairness and total workload completion time under different CC settings.

Workload completion time. In this experiment, we have two entities, entity A and entity B. Each entity has four VMs and runs the web search trace. Figure 10(a) shows the entity fairness between the two entities with different CC algorithms and different approaches. The entity fairness with AQ, PRL, and DRL is very close to 1, indicating that the two entities have almost the same throughput. However, the entity fairness with PQ is just around 0.6. This is because entity A can grab most of the bandwidth when it shares the network with entity B until it completes its workload. The complete time of entity B is much longer than that of entity A. Figure 10(b) compares the total completion time of the two entities. AQ and PQ have the same completion time because they both can fully utilize the network bandwidth. However, despite enforcing entity fairness, both PRL and DRL have a significantly longer completion time due to the under-utilization of the network caused by rate-limiting solutions, as discussed in Section 5.2.

Bandwidth sharing. In this experiment, we use two entities that generate the same number of flows. With PQ, they can evenly share the network when they apply the same CC algorithm. However, their throughput has a disparity with different CC algorithms. As listed in Table 2, the DCTCP traffic can aggressively seize the bandwidth and starve traffic from other CC algorithms; the CUBIC traffic can also starve the Swift traffic. In contrast, with AQ, the two entities can always achieve similar throughput, regardless of the CC algorithms applied. They can still fairly share the bandwidth even when they have different numbers of flows. We also use four entities for the evaluation. One of them generates a UDP flow and the other three entities generate three TCP flows, respectively, with different CC algorithms. With PQ, the UDP flow can grab a dominant share of the bandwidth and its throughput is 8.9Gbps, but the total throughput of the TCP flows is only 0.4Gbps. With AQ, all four entities can have almost the same throughout, demonstrating the fair sharing of the network.

5.4 Network Performance of VMs

We will demonstrate that AQ can provide a bi-directional absolute bandwidth guarantee for VMs in the network with the Tofino testbed. The network topology is illustrated in Figure 2. Four VMs are connected to a switch and the network link capacity is 25Gbps. VM A has a traffic profile with a 5Gbps outbound bandwidth and a 5Gbps inbound bandwidth. It runs the web search trace and the destinations are other three VMs. VMs B, C, and D also run the web search trace and their destinations are VM A. In PRL, the rate

Congestion control	PQ	AQ
5 CUBIC+5 CUBIC	4.7Gbps+4.7Gbps	4.7Gbps+4.7Gbps
5 CUBIC+5 DCTCP	0.7Gbps+8.7Gbps	4.6Gbps+4.7Gbps
5 NewReno+5 DCTCP	0.5Gbps+8.9Gbps	4.7Gbps+4.7Gbps
5 Illinois+5 DCTCP	1.7Gbps+7.7Gbps	4.6Gbps+4.7Gbps
5 CUBIC+5 Swift	9.1Gbps+0.2Gbps	4.7Gbps+4.6Gbps
5 DCTCP+5 Swift	9.2Gbps+0.1Gbps	4.7Gbps+4.6Gbps
10 DCTCP+5 NewReno	9.1Gbps+0.3Gbps	4.7Gbps+4.7Gbps
10 DCTCP+5 Swift	9.2Gbps+0.1Gbps	4.7Gbps+4.6Gbps
1 UDP+3 CUBIC +3 DCTCP+3 Swift	8.9Gbps+0.1Gbps +0.2Gbps+0.1Gbps	2.4Gbps+2.3Gbps +2.4Gbps+2.2Gbps

Table 2: Throughput of entities with different CC settings.

Approaches	Outbound Rate Range	Inbound Rate Range
Ideal	5Gbps	5Gbps
PQ-testbed	23.1Gbps ~ 23.6Gbps	23.2Gbps ~ 23.6Gbps
PRL-testbed	4.8Gbps ~ 5.1Gbps	14.6Gbps ~ 15.3Gbps
DRL-testbed	3.1Gbps ~ 4.9Gbps	3.3Gbps ~ 4.8Gbps
AQ-testbed	4.9Gbps ~ 5.2Gbps	4.8Gbps ~ 5.2Gbps
AQ-simulator	4.8Gbps ~ 5.3Gbps	4.9Gbps ~ 5.1Gbps

Table 3: The outbound and inbound rates of VM A with different approaches.

limiters for the four VMs are 5Gbps. In DRL, the outbound and inbound bandwidth allocated to VM A is 5Gbps; the bandwidth allocation for VMs B, C, and D is dynamically adjusted.

Table 3 lists the range of the outbound rate and inbound rate of VM A with different approaches. We observe that PQ is unable to limit the outbound rate and inbound rate to the allocated bandwidth. Both rates are much higher than the 5Gbps expected rate because the rates can keep increasing until they reach the link capacity and are restricted by congestion. The outbound rate with PRL is around 5Gbps, but its inbound rate is around 15Gbps because three VMs send traffic to VM A simultaneously. It violates the inbound bandwidth required by the traffic profile. DRL can adjust the VM traffic rates to attempt to satisfy the traffic profile, but the traffic pattern can change dramatically. Therefore, both the outbound and inbound rates can be less than 5Gbps when the real demand is lower than the allocated rate. In contrast, AQ can guarantee that both rates of VM A are around 5Gbps, indicating that it can satisfy the traffic profile. We also measure the performance of AQ on simulation using the same setup. The results are consistent with those obtained in the testbed, confirming the fidelity of the AQ.

5.5 Other Factors

CC behaviors in AQ and PQ. We demonstrate with the testbed that the AQ abstraction can preserve a CC's traffic behaviors, such as the throughput and queuing delay. Specifically, for an entity using AQ with a specified bandwidth and a CC algorithm, its traffic should behave as if it was running in a physical network exclusively with the same network bandwidth. We measure the throughput and the queuing delay distribution of an entity with AQ and PQ, respectively. For AQ, its allocated bandwidth is 25Gbps in a 100Gbps network; for PQ, we configure the link bandwidth to 25Gbps. Their performance comparison under different CC algorithms is listed in Table 4. We can observe that an entity can achieve almost the same

	PQ		AQ	
	Throughput	95% delay	Throughput	95% delay
CUBIC	23.6Gbps	698us	23.6Gbps	687us
New Reno	23.6Gbps	721us	23.6Gbps	712us
DCTCP	23.5Gbps	88us	23.6Gbps	86us

Table 4: Comparisons between AQ and PQ in terms of throughput and queuing delay on the Tofino testbed.

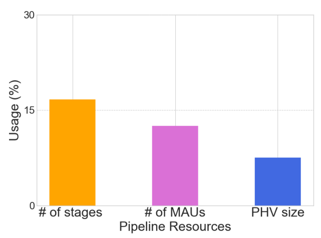


Figure 11: Usage of data plane resources on Tofino switch.

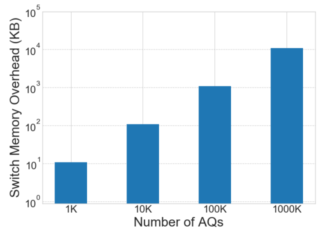


Figure 12: Memory consumption with different numbers of traffic constituents.

throughput under the PQ and AQ environments. For example, the throughputs with CUBIC are both 23.6Gbps using PQ and AQ. In addition, we use the virtual queuing delay (refer to Section 3.3.2) as the queuing delay with AQ. We can observe that the virtual queuing delay distribution is very close to the experienced queuing delay distribution in a physical network. For example, the relative difference between their 95th percentage is less than 2.3% under the three applied CC algorithms.

Switch resource overhead. We demonstrate that our AQ abstraction can be feasibly deployed in the hardware switches. Figure 11 shows the resource usage of AQ on the switch data plane of our Tofino testbed. We can observe that only a very small percentage of each resource type is required on our testbed, *e.g.*, 16.8% for pipeline stages, 12.5% for MAUs, and 7.5% for PHV size. Under the scale of production networks, existing mechanisms [51, 57] can further shrink AQ's resource usage. Figure 12 displays the switch memory consumption with respect to the number of deployed AQs. In our current implementation, we use four bytes for each AQ ID to support millions of entities and three bytes for the AQ rate to support a wide range of bandwidth allocations, *e.g.*, 1MB ~ 1TB. Each AQ requires 15 bytes in total based on the fields listed in Table 1. We can see that today's programmable switches, which usually have tens of MB of memory, can comfortably support millions of concurrent AQs.

6 DISCUSSIONS

Work conservation. This paper aims to provide precise bandwidth guarantees for different traffic constituents. Referring to Section 2.3, the inbound and outbound bandwidth guarantees for VMs are contradictory to work conservation in the network. In order to accommodate scenarios that require work conservation, one possible mechanism is to invoke AQ only when the physical queue starts to build up in switches. When the physical queue is empty, the switch can bypass AQ to allow traffic constituents to grab higher network bandwidth than their allocations for better network utilization; when there is queuing in physical queues, AQ is invoked to limit the traffic rate and reduce traffic interference. Another possible mechanism is to dynamically adjust the allocated bandwidth of traffic constituents with the AQ abstraction. For example, network providers can measure their arrival rates in the network and then allow AQ to periodically recompute their allocated bandwidth according to similar rate control algorithms proposed in EyeQ [29] and Seawall [53]. When there is available bandwidth, this mechanism can increase the allocated bandwidth of entities to better utilize the network. It will be interesting future work to achieve work conservation with other mechanisms based on the AQ abstraction.

AQ limit configurations. The AQ limit of each entity should absorb its traffic burst and allow its traffic to achieve the allocated bandwidth. A possible configuration mechanism is to set the AQ limit as the physical queue (PQ) limit. It allows entities to configure their CC fields in the same way as the settings for PQ. However, because the total AQ limit of all entities is greater than the PQ limit, there would be situations in which packets are scheduled in AQs but dropped at PQs, leading to inaccurate network feedback generated in Algorithm 2. An alternative to setting the AQ limits is to divide the PQ limit proportionally according to the allocated bandwidth. It ensures that the total number of packets queued in the AQs of all entities is no greater than that queued in the physical queue when the network is congested. In this way, the overall traffic behaviors of AQs can be consistent with the physical queue behaviors. However, low allocated bandwidth can lead to a small AQ limit, which might hinder the entity to achieve its allocated bandwidth due to excess packet drops. It might require network providers to determine the minimum AQ limit under which the packets are not overly dropped and the allocated bandwidth is achievable. This minimum AQ limit is empirical-driven and it depends on multiple factors, such as CC algorithms and the allocated bandwidth. We will take the exploration of setting AQ limits as our future work.

7 RELATED WORK

Rate-limiting based solutions. Many rate-limiting solutions have been proposed to control the traffic rate and can be divided into two categories: pre-determined rate limiters and dynamic rate limiters. Pre-determined rate limiters [7, 35, 48] limit the outbound traffic rate of VMs according to a fixed rate configuration, but they cannot accommodate arbitrary traffic patterns, especially for distributed applications that involve multiple VM. Dynamic rate limiters [8, 21, 28, 29, 36, 45–47, 53, 61] can periodically adjust the rate configuration for each VM. However, the inbound bandwidth specification of a VM can be either under-utilized or violated depending on the traffic pattern among VMs. Moreover, they cannot provide precise bandwidth guarantees for the transport layer. In contrast, AQ is an in-network solution and it can provide precise bandwidth guarantees for the application layer, the transport layer, and the link layer.

Fair-queuing based solutions. Some existing fair queuing mechanisms can mitigate traffic interference among traffic constituents and limit the traffic rate in case of congestion. Conventional fair queuing algorithms, such as WFQ [9], DRR [54] and Stochastic Fair Queuing [42], require per-flow queue. However, the limited number of physical queues in today's commodity switches hinders their deployment. Several approximate solutions, such as PIFO [56], SP-PIFO [3], AIFO [64], and AFO [52], are proposed to approximate fair queuing with one or a few physical queues. They rely on network feedback for traffic control. However, they cannot provide inbound bandwidth guarantees for VMs because they cannot bound the traffic rate when there is no network congestion. Furthermore, they are unable to generate different CC signals for different CC algorithms simultaneously at the transport layer. In contrast, AQ is scalable to today's data center with only one physical queue and supports precise bandwidth guarantees for the application layer, the transport layer, and the link layer.

Congestion Control algorithms. End-to-end CC algorithms [4, 17, 22, 34, 40, 43, 67] allow fair sharing of the network, but they cannot support flexible bandwidth guarantees. Moreover, they are unable to control the traffic rate when there is no congestion in the network. In contrast, AQ can support flexible bandwidth guarantees and its traffic rate control is independent of the physical queue length. It focuses on supporting the coexistence of multiple CC algorithms, rather than proposing new ones. AQ can support different types of CC algorithms, such as drop-based CC algorithms [17, 22, 40], ECN-based CC algorithms [4, 67], and Delay-based CC algorithms [34, 43]. Other CC algorithms, such as TCP BBR [12], HPCC [39] and HULL [5], can also accommodate to the AQ abstraction. TCP BBR relies on both the arrival rate and the delay to conduct the rate adjustments; our AQ abstraction is capable of providing such information in the network. HULL and HPCC leverage bandwidth headroom to reduce the latency for network applications; AQ can also set aside an amount of bandwidth in the network as the bandwidth headroom. Several virtualized congestion control solutions, such as AC/DC TCP [24] and vCC [13], can mitigate the traffic interference of different CC algorithms. However, they focus on introducing a new unified DCTCP-like congestion control and forcing all other congestion control algorithms to perform the same. In contrast, AQ allows different CC algorithms to perform differently according to different types of network feedback signals.

8 CONCLUSION

In this paper, we analyze the fundamental limitations of physical queues for data center network sharing. We then propose Augmented Queue (AQ), an in-network abstraction to scalably provide precise bandwidth guarantees for different traffic constituents. AQ allows traffic constituents to adjust their traffic rates only based on their own traffic, regardless of the applied protocols, the CC algorithms, and the traffic patterns. We prototype AQ with programmable switches and demonstrate that it can provide precise bandwidth guarantees and scale to today's data centers. We believe that AQ's underlying concept, which relieves the reliance on physical queues, has significant potential for data center network sharing and could serve as a guide for future network development. This work does not raise any ethical issues.

ACKNOWLEDGMENT

We would like to thank our shepherd Justine Sherry and the anonymous reviewers for providing valuable feedback. This work is partially supported by the NSF under CNS-2214272 and CNS-1815525.

REFERENCES

[1] 2022. Amazon Elastic Compute Cloud (Amazon EC2). (2022). http://aws.amazon.com/ec2/

[2] 2022. Behavior Model of Programmable Switches. https://github.com/p4lang/behavioral-model. (2022).

[3] Albert Gran Alcoz, Alexander Dietmüller, and Laurent Vanbever. 2020. SP-PIFO: Approximating Push-In First-Out Behaviors using Strict-Priority Queues. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 59–76.

[4] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*. 63–74.

[5] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. 2012. Less Is More: Trading a Little Bandwidth for Ultra-Low Latency in the Data Center. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. 253–266.

[6] Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. 2016. SNAP: Stateful network-wide abstractions for packet processing. In *Proceedings of the 2016 ACM SIGCOMM Conference*. 29–43.

[7] Doru Gabriel Balan and Dan Alin Potorac. 2009. Linux HTB queuing discipline implementations. In *2009 First International Conference on Networked Digital Technologies*. IEEE, 122–126.

[8] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. 2011. Towards predictable datacenter networks. In *Proceedings of the ACM SIGCOMM 2011 Conference*. 242–253.

[9] Albert Banchs and Xavier Perez. 2002. Distributed weighted fair queuing in 802.11 wireless LAN. In *2002 IEEE International Conference on Communications. Conference Proceedings. ICC 2002 (Cat. No. 02CH37333)*, Vol. 5. IEEE, 3121–3127.

[10] Manu Bansal, Jeffrey Mehlman, Sachin Katti, and Philip Levis. 2012. Openradio: a programmable wireless dataplane. In *Proceedings of the first workshop on Hot topics in software defined networks*. 109–114.

[11] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. *ACM SIGCOMM Computer Communication Review* 43, 4 (2013), 99–110.

[12] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. 2016. Bbr: Congestion-based congestion control: Measuring bottleneck bandwidth and round-trip propagation time. *Queue* 14, 5 (2016), 20–53.

[13] Bryce Cronkite-Ratcliff, Aran Bergman, Shay Vargaftik, Madhusudhan Ravi, Nick McKeown, Ittai Abraham, and Isaac Keslassy. 2016. Virtualized congestion control. In *Proceedings of the 2016 ACM SIGCOMM Conference*. 230–243.

[14] Nick G Duffield, Pawan Goyal, Albert Greenberg, Partho Mishra, Kadangode K Ramakrishnan, and Jacobus E van der Merive. 1999. A flexible model for resource management in virtual private networks. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*. 95–108.

[15] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. https://www.flux.utah.edu/paper/duplyakin-atc19. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 1–14.

[16] Thomas Erlebach and Maurice Ruegg. 2004. Optimal bandwidth reservation in hose-model VPNs with multi-path routing. In *IEEE INFOCOM 2004*, Vol. 4. IEEE, 2275–2282.

[17] Sally Floyd, Tom Henderson, and Andrei Gurtov. 2004. *The NewReno modification to TCP's fast recovery algorithm*. Technical Report.

[18] Sally Floyd and Van Jacobson. 1993. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on networking* 1, 4 (1993), 397–413.

[19] Tommaso Frassetto, Patrick Jauernig, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2018. {IMIX}:{In-Process} Memory Isolation {EXtension}. In *27th USENIX Security Symposium (USENIX Security 18)*. 83–97.

[20] Peter Gomber and Martin Haferkorn. 2015. High frequency trading. In *Encyclopedia of Information Science and Technology, Third Edition*. IGI Global, 1–9.

[21] Chuanxiong Guo, Guohan Lu, Helen J Wang, Shuang Yang, Chao Kong, Peng Sun, Wenfei Wu, and Yongguang Zhang. 2010. Secondnet: a data center network virtualization architecture with bandwidth guarantees. In *Proceedings of the 6th International COnference*. 1–12.

[22] Sangtae Ha, Injong Rhee, and Lisong Xu. 2008. CUBIC: a new TCP-friendly high-speed TCP variant. *ACM SIGOPS operating systems review* 42, 5 (2008), 64–74.

[23] David Hancock and Jacobus Van der Merwe. 2016. Hyper4: Using p4 to virtualize the programmable data plane. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*. 35–49.

[24] Keqiang He, Eric Rozner, Kanak Agarwal, Yu Gu, Wes Felter, John Carter, and Aditya Akella. 2016. AC/DC TCP: Virtual congestion control enforcement for datacenter networks. In *Proceedings of the 2016 ACM SIGCOMM Conference*. 244–257.

[25] Kuo-Feng Hsu, Ryan Beckett, Ang Chen, Jennifer Rexford, Praveen Tammana, and David Walker. 2020. Contra: A programmable system for performance-aware routing. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020*.

[26] Chengchen Hu, Yi Tang, Xuefei Chen, and Bin Liu. 2007. Per-flow queueing by dynamic queue sharing. In *IEEE INFOCOM 2007-26th IEEE International Conference on Computer Communications*. IEEE, 1613–1621.

[27] Călin Iorgulescu, Reza Azimi, Youngjin Kwon, Sameh Elnikety, Manoj Syamala, Vivek Narasayya, Herodotos Herodotou, Paulo Tomita, Alex Chen, Jack Zhang, et al. 2018. {PerfIso}: Performance Isolation for Commercial {Latency-Sensitive} Services. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 519–532.

[28] Keon Jang, Justine Sherry, Hitesh Ballani, and Toby Moncaster. 2015. Silo: Predictable message latency in the cloud. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. 435–448.

[29] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, Balaji Prabhakar, Albert Greenberg, and Changhoon Kim. 2013. {EyeQ}: Practical Network Performance Isolation at the Edge. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. 297–311.

[30] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. 2020. A Unified Architecture for Accelerating Distributed {DNN} Training in Heterogeneous GPU/CPU Clusters. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*. 463–479.

[31] Leonard Kleinrock. 1976. Computer applications. *Queueing systems* 3 (1976).

[32] Alok Kumar, Sushant Jain, Uday Naik, Anand Raghuraman, Nikhil Kasinadhuni, Enrique Cauich Zermeno, C Stephen Gunn, Jing Ai, Björn Carlin, Mihai Amarandei-Stavila, et al. 2015. BwE: Flexible, hierarchical bandwidth allocation for WAN distributed computing. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. 1–14.

[33] Amit Kumar, Rajeev Rastogi, Avi Silberschatz, and Bulent Yener. 2002. Algorithms for provisioning virtual private networks in the hose model. *IEEE/ACM transactions on networking* 10, 4 (2002), 565–578.

[34] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan MG Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, et al. 2020. Swift: Delay is simple and effective for congestion control in the datacenter. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 514–528.

[35] Praveen Kumar, Nandita Dukkipati, Nathan Lewis, Yi Cui, Yaogong Wang, Chonggang Li, Valas Valancius, Jake Adriaens, Steve Gribble, Nate Foster, et al. 2019. PicNIC: predictable virtualized NIC. In *Proceedings of the ACM Special Interest Group on Data Communication*. 351–366.

[36] Vinh The Lam, Sivasankar Radhakrishnan, Rong Pan, Amin Vahdat, and George Varghese. 2012. Netshare and stochastic netshare: predictable bandwidth allocation for data centers. *ACM SIGCOMM Computer Communication Review* 42, 3 (2012), 5–11.

[37] Ang Li, Xiaowei Yang, Srikanth Kandula, and Ming Zhang. 2010. CloudCmp: comparing public cloud providers. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*. 1–14.

[38] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. 2020. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704* (2020).

[39] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, et al. 2019. HPCC: High precision congestion control. In *Proceedings of the ACM Special Interest Group on Data Communication*. 44–58.

[40] Shao Liu, Tamer Başar, and Ravi Srikant. 2006. TCP-Illinois: A loss and delay-based congestion control algorithm for high-speed networks. In *Proceedings of the 1st international conference on Performance evaluation methodolgies and tools*. 55–es.

[41] James McCauley, Aurojit Panda, Arvind Krishnamurthy, and Scott Shenker. 2019. Thoughts on load distribution and the role of programmable switches. *ACM SIGCOMM Computer Communication Review* 49, 1 (2019), 18–23.

[42] Paul E McKenney. 1990. Stochastic fairness queueing. In *IEEE INFOCOM'90*. IEEE Computer Society, 733–734.

[43] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. 2015. TIMELY: RTT-based congestion control for the datacenter. *ACM SIGCOMM Computer Communication Review* 45, 4 (2015), 537–550.

[44] Abhay K Parekh and Robert G Gallager. 1993. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM transactions on networking* 1, 3 (1993), 344–357.

[45] Lucian Popa, Gautam Kumar, Mosharaf Chowdhury, Arvind Krishnamurthy, Sylvia Ratnasamy, and Ion Stoica. 2012. FairCloud: Sharing the network in cloud computing. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*. 187–198.

[46] Lucian Popa, Praveen Yalagandula, Sujata Banerjee, Jeffrey C Mogul, Yoshio Turner, and Jose Renato Santos. 2013. Elasticswitch: Practical work-conserving bandwidth guarantees for cloud computing. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*. 351–362.

[47] Henrique Rodrigues, Jose Renato Santos, Yoshio Turner, Paolo Soares, and Dorgival Guedes. 2011. Gatekeeper: Supporting bandwidth guarantees for multi-tenant datacenter networks. In *3rd Workshop on I/O Virtualization (WIOV 11)*.

[48] Ahmed Saeed, Nandita Dukkipati, Vytautas Valancius, Vinh The Lam, Carlo Contavalli, and Amin Vahdat. 2017. Carousel: Scalable traffic shaping at end hosts. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. 404–417.

[49] Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. 2010. Runtime measurements in the cloud: observing, analyzing, and reducing variance. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 460–471.

[50] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799* (2018).

[51] Naveen Kr Sharma, Antoine Kaufmann, Thomas E Anderson, Arvind Krishnamurthy, Jacob Nelson, and Simon Peter. 2017. Evaluating the Power of Flexible Packet Processing for Network Resource Allocation.. In *NSDI*. 67–82.

[52] Naveen Kr Sharma, Chenxingyu Zhao, Ming Liu, Pravein G Kannan, Changhoon Kim, Arvind Krishnamurthy, and Anirudh Sivaraman. 2020. Programmable calendar queues for high-speed packet scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 685–699.

[53] Alan Shieh, Srikanth Kandula, Albert Greenberg, Changhoon Kim, and Bikas Saha. 2011. Sharing the data center network. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*.

[54] Madhavapeddi Shreedhar and George Varghese. 1995. Efficient fair queueing using deficit round robin. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*. 231–242.

[55] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, et al. 2015. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. *ACM SIGCOMM computer communication review* 45, 4 (2015), 183–197.

[56] Anirudh Sivaraman, Keith Winstein, Suvinay Subramanian, and Hari Balakrishnan. 2013. No silver bullet: extending SDN to the data plane. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in networks*. 1–7.

[57] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, Shan Muthukrishnan, and Jennifer Rexford. 2017. Heavy-hitter detection entirely in the data plane. In *Proceedings of the Symposium on SDN Research*. 164–176.

[58] Brent E Stephens, Darius Grassi, Hamidreza Almasi, Tao Ji, Balajee Vamanan, and Aditya Akella. 2021. TCP is Harmful to In-Network Computing: Designing a Message Transport Protocol (MTP). In *Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks*. 61–68.

[59] Vineeth Sagar Thapeta, Komal Shinde, Mojtaba Malekpourshahraki, Darius Grassi, Balajee Vamanan, and Brent E Stephens. 2021. Nimble: Scalable tcp-friendly programmable in-network rate-limiting. In *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)*. 27–40.

[60] Shuai Wang, Kaihui Gao, Kun Qian, Dan Li, Rui Miao, Bo Li, Yu Zhou, Ennan Zhai, Chen Sun, Jiaqi Gao, et al. 2022. Predictable vFabric on informative data plane. In *Proceedings of the ACM SIGCOMM 2022 Conference*. 615–632.

[61] Di Xie, Ning Ding, Y Charlie Hu, and Ramana Kompella. 2012. The only constant is change: Incorporating time-varying network reservations in data centers. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*. 199–210.

[62] Yunjing Xu, Zachary Musgrave, Brian Noble, and Michael Bailey. 2013. Bobtail: Avoiding long tails in the cloud. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. 329–341.

[63] Liangcheng Yu, John Sonchack, and Vincent Liu. 2022. Cebinae: scalable in-network fairness augmentation. In *Proceedings of the ACM SIGCOMM 2022 Conference*. 219–232.

[64] Zhuolong Yu, Chuheng Hu, Jingfeng Wu, Xiao Sun, Vladimir Braverman, Mosharaf Chowdhury, Zhenhua Liu, and Xin Jin. 2021. Programmable packet scheduling with a single queue. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. 179–193.

[65] Hui Zhang and Jon CR Bennett. 1996. WF2Q: worst-case fair weighted fair queueing. In *IEEE INFOCOM*, Vol. 96. 120–128.

[66] Hang Zhu, Tao Wang, Yi Hong, Dan RK Ports, Anirudh Sivaraman, and Xin Jin. 2022. NetVRM: Virtual Register Memory for Programmable Networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 155–170.

[67] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. 2015. Congestion control for large-scale RDMA deployments. *ACM SIGCOMM Computer Communication Review* 45, 4 (2015), 523–536.

[68] Yibo Zhu, Monia Ghobadi, Vishal Misra, and Jitendra Padhye. 2016. ECN or Delay: Lessons Learnt from Analysis of DCQCN and TIMELY. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*. 313–327.