EyeQ: Practical Network Performance Isolation at the Edge

Vimalkumar Jeyakumar[1], Mohammad Alizadeh[1,2], David Mazières[1], Balaji Prabhakar[1], Changhoon Kim[3], and Albert Greenberg[3]

[1] *Stanford University* [2] *Insieme Networks* [3] *Windows Azure*

Abstract

The datacenter network is shared among untrusted tenants in a public cloud, and hundreds of services in a private cloud. Today we lack fine-grained control over network bandwidth partitioning across tenants. In this paper we present EyeQ, a simple and practical system that provides tenants with bandwidth guarantees as if their endpoints were connected to a dedicated switch. To realize this goal, EyeQ leverages the high bisection bandwidth in a datacenter fabric and enforces admission control on traffic, regardless of the tenant transport protocol. We show that this pushes bandwidth contention to the network's edge, enabling EyeQ to support end-to-end minimum bandwidth guarantees to tenant endpoints in a simple and scalable manner at the servers. EyeQ requires no changes to applications and is deployable with support from the network available today. We evaluate EyeQ with an efficient software implementation at 10Gb/s speeds using unmodified applications and adversarial traffic patterns. Our evaluation demonstrates EyeQ's promise of predictable network performance isolation. For instance, even with an adversarial tenant with bursty UDP traffic, EyeQ is able to maintain the 99.9th percentile latency for a collocated memcached application close to that of a dedicated deployment.

1 Introduction

In the datacenter, we seek to virtualize the network for its tenants, as has been done for compute and storage. Ideally, a tenant running on shared physical infrastructure should see the same range of control- and data-path capabilities on its virtual network, as it would see on a dedicated physical network. This vision has been in full swing for some years in the control plane [1, 2]. An early innovator in the control plane was Amazon Web Services, where a tenant can create a "Virtual Private Cloud" [1] with their IP addresses without interfering with other tenants. In the data plane, there has been little comparable progress.

To make comparable progress, we posit that the provider should present a simple performance abstraction of a dedicated switch connecting a tenant's endpoints [3], independent of the underlying physical topology. The endpoints may be anywhere in the datacenter, but a tenant should be able to attain full line rate for any traffic pattern between its endpoints, constrained only by endpoint capacities. Bandwidth assurances to this tenant should suffer no negative impact from the behavior and churn of other tenants in the datacenter. This abstraction has been a consistent ask of enterprise customers considering moving to the cloud, as the enterprise mission demands a high degree of infrastructure predictability [4].

Is this abstraction realizable? EyeQ described in this paper attempts to deliver this abstraction for every tenant. This requires three key components of which EyeQ provides the final missing piece.

First, with little to no knowledge of tenant communication patterns, promising bandwidth guarantees to endpoints requires smart endpoint placement in a network with adequate capacity (for the worst case). Hence, topologies with bottlenecks between server–server ("east–west") traffic are undesirable. Fortunately, recent proposals [5, 6, 7] have demonstrated cost-effective means of building "high bisection bandwidth" network topologies. These topologies are realizable in practice (§2.3), and substantially lower the complexity of endpoint placement (§3.5) as server–server capacity is more uniform. Second, utilizing this high bisectional bandwidth requires effective traffic load balancing schemes to mitigate network hotspots. While today's routing protocols (e.g. Equal-Cost Multi-Path [8]) do a reasonable job of utilizing available capacity, there has

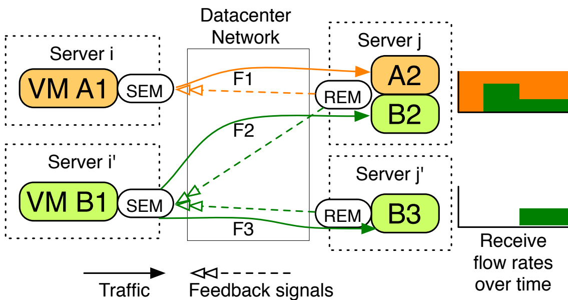


Figure 1: EyeQ's sender module (SEM) and receiver module (REM) work in a distributed fashion by exchanging feedback messages, to iteratively converge to bandwidth guarantees.

been continuous improvements on this front [9, 10, 11].

The third (and missing) piece, is a bandwidth arbitration mechanism that schedules tenant flows in accordance with the bandwidth guarantees, *even with misbehaving or malicious tenants.* Today, TCP's congestion control shares bandwidth equally across flows and is agnostic to tenant requirements, and thus falls short of predictably sharing bandwidth across tenants.

EyeQ, the main contribution of this paper, is a programmable bandwidth arbitration mechanism for the datacenter network. Our design is based on the key insight that by relieving the network's core of persistent congestion, we can partition bandwidth in a simple and distributed manner, completely at the edge. EyeQ uses server-to-server congestion control mechanisms to partition bandwidth locally at senders and receivers. The design is highly scalable and responsive and ensures bandwidth guarantees are met even in the presence of highly volatile traffic patterns. The congestion control mechanism pushes overloads back to the sources, while draining traffic at maximal rates. This ensures that network bandwidth is not wasted.

The EyeQ model. EyeQ allows administrators to configure a minimum and a maximum bandwidth to a VM's Virtual Network Interface Card (vNIC). The lower bound on bandwidth permits a work-conserving allocation among vNICs collocated on a physical machine.

The EyeQ arbitration mechanism. We explain the distributed mechanism using the example shown in Figure 1. VMs of tenants A and B are given a minimum bandwidth guarantee of 2Gb/s and 8Gb/s respectively. The first network flow F1 starts at VM A1 destined for A2. In the absence of contention, it is allocated the full line rate of 10Gb/s. While F1 is in progress, a second flow F2 starts at B1 destined for B2, creating congestion at server j. The Receiver EyeQ Module (REM) at j detects this contention for bandwidth, and uses end-to-end feedback to rate limit F1 to 2Gb/s, and F2 to 8Gb/s. Now, suppose flow F3 starts at VM B1 destined for B3. The Sender EyeQ Module (SEM) at server i' partitions its link bandwidth between F2 and F3 equally. Since this lowers the rate of F2 at server j to 5Gb/s, the REM at j will allocate the spare 3Gb/s bandwidth to F1 through subsequent feedback. In this way EyeQ recursively and distributedly schedules bandwidth across a network, to simultaneously maximize utilization, and meet bandwidth guarantees.

EyeQ is practical; the SEM and REM shim layers enforce traffic admission control *without* awareness of application traffic demands, traffic patterns, or transport protocol behavior (TCP/UDP) and *without* requiring any more support from network switches than what is already available today. We demonstrate this through extensive evaluations on real applications.

In summary, our main contributions are:

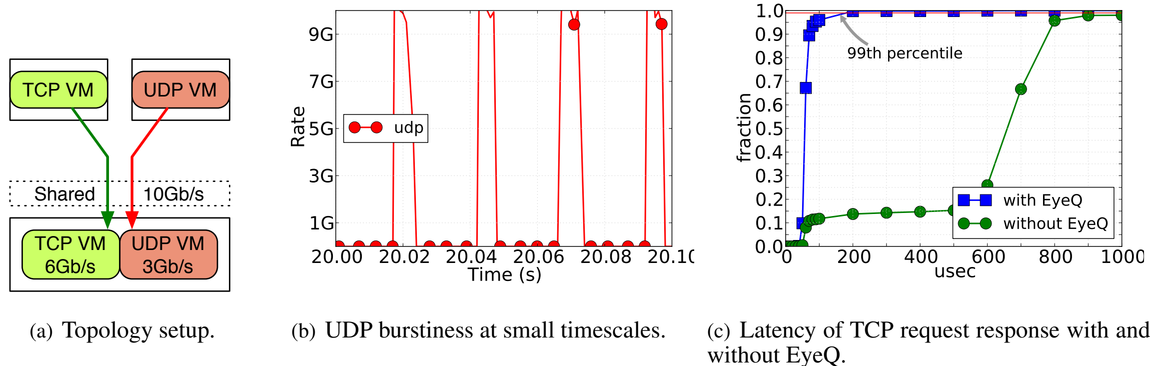
- The design of EyeQ that simultaneously achieves predictable and work-conserving bandwidth arbitration in a scalable fashion, completely from the network edge (host network stack, hypervisor, or NIC).
- An open implementation of EyeQ in software that scales to high line rates.
- An evaluation of EyeQ's feasibility at 10Gb/s on real applications.

The rest of the paper is organized as follows. We describe the nature of EyeQ's guarantees and discuss insights about network contention from a production cluster (§2) that motivate our design. We then delve into the design (§3), our software implementation (§4), and evaluation (§5) using micro- and macro-benchmarks. We summarize related work (§6) and conclude (§7).

We are committed to making our work easily available for reproducibility. Our implementation and evaluation scripts are online at `http://jvimal.github.com/eyeq`.

2 Predictable Bandwidth Partitioning

The goal of EyeQ is to schedule network traffic across a datacenter network such that it meets tenant endpoint bandwidth guarantees *over short intervals of time (e.g., a few milliseconds).* In this section, we define this notion of bandwidth guarantees more precisely and explain why bandwidth guarantees need to be met over short timescales. Then, we describe the key insight that makes EyeQ's simple design possible: The fact that the network's core in today's high bisection bandwidth datacenter networks can be kept free of persistent congestion. We show measurements from a Windows Azure production storage cluster that validate this claim.



(a) Topology setup.

(b) UDP burstiness at small timescales.

(c) Latency of TCP request response with and without EyeQ.

Figure 2: The UDP tenant bursts for 5ms and sleeps for 15ms, which even at 100ms timescales seems benign (2.5Gb/s or 25% utilization). These finer timescale interactions put a stringent performance requirement on the reaction times of any isolation mechanism, and mechanisms that react at large timescales may not see the big picture. EyeQ rate limits UDP over short timescales and improves the TCP tenant's median latency by over 10x. (There were no TCP timeouts in this experiment.)

2.1 EyeQ's bandwidth guarantees

EyeQ provides bandwidth guarantees for every endpoint (e.g., VM NIC) in order to mimic the performance of a dedicated switch for each tenant. The bandwidth guarantee for each endpoint is configured at provision time. The endpoints should be able to attain their guaranteed bandwidth as long as their traffic does not oversubscribe any endpoint's capacity.[1] For instance, if N VMs, each with 1Gb/s capacity, attempt to send traffic at full rate to a single 1Gb/s receiver, EyeQ only guarantees that the receiver (in aggregate) receives 1Gb/s of traffic. The excess traffic is dropped at the senders. Hence, EyeQ enforces traffic admissibility and only promises bandwidth guarantees for the *bottleneck port* (the receiver) and allocates bandwidth across senders in a max-min fashion.

There are different notions for bandwidth guarantees, ranging from exact rate and delay guarantees at the level of individual packets [13], to approximate "average rate" [14] guarantees over an acceptable interval of time. As observed in prior work [14], exact rate guarantees require per-endpoint queues and precise packet scheduling mechanisms [15, 16, 17] at every switch in the network. Such mechanisms are expensive to implement and are not available in switches today at a scale to isolate thousands of tenants and millions of VMs [18]. Hence, with EyeQ, we strive to attain average rate guarantees over an interval of time that is as short as possible.

2.2 Rate guarantees at short timescales

Datacenter traffic has been found to be highly volatile and bursty [5, 19, 20], leading to interactions at short timescales of a few milliseconds that adversely impact flow throughput and tail latency [21, 22, 23]. This is exacerbated by high network speeds and the use of shallow buffered commodity switches in datacenters. Today, a single large TCP flow is capable of causing congestion on its path in a matter of milliseconds, exhausting switch buffers [21, 24]. We refer the reader to [25] and our prior work [26] that demonstrate how bursty packet losses can adversely affect TCP's throughput.

The prior demonstrations highlight an artifact of TCP's behavior, but such interactions can also affect end-to-end *latency*, regardless of the transport protocol. To see this, consider a multi-tenant setting with a TCP and UDP tenant shown in Figure 2(a). Two VMs (one of each tenant) collocated on a physical machine receive traffic from their tenant VMs on other machines. Assume an administrator divides 9Gb/s of the access link bandwidth (at the receiver) between TCP and UDP tenants in the ratio 2:1 (the spare 1Gb/s or 10% bandwidth headroom is reserved to ensure good latency [22]). The UDP tenant transmits at an average rate of 2.5Gb/s by bursting in an ON-OFF fashion (ON at 10Gb/s for 5ms, OFF for 15ms). The TCP client issues back-to-back 1-byte requests over one connection and receives 1-byte responses from the server collocated with the UDP tenant.

Figure 2(c) shows the latency distribution of the TCP tenant with and without EyeQ. Though the average throughput of the UDP tenant (2.5Gb/s) is less than its allocated 3Gb/s, the TCP tenant's median and 99th percentile latency increases by over 10x. This is because of the congestion caused by the UDP tenant during the 5ms bursts at line rate, as shown in Figure 2(b). When EyeQ is enabled, it cuts off UDP's bursts at short timescales. We see that the latency with EyeQ is about 55μs, which

[1] This constraint is identical to what would occur with a dedicated switch, and is sometimes referred to as a *hose* constraint [12].

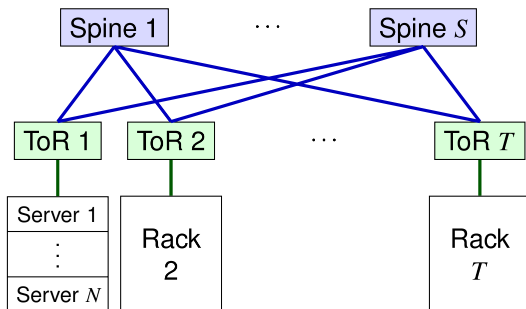


Figure 3: Emerging datacenter network architectures are over-subscribed only at the Top-of-Rack switches, which are connected to a spine layer that offers uniform bandwidth between racks. The over-subscription ratio is typically less than 3.

is close to bare-metal performance that we saw when running the TCP tenant without the UDP tenant.

Thus, any mechanism that only looks at average utilization over large timescales (e.g., over 100 milliseconds) fails to see the contention happening at finer timescales, which can substantially degrade both bandwidth and latency for contending applications. To address this challenge, EyeQ operates in the *dataplane* in a distributed fashion, and uses a responsive rate-based congestion control mechanism based on the Rate Control Protocol [27]. This enables EyeQ to quickly react to congestion in 100s of microseconds. We believe this timescale is short enough to at least protect tenants from persistent packet drops caused by other tenants as most datacenter switches have a few milliseconds worth of buffering.

2.3 The Fat and Flat Datacenter Network

In this section, we show how the high bisection bandwidth network architecture of a datacenter can simplify the task of bandwidth arbitration, which essentially boils down to managing network congestion wherever it occurs. In a flat datacenter network with little to no flow aggregation, congestion can occur everywhere, and therefore, switches need to be aware of thousands of tenants. Configuring every switch as tenants and their VMs come and go is unrealistic. We investigate where congestion actually occurs in a datacenter.

An emerging trend in datacenter network architecture is that the over-subscription ratio, typically less than 3:1, exists only at the Top-of-Rack (ToR) switches (Figure 3). Beyond the ToR switches, the network design eliminates any structural bottleneck, and offers uniform high capacity between racks in a cluster. To study where congestion occurs in such a topology, we collected link utilization statistics from Windows Azure's production storage

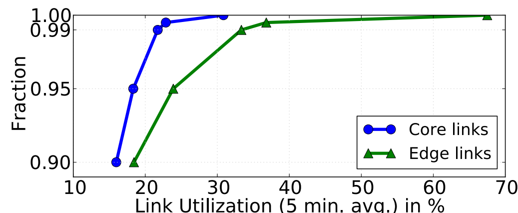


Figure 4: Utilization trends observed in a cluster running a multi-tenant storage service. The edge links exhibit higher peak and variance in link utilization compared to core links.

cluster, whose network has a small oversubscription (less than 3:1). Figure 4 plots the top 10 percentiles of link utilization (averaged over 5 minute intervals) on edge and core links using data collected from 20 racks over the course of one week. The plot reveals two trends. First, we observe that edge links have higher peak link utilization. This suggests that persistent congestion manifests itself more often, and earlier, on server ports than the network core. Second, the variation of link utilization on core links is smaller. This suggests that the network core is evenly utilized and is free of persistent hot-spots.

The reason we observe this behavior is two fold. First, we observed that TCP is the dominant protocol in our datacenters [19, 21]. The nature of TCP's congestion control ensures that traffic is *admissible*, i.e., sources do not send more traffic (in aggregate) to a sink than the bottleneck capacity along the paths. In a high capacity fabric, the only bottlenecks are at over-subscription points—the server access links and the links between the ToRs and Spines—*provided* packets are optimally routed at other places. Second, datacenters today use Equal-Cost Multi-Path (ECMP) to randomize routing at the level of flows. In practice, ECMP is "good enough" to mitigate contention within the fabric, particularly when most flows are short-lived. While the above link utilizations reflect persistent congestion over 5 minute intervals, we conducted a detailed packet-level simulation study, and found that randomized per-packet routing can push even millisecond timescale congestion to the edge (§5.3).

Thus, if (a) the network has high bisection bandwidth, (b) the network employs randomized traffic routing, and (c) traffic is admissible, persistent congestion only occurs at the access links and not in the core. This observation guides our design in that it is sufficient if per-tenant state is pushed to the edge, where it is already available.

3 EyeQ Design

We now describe EyeQ's design in light of the observations in the §2. For ease of exposition, we first abstract the datacenter network as a single switch and describe

how EyeQ ensures rate guarantees to each endpoint connected to this switch. Then in §3.5, we explain how this design fits in a network of switches. Finally, we describe how endpoints that do not need guarantees can coexist.

If a single switch connects all servers, bandwidth contention happens only at the first-hop link connecting the sender to the switch, and the last-hop link connecting the switch to the receiver. Therefore, any contention is local to the servers, where the number of competing entities is small (typically 8–32 services/VMs per server). To resolve local contention between endpoints, two features are indispensable: (a) a mechanism that detects and accounts for contention, and (b) a mechanism that enforces rate limits on flows that violate their share. We now describe mechanisms to detect and resolve contention at senders and receivers.

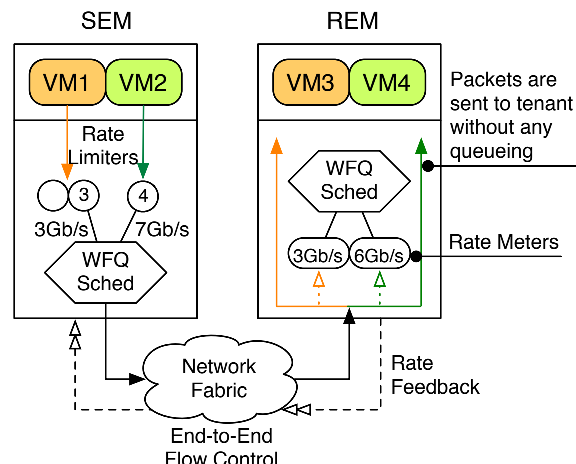


Figure 5: The design consists of a Sender EyeQ Module (SEM) and Receiver EyeQ Module (REM) at every end-host. The SEM consists of hierarchical rate limiters to enforce admission control, and a WRR scheduler to enforce minimum bandwidth guarantees. The REM consists of rate meters that detect and signal congestion. In tandem, the SEM and REM achieve end-to-end flow control.

3.1 Detecting and Resolving Contention

Senders. Contention between transmitters at the sender is straightforward to detect and resolve as the first point of contention is the server NIC. To resolve this and achive rate guarantees at the sender, EyeQ uses weighted fair queueing, where weights are set proportional to the endpoint's minimum bandwidth guarantees.

Receivers. However, contention at the receiver first happens *inside* the switch, and not at the receiving server. To see this, consider the example shown in Figure 2 where UDP generates highly bursty traffic that leads to 25% average utilization of the receiver link. When TCP begins to transmit packets, the link utilization soon approaches 100%, and packets are queued up in the limited buffer space inside the switch. If the switch does not differentially treat TCP and UDP packets, TCP's request/response experiences high latency.

Unfortunately, neither the sender nor receiver server has accurate, if any, visibility into this switch-internal contention, especially at timescales it takes to fill the switch packet buffers. These timescales can be very small as datacenter switches have limited packet buffers. Consider a scenario where two switch ports send data to a common port that has 1MB buffer. If each port starts sending at line rate, it takes just 800μs (at 10Gb/s) to fill the shared buffer inside the switch.

Fortunately, the scenario in Figure 2 offers an insight into the problem: contention happens when the link utilization, at short timescales, approaches its capacity. EyeQ therefore measures rate every 200μs and uses this information to rate limit flows before they cause further congestion. EyeQ stands to benefit if the network can further assist in quickly detecting any congestion, either using Explicit Congestion Notification (ECN) marks from a shared queue when buffers exceed a configured queue occupancy, or using per-tenant dedicated queues only on the access link. But EyeQ does not need per-tenant network queues to function.

Resolving receiver contention. Unlike sender violation, performing both detection and rate limiting at the receiver is not effective, as rate limiting at the receiver can only control well-behaved TCP-like flows (by having them back off via drops). Unfortunately, VMs may use transport protocols such as UDP, which do not react to any downstream drops. EyeQ implements receiver-side detection, sender-side reaction. Specifically EyeQ detects bandwidth violation at the receiver using per-endpoint rate meters, and enforces rate limits at the senders using per-destination rate limiters. These per-destination rate limiters are programmed by *congestion feedback* generated by the receiver (§3.4).

In summary, EyeQ's design has two main components: (a) a rate meter at receivers that sends feedback to (b) rate limiters at senders. A combination of the above is needed to address both contention at the receiver indicated using feedback, as well as local contention at the sender. The rate limiters work in a distributed fashion using a control algorithm to iteratively converge to the 'right' rates.

3.2 Receiver EyeQ Module

The Receiver EyeQ Module (REM) consists of an RX scheduler and rate meters for every endpoint. The rate

meter is just as a byte counter that periodically (200μs) tracks the endpoint's receive rate, and invokes the RX scheduler which computes a capacity for each endpoint as $C_i = B_i \times C/(\sum_{j \in A} B_j)$. Here, A is the set of active VMs that are receiving traffic at non-zero rate, and B_i is the minimum bandwidth guarantee to VM i. Rate meters can be hierarchical; for example, a tenant can create rate meters to further split its capacity C_i across tenants communicating with it.

REM is clocked by incoming packets and measures each tenant's aggregate utilization every 200μs. A packet arrival triggers a feedback to source of the current packet. The feedback is a 16-bit value R computed by the rate meter using a control algorithm. To avoid generating excessive feedback, we send feedback only to the source address of packet sampled every 10kB of received data. This tends to choose senders that are communicating at high rates over a period of time.

This sampling also restricts the maximum bandwidth consumed by feedback. Since feedback packets are minimum sized packets (64 bytes), feedback traffic will not consume more than 64Mb/s (on a 10Gb/s link). This does not depend on the number of rate meters or senders transmitting to a single machine. We call this feedback packet a host-ACK, or HACK. The HACK is a special IP packet that is never communicated to the tenant; we picked the first unused IP protocol number (143) as a 'HACK.' The HACK encodes a 16-bit rate in its IPID field. However, this feedback can also be piggybacked on traffic to the source.

3.3 Sender EyeQ Module

To enforce traffic admission control, SEM uses multiple rate limiters organized in a hierarchical fashion. To see this hierarchy, consider the scenario in Figure 5. The root WRR scheduler schedules packet transmissions so that VM1 and VM2 get (say) equal share of the transmit bandwidth. To further ensure that traffic does not congest a destination, there are a set of rate limiters at the leaf level, one per *congested* destination. Per-destination rate limiters ensure that traffic to uncongested destinations are not head-of-line blocked by traffic to congested destinations. These per-destination rate limiters set their rate dictated by HACKs from receivers (§3.4). EyeQ associates traffic to a rate limiter only when the destination signals congestion through rate feedback. If a feedback is not received within 100 milliseconds, the rate limiter halves its rate until it hits 1Mb/s, to avoid congesting the network if the receiver is unresponsive (e.g. due to failures or network partitions).

3.4 Rate Control Loop

The heart of EyeQ's architecture is a rate control algorithm that computes the rates at which senders should converge to, to avoid overwhelming the receiver's capacity limits. The goal of this algorithm is to compute one rate R_i to which all flows of a tenant destined to endpoint i should be rate limited. If N senders all send long-lived flows, then R_i is simply C_i/N. In practice, N is hard to estimate as senders are in a constant state of flux, and not all of them may want to send traffic at rate R_i. Hence, we need a mechanism that can compute R_i without estimating the number of senders, or their demands. This makes the implementation practical, and more importantly, makes it possible to offload this functionality in hardware such as programmable NICs [28].

The control algorithm (operating at each endpoint) uses the measured receive rate y_i and the endpoint's allowed receive capacity C_i (determined by the RX scheduler) to compute a rate R_i that is advertised to senders communicating only with this endpoint. The basic idea is that the algorithm starts with an initial rate estimate R_i, and periodically corrects it based on observed y_i; if the incoming rate y_i is too small, it increases R_i, and if y_i is larger than C_i, it decreases R_i. This iterative procedure to compute R_i can be written as follows, taking care to keep R_i positive:

$$R_i \leftarrow R_i \left(1 - \alpha \cdot \frac{y_i - C_i}{C_i}\right)$$

The algorithm is a variant of the Rate Control Protocol (RCP) proposed in [27, 29], but there is an important difference. RCP's control algorithm operates on links in the network to split the link capacity among every flow in a max-min fashion. This achieves *per-flow* max-min fairness, and therefore, RCP suffers from the same problems as TCP. Instead, we operate the control algorithm in a hierarchical fashion. At the top level, the physical link capacity is divided by the RX scheduler into multiple virtual link capacities (C_i), one per VM, which *isolates* VMs from one another. Next, we operate the above algorithm independently on each virtual link.

The sensitivity of the algorithm is controlled by parameter α; higher values make the algorithm more aggressive in adjusting R_i. The above equation can be analyzed as follows. In the case where N flows traverse a single congested link of unit capacity, the rate evolution of R can be described in the standard form: [2] $z[n+1] = bz[n](1 - z[n])$, where $z[n] = \left(\frac{b-1}{b}\right)\frac{R[n]}{R^*}$, $R^* = \frac{1}{N}$, and $b = 1 + \alpha$. It can be shown that R^* is the only stable fixed point of the above recurrence if $1 < b < 3$, i.e.

[2] This is a standard non-linear one-dimensional dynamical system called the Logistic Map.

$0 < \alpha < 2$. By linearizing the recurrence about its fixed point, we can show that $R[n] \approx R^* + (R[0] - R^*)(1-\alpha)^n$. Therefore the system converges linearly. In practice, we found that high values of α lead to oscillations around R^* and therefore we recommend setting α conservatively to 0.5, for which $R[n]$ converges within 0.01% of R^* in about 30 iterations, irrespective of $R[0]$.

Though EyeQ makes an assumption about congestion free network core, ECN marks from network enable EyeQ to gracefully degrade in the presence of in-network congestion that can arise, for example, when links fail. In the rare event of persistent in-network congestion, we estimate the fraction of marked incoming packets as β and reduce R_i proportionally: $R_i \leftarrow R_i(1-\beta/2)$. This term β aids in reducing the rate of transmitting endpoints *only* in such transient cases. Though minimum bandwidth guarantees cannot be met in this case, it prevents starvation where one endpoint is completely dominated by another. In this case, bottleneck bandwidth is shared equally among all receiving endpoints.

We experimented with other control algorithms based on Data Center TCP (DCTCP) [21] and Quantized Congestion Notification (QCN) [30], and found that they have different convergence and stability properties. For example, DCTCP's convergence was on the order of 100–150ms, whereas QCN converged within 20–30ms. It is important that the control loop be fast and stable, to react to bursts without over-reaching, and RCP converges within a few milliseconds to the right rate. Since EyeQ computes rates every 200μs, the worst case convergence time (30 iterations) is 6ms. In practice, it is much faster.

3.5 EyeQ on a network

So far, we described EyeQ with the assumption that all end-hosts are connected to a single switch. A few steps must be taken to ensure EyeQ's design is directly applicable to networks that have a little over-subscription at the ToR switches (Figure 3). Clearly, if the policy is to guarantee minimum bandwidth to each VM, the cluster manager must ensure that capacity is not overbooked. This becomes simpler in such networks, where the core of the network is guaranteed to be congestion free, and hence admission control must only ensure that:

- The access links at end-hosts are not over-subscribed: i.e., the sum of bandwidth guarantees of VMs on a server is less than 10Gb/s.
- The ToR's uplink capacity is not over-subscribed: i.e., the sum of bandwidth guarantees of VMs under a ToR switch is less than the switch's total capacity to the Spine layer.

The above conditions ensure that VMs are guaranteed their bandwidth in the worst case when every VM needs it. The remaining capacity can be used for VMs that have no bandwidth requirements. Traffic from VMs that do not need bandwidth guarantees are mapped to a low priority, "best-effort" network class. This requires (i) a *one-time network configuration* of a low priority queue, which is easily possible in today's commodity switches, and (ii) end-hosts to mark packets so they can be classified to low priority network queues. This partitions the available bisection bandwidth across a class of VMs that need performance guarantees, and those that do not. As we saw in §3.4, EyeQ gracefully degrades in the presence of network congestion that can happen due to oversubscription, by sharing bandwidth equally among all receiving endpoints.

4 Implementation

EyeQ needs two components: rate limiters and rate metering. These components can be implemented in software, or hardware or a combination of two for optimum performance. In this paper, we present a full-software implementation of EyeQ's mechanisms, addressing the following challenges: (a) maintaining line rate performance at 10Gb/s while reacting quickly to deal with contentions at fine timescales, (b) co-existing with today's network stacks that use various offload techniques to speed up packet processing. EyeQ uses a combination of simple and well known techniques to reduce CPU overhead. We avoid writing to data structures shared across multiple CPUs to minimize cache misses. If sharing is inevitable, we minimize updates of shared data as much as possible through batching.

In untrusted environments, EyeQ is implemented in the trusted codebase at the (hypervisor or Dom0) virtual switch. In a non-virtualized, trusted environment, EyeQ resides in the network stack as a shim layer above the device driver. As a prototype, we implemented RX and TX processing for a VMSwitch filter driver for Windows Server 2008, and a kernel module for Linux which we use for all our experiments in this paper. The kernel module implements a queueing discipline (`qdisc`) in about 1900 lines of C code and about 700 lines of header files. We implemented a simple hash table based IP based classifier to identify endpoints. EyeQ hooks into the RX datapath using `netdev_rx_handler_register`.

4.1 Receiver EyeQ Module

The REM consists of rate meters, a scheduler and a HACK generator. A rate meter is created for each VM, and tracks the VM's receive rate in an integer. Clocked

by incoming packets, the scheduler determines each endpoint's allowed rate. The scheduler distributes the receive capacity among active endpoints, in accordance with their minimum bandwidth requirements.

At 10Gb/s, today's NICs use techniques such as Receive Side Scaling [31] to scale software packet processing by load balancing interrupts across CPU cores. A single, atomically updated byte counter in the critical path is a bottleneck and limits parallelism. To avoid such inefficiencies, we exploit the fact that today's NICs use interrupt coalescing to deliver multiple packets in a single interrupt, and therefore batch counter updates over 200μs time intervals. A smaller interval results in inaccurate rate measurement due to tiny bursts, and a larger interval decreases the rate meter's ability to detect short bursts that can cause interference. In a typical shallow buffered ToR switch that has a 1MB shared buffer, it takes 800μs to fill 1MB if two ports are sending at line rate to a common receiver. Thus, the choice of 200μs interval is to balance the ability to detect short bursts, and measure rate reasonably accurately.

4.2 Sender EyeQ Module

SEM consists of multiple TX-contexts, one per endpoint, that are isolated from one another. The SEM classifies packets to their corresponding TX-context. Each context has one root rate limiter, and a hash table of rate limiters keyed by IP destination d. The hash table stores the rate control state ($R_d^{(i)}$). Recall that rate enforcement is done hierarchically; leaf rate limiters enforce per-destination rates determined by end-to-end feedback loop, and the root rate limiter enforces a per-endpoint aggregate rate determined by the TX scheduler.

Rate limiters to IP destinations are created only on a need-to-rate limit basis. At start, packets to a destination are rate limited only at the root level. A per-destination rate limiter to a destination d is created, and added to the hierarchy, only on receiving a congestion feedback from the receiver d. Inactive rate limiters are garbage collected every few seconds. The TX WRR scheduler executes every 200μs and reassigns the total TX capacity to *active* endpoints, i.e., those that have a backlog of packets waiting to be transmitted in rate limiters.

Multi-queue rate limiter. The rate limiter is implemented as a token bucket, which has an associated linked-list (tail-drop) FIFO queue, a timer, a rate R and some tokens. This simple design can be inefficient, as a single queue rate limiter increases lock contention, which degrades performance significantly, as the queue is touched for every packet. Hence, we split the ideal rate limiter's FIFO queue into a per-CPU queue, and the total tokens into a local token count (t_c) on each CPU c. The value t_c is the number of bytes that Q_c can transmit without violating the global rate limit. Only if Q_c runs out of tokens to transmit the head of the queue, it grabs the rate limiter's lock to borrow all total tokens.

If the borrow fails due to lack of total tokens, the per-CPU queue is throttled and appended to a per-CPU list of backlogged queues. We found that having a timer for every rate limiter was very expensive. Therefore, a single per-CPU timer fires every 50μs and clocks only the *backlogged* rate limiters on that CPU. Decreasing the firing interval increases the precision of the rate limiter, but increases CPU overhead as it doubles the number of interrupts per second. In practice, we found that 50μs is sufficient. At 10Gb/s, at most 64kB can be transmitted every 50μs without violating rate constraints.

The rate limiter's per-CPU FIFO maximum queue size is restricted to 128kB, beyond which it back-pressures the network stack by refusing to accept more packets. While a TCP flow responds to this immediate feedback by stopping transmission, UDP applications may continue to send packets that will be dropped. Stopped TCP flows will be resumed by incoming ACKs.

Rate limiter accuracy. Techniques such as large segmentation offload (LSO) make it challenging to enforce rates precisely. With default configuration, the TCP stack can transmit data in 64kB chunks, which takes 51.2μs to transmit at 10Gb/s. If a flow is rate limited to 1Gb/s, the rate limiter would transmit one 64kB chunk every 512μs. This burstiness affects the accuracy with which the rate meter measures rates. To limit burstiness, we restrict the maximum LSO packet size to 32kB, which enables reasonably accurate rate metering at 256μs intervals. For rates less than 1Gb/s, the rate limiter selectively disables segmentation offload by splitting large packets into smaller chunks of at most the MTU (1500 bytes). This improves rate precision without incurring much CPU overhead. Limiting the size of an LSO packet also improves latency for short flows by reducing head of line blocking at the NIC.

5 Evaluation

We evaluate EyeQ to understand the following aspects:

- **Responsiveness**: We stress EyeQ's convergence times against a large burst of UDP streams and find that EyeQ converges within 5ms to protect a collocated TCP tenant.
- **CPU overhead**: At 10Gb/s, we evaluate the main overhead of EyeQ due to its rate limiters. We find it outperforms the software rate limiters in Linux.

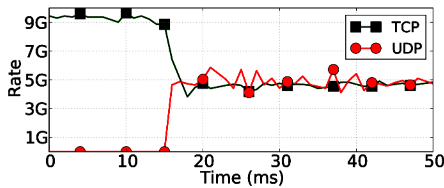


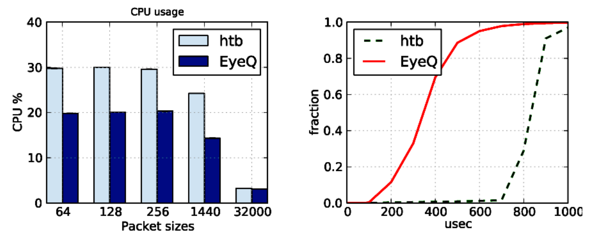
Figure 6: From rate samples taken every millisecond, we see that EyeQ converges within 5ms after UDP senders begin bursting at 14x *line rate*.

- **Flexibility**: Through EyeQ, we find that setting minimum bandwidth guarantees can directly affect the overall completion of a distributed all-to-all shuffle representative of Hadoop transfers.
- **Application performance**: In the presence of low volume, but highly bursty UDP traffic, we find that EyeQ is able to improve cluster utilization, and keep the 99.9th percentile response latency of a memcached cluster close to bare-metal performance.
- **Congestion in the fabric core**: Finally, we demonstrate using packet-level simulations that at high load, congestion even at millisecond timescales occurs more often at the edge than the network core.

Our evaluation cluster consists of 16 servers, each with a quad-core (2-way hyper-threaded) Intel Xeon 2.40Ghz processors. The servers run Linux 3.4 and are equipped with 10GbE NICs. All the servers are connected to a single shallow-buffered 24-port 10GbE switch, which has 20kB dedicated per-port buffer and 2MB shared memory. Since our switch lacks ECN support, we configure EyeQ to maintain a 10% bandwidth headroom at all times. In all our experiments, we enable Large Segmentation Offload, Receive Side Scaling (with 4 queues) and adaptive interrupt moderation. In our experiments, SEM (REM) use the source (destination) IP address to define endpoints in the transmit (receive) path, by creating subnets `11.0.T.0/24`, where T is the tenant ID.

5.1 Micro Benchmarks

Convergence times. We show a scenario that stresses the ability of EyeQ to quickly adapt and provide good isolation. There are two tenants: one TCP (1 sender and 1 receiver) and one UDP tenant: (N senders, and 1 receiver). Both receivers are collocated on the same host, but the senders are all collocated on different hosts. The TCP sender creates one long-lived TCP flow to the receiver. The UDP senders creates UDP streams, one each, simultaneously at t=30s, at maximum rate. As one would expect, without EyeQ, the TCP tenant starves while the UDP tenant runs active.



(a) CPU overhead at high load. (b) CDF of latency at low load.

Figure 7: EyeQ's rate limiter is multi-core aware, and is more efficient both in terms of CPU usage and packet latency, than today's Linux's Hierarchical Token Bucket `htb`.

When enabling EyeQ, the per-destination rate limiters are configured to start with an initial rate of 10Gb/s. This is representative of a possible worst case scenario for EyeQ, as it creates a sudden incast at rate 10N Gb/s that oversubscribes the access link by a factor of N. Each rate limiter starts at 10Gb/s and eventually has to converge to 5/N Gb/s. This initial burst eats into the headroom, and stresses the ability of EyeQ's convergence times. Figure 6 demonstrates the ability of EyeQ to enforce predictable performance, and provide a minimum bandwidth guarantee to TCP for N=14. EyeQ quickly converges within 5ms. Moreover, when the UDP tenant is not active, TCP is able to grab all 10Gb/s of bandwidth, which demonstrates the work-conserving nature of EyeQ, upto 90% of the capacity.

CPU overhead. Most of EyeQ's overhead is in rate limiting. We measure the overhead of a *single* EyeQ's rate limiter and contrast it to Linux's Hierarchical Token Bucket `htb`. Only for this test, we use a dual socket, 24 core Intel processor. To measure CPU overhead, we create a single rate limiter (at 5Gb/s), and generate traffic with varying packet sizes using 512 netperf processes. Figure 7(a) compares the CPU usage of `htb` with EyeQ, for varying packet sizes. We see that EyeQ's rate limiters have 1.5x lower CPU usage compared to `htb`. As one would expect, for a given rate, larger packet sizes implies smaller number of packets/second in software, and therefore the overhead is comparable for 32kB packets.

To measure latency overhead of `htb` due to locking, we ran 512 netperf processes, each generating back-to-back 1-byte requests and waiting for 1-byte responses. On a bare-metal system, this test generated about 10Mb/s of traffic, and thus the rate limiter configured at 5Gb/s should not have any effect. EyeQ's rate limiters work well, but with `htb`, we observe a 2.2x increase in the *median* per-transaction latency, as shown in Figure 7(b). Linux's `htb` acquires a single spinlock for each packet, and this increases the latency of enqueueing a packet into the rate limiter when the lock is heavily contended.

Recall that EyeQ requires a number of rate limiters that varies depending on the number of flows. In practice, the number of *active* flows (flows that have outstanding data) is typically less than a few 100 on a machine [5]. Nevertheless, we evaluated EyeQ's rate limiters by creating 20000 long lived flows that are assigned to a number of rate limiters in a round robin fashion. As we increased the number of rate limiters connected to the root (which is limited to 5Gb/s) from 1 to 1000 to 10000, we found that the CPU usage stays the same. This is because the net work output (packets per second) is the same in all cases, except for the (small) overhead involved in book keeping rate limiters.

5.2 Macro Benchmarks

The micro-benchmarks show that EyeQ is efficient, and responsive in mitigating congestion. In this section, we explore the benefits of EyeQ on traffic characteristics of two real world applications: a long data shuffle (e.g. Hadoop) and memcached.

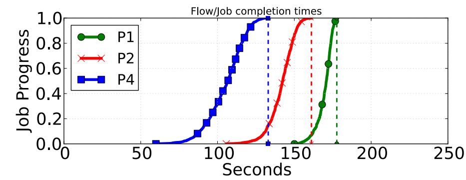
5.2.1 All-to-all data shuffle

To mimic a Hadoop job's network traffic component, we generate an all to all traffic pattern of a sort job using a traffic generator.[3] Hadoop's reduce phase is bandwidth intensive and job completion times depend on the availability of bandwidth [32]. In the sort workload of S TB of data, using a cluster of N nodes involves roughly an equal amount of data shuffle between all pairs; in effect, $\frac{S}{N(N-1)}$ TB of data is shuffled between every pair of nodes. We use a TCP traffic generator to create long lived flows according to the above traffic pattern, and record the flow completion times of all the $N(N-1)$ flows. We then plot the CDF of flow completion times for every job to visualize its progress over time; the job is complete when the last flow completes. We make no optimizations to mitigate stragglers.

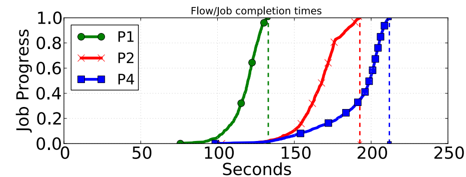
Multiple all-to-all shuffles. In this test, we run three collocated all-to-all shuffle jobs. Each job has 16 workers, one on each server in the cluster; and each server has three workers, one of each job. Each job has a varying degree of aggressiveness when consuming network bandwidth; job P_i $(i = 1,2,4)$ creates i parallel TCP connections between each pair of its nodes, and each TCP connection an transfers equal amount of data. The jobs are all temporally and spatially collocated with each other and run a common 1 TB sort workload.

Figure 8(a) shows that jobs that open more TCP connections complete faster. However, EyeQ provides flexibility to explicitly configure job priorities, irrespective of the traffic, or protocol behavior. Figure 8(b) shows the job completion times if the lesser aggressive jobs are given higher priority; the priority can be inverted, and the job completion times reflect the change of priorities. The job priority is inverted by assigning minimum bandwidth guarantees B_i to jobs P_i that is inversely proportional to their aggressiveness; i.e., $B_1 : B_2 : B_4 = 4 : 2 : 1$. The final completion time in EyeQ increases from 180s to 210s, due to two reasons. First, EyeQ's congestion detectors maintain a 10% bandwidth headroom inorder to work at the end hosts without network ECN support. Second, the REM (§3.2) does not share bandwidth in a fine-grained, per-packet fashion, but over a 200μs time window. This leads to a small loss of utilization, when (say) P_1 is allocated some bandwidth but does not use it.

[3] We used a traffic generator as our disks could not sustain enough write throughput to saturate a 10Gb/s network.



(a) *Without EyeQ*, job P4 creates 4 parallel TCP connections between its workers and completes faster.



(b) *With different minimum bandwidth guarantees*, EyeQ can directly affect the job completion times.

Figure 8: EyeQ's ability to differentially allocate bandwidth to all-to-all shuffle jobs can affect their completion times, and can be done at runtime, without reconfiguring jobs.

5.2.2 Memcached

Our final macro-evaluation is a scenario where a memcached tenant is collocated alongside an adversarial UDP tenant. The memcached tenant is a cluster consists of 16 processes: 4 memcached instances and 12 clients. Each process is located on a different host. At the start of the experiment, each cache instance allocates 8GB of memory, each client starts one thread per cache instance, and each thread opens 10 permanent TCP connections to its designated cache instance.

Throughput test. We generate an external load of about 288k requests/sec load balanced equally across all clients; at each client, the mean load is 6000 requests/sec to each cache instance. The clients generate SET re-

	Latency percentiles (μs)		
Scenario	**50th**	**99th**	**99.9th**
Bare	98	370	666
Bare+EyeQ	100	333	630
Bare+UDP	4127	0.89×10^6	1.1×10^6
Bare+UDP+EyeQ	102	437	750

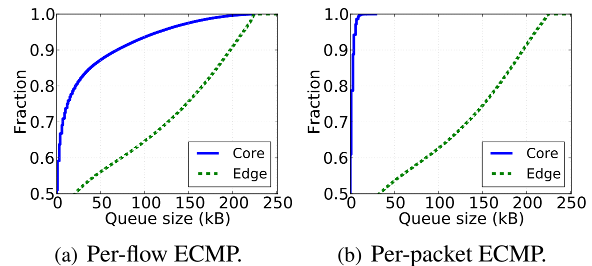
Table 1: Latency of memcached SET requests at low load (144k req/s). In all cases, the cluster throughput was the same, but EyeQ protects memcached from bursty traffic, bringing the 99.9th percentile latency closer to bare-metal performance.

quests of 6kB values and 32B keys and record the latency of each operation. We contrast the performance under four cases. First, we dedicate the cluster to the memcached tenant and establish baseline performance. The cluster was able to sustain the external load of 288k requests/sec. Second, we enable EyeQ on the same setup and found that EyeQ does not affect total throughput.

Third, we collocate memcached with a UDP tenant, by instantiating a UDP node on every end host. Each UDP node sends half-a-second burst of data to one other node (chosen in a round robin fashion), sleeping for half-a-second between bursts. Thus, the average utilization of UDP tenant is 5Gb/s. We chose this pattern as some cloud providers today allow a VM to burst at high rates for a few seconds before it is throttled. In this case, we find that the cluster was able to keep up only with 269k requests/sec which caused many responses *timed out* even though UDP tenant is consuming only 5Gb/s. Finally, we set equal bandwidth guarantees (5Gb/s) to both UDP and memcached tenant. We find that the cluster is can sustain the demand of 288k requests/sec. This shows that EyeQ is able to protect memcached tenant from the bursty UDP traffic.

Latency test. We over-provisioned the cluster by halving the external load (144k requests/sec). When memcached is collocated with UDP without EyeQ's protection, we observed that the cluster was able to meet its demand, but UDP was still able to affect the latency of memcached requests, increasing the 99th percentile latency by over three orders of magnitude. When enabled, EyeQ was able to protect the memcached tenant from fine-grained traffic bursts, and bring the 99.9th percentile latency to 750μs. The latency is still more than bare metal as the total load on the network is higher.

Takeaways. This experiment highlights a subtle point. Though we pay a 10% bandwidth price for low latency, EyeQ *improves* the net cluster utilization and tail latency performance. In a real setup, an unsuspecting client would pay money to spin up additional memcached instances to cope with the additional load. While this does increase revenue for providers, we believe they can earn more by using fewer resources to achieve the same level of performance. In datacenters, servers account for over 60% of the cost, but network accounts for only 10–15% [33]. With EyeQ, cloud operators can hope to achieve better CPU packing without worrying about network interference.



(a) Per-flow ECMP. (b) Per-packet ECMP.

Figure 9: Queue occupancy distribution at the edge and core links, for two different routing algorithms. The maximum queue size is 225kB (150 packets). Large queue sizes indicate more congestion. In all cases, the network edge is more congested even at small timescales (1ms).

5.3 Congestion in the Fabric

In §2.3 we used link utilization from a production cluster as evidence that network congestion happens more often at the edge than the network core. These coarse-grained average link utilizations over five minute intervals show macroscopic trends, but it does not capture congestion at packet timescales. Using packet-level simulations in ns2 [34], we study the extent to which transient congestion can be mitigated at the network core using two routing algorithms: (a) per-flow ECMP and (b) a per-packet variant of ECMP. In the per-packet variant, each *packet*'s route is chosen uniformly at random among all next hops [35]. To cope with packet reordering, we increase TCP's duplicate ACK threshold.

We created a full bisection bandwidth topology of 144x10GbE hosts, 9 ToRs and 16 Spines as in our datacenters (Figure 3). Servers open TCP connections to every other server and generate traffic at an average rate of 10Gb/s$\times\lambda$, where λ is the offered load. Each server picks a random TCP connection to transmit data. Flow sizes are drawn from a distribution observed in a large datacenter [21]; the median, mean and 90th percentile flow sizes are 19kB, 2.4MB, 133kB respectively. We set queue sizes of all network queues to 150 packets and collect queue samples every 1ms. Figure 9 shows queue occupancy percentiles at the edge and core when $\lambda = 0.9$.

Even at high load, we observe that the core links are far less congested than the edge links. With per-packet ECMP, the *maximum* queue occupancy is less than 50kB

in the core links. All packet are dropped at the server access links. In all cases, we observed that per-packet ECMP practically eliminates in-network congestion *irrespective of traffic pattern*.

6 Related work

EyeQ's goals fall under the umbrella of Network Quality of Service (QoS), which has a rich history. Early QoS models [13, 36] and their implementations [15, 16, 36] focus on link-level and network-wide rate and delay guarantees for flows between a source and destination. Protocols such as Resource Reservation Protocol (RSVP) [37] reserve/relinquish resources across multiple links using such QoS primitives. Managing network state for every flow becomes untenable when there are a lot of flows. This led to approaches that relax strict guarantees for "average bandwidth," [14, 38, 39] while incurring lower state management overhead. A notable candidate is Core-Stateless Fair Queueing (CSFQ) that distributes state between the network core and the network edge, relying on flow aggregation at edge routers. In a flat datacenter network, tenant VMs are distributed across racks for availability, and hence there is little to no flow aggregation. OverQoS [40] provided an abstraction of a controlled loss virtual link with statistical bandwidth guarantees, between two nodes on an overlay network; this "pipe" model requires customers to specify bandwidth requirements between all communicating pairs, in contrast to a hose model [12].

Among recent approaches, Seawall [18] shares bottleneck capacity across competing source VMs (relative to their weights). This notion of sharing lacks predictability, as a tenant can grab more bandwidth by launching more source VMs. Oktopus [3] argues for predictability by enforcing a static hose model using rate limiters. It computes rates using a pseudo-centralized mechanism, where VMs communicate their pairwise bandwidth consumption to a tenant-specific centralized coordinator. This control plane overhead limits reaction times to about 2 seconds. However, as we have seen (§2.2), any isolation mechanism has to react quickly to be effective. SecondNet [41] is limited to providing static bandwidth reservations between pairs of VMs. In contrast to Oktopus and SecondNet, EyeQ supports both static and work conserving bandwidth allocations.

The closest related work to EyeQ is Gatekeeper [42], which also argues for predictable bandwidth allocation, and uses congestion control to provide rate guarantees to VMs. While the high level architecture is similar, Gatekeeper lacks details on the system design, especially the rate control mechanism, which is critical to providing bandwidth guarantees at short timescales. Gatekeeper's evaluation is limited to static scenarios with long lived flows. Moreover, Gatekeeper uses Linux's hierarchical token bucket, which incurs high overhead at 10Gb/s.

FairCloud [43] explored fundamental trade-offs between network utilization, min-guarantees and payment proportionality, for a number of sharing policies. FairCloud demonstrated the effect of such policies with per-flow queues in switches and CSFQ, which have limited or no support in today's commodity switches. However, the minimum-bandwidth guarantee that EyeQ supports conforms to FairCloud's 'Proportional-sharing on proximate links (PS-P)' sharing policy, which, as the authors demonstrate, outperform many other sharing policies. NetShare [44] used in-network weighted fair queueing to enforce bandwidth sharing among VMs. This approach, unfortunately, does not scale well due to the limited queues (8–64) per port.

The literature on congestion control mechanisms is vast; however, the fundamental unit of allocation is still per-flow, and therefore, the mechanisms are not adequate for network performance isolation. We refer the interested reader to [45] for a more comprehensive survey about recent efforts to address performance unpredictability in datacenter networks.

7 Concluding Remarks

In this paper, we presented EyeQ, a platform to enforce predictable network bandwidth sharing within the datacenter, using minimum bandwidth guarantees to endpoints. Our design and evaluation shows that a synthesis of well known techniques can lead to a simple and scalable design for network performance isolation. EyeQ is practical, and is deployable on today's, and next generation high speed datacenter networks with no changes to network hardware or applications. With EyeQ, providers can flexibly and efficiently apportion network bandwidth across tenants by giving each tenant endpoint a predictable minimum bandwidth guarantee, eliminating the problem of accidental, or malicious traffic interference.

Acknowledgments

We would like to thank the anonymous reviewers, Ali Ghodsi and our shepherd Lakshminarayanan Subramanian for their feedback and suggestions. The work at Stanford was funded by NSF FIA award CNS–1040190, by a gift from Google, and by DARPA CRASH award #N66001-10-2-4088. Opinions, findings, and conclusions do not necessarily reflect the views of the NSF or other sponsors.

References

[1] Amazon Virtual Private Cloud. `http://aws.amazon.com/vpc/`.

[2] Jayaram Mudigonda, Praveen Yalagandula, Jeff Mogul, Bryan Stiekes, and Yanick Pouffary. Netlord: a scalable multi-tenant network architecture for virtualized datacenters. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 62–73. ACM, 2011.

[3] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. Towards predictable datacenter networks. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 242–253. ACM, 2011.

[4] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[5] Albert Greenberg, James R Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: a scalable and flexible data center network. In *ACM SIGCOMM Computer Communication Review*, volume 39, pages 51–62. ACM, 2009.

[6] Radhika Niranjan Mysore, Andreas Pamboris, Nathan Farrington, Nelson Huang, Pardis Miri, Sivasankar Radhakrishnan, Vikram Subramanya, and Amin Vahdat. Portland: a scalable fault-tolerant layer 2 data center network fabric. In *ACM SIGCOMM Computer Communication Review*, volume 39, pages 39–50. ACM, 2009.

[7] Chuanxiong Guo, Haitao Wu, Kun Tan, Lei Shi, Yongguang Zhang, and Songwu Lu. Dcell: a scalable and fault-tolerant network structure for data centers. In *ACM SIGCOMM Computer Communication Review*, volume 38, pages 75–86. ACM, 2008.

[8] C.E. Hopps. Analysis of an equal-cost multi-path algorithm. 2000.

[9] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, pages 19–19, 2010.

[10] Theophilus Benson, Aditya Akella, Anees Shaikh, and Sambit Sahu. Cloudnaas: a cloud networking platform for enterprise applications. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 8. ACM, 2011.

[11] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. Improving datacenter performance and robustness with multipath tcp. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 266–277. ACM, 2011.

[12] Nicholas G Duffield, Pawan Goyal, Albert Greenberg, Partho Mishra, Kadangode K Ramakrishnan, Jacobus E Van der Merwe, et al. A flexible model for resource management in virtual private networks. *ACM SIGCOMM Computer Communication Review*, 29(4):95–108, 1999.

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

[14] Rong Pan, Lee Breslau, Balaji Prabhakar, and Scott Shenker. Approximate fairness through differential dropping. In *ACM SIGCOMM Computer Communication Review*, volume 33, pages 23–39. ACM, 2003.

[15] Alan Demers, Srinivasan Keshav, and Scott Shenker. Analysis and simulation of a fair queueing algorithm. In *ACM SIGCOMM Computer Communication Review*, volume 19, pages 1–12. ACM, 1989.

[16] Madhavapeddi Shreedhar and George Varghese. Efficient fair queueing using deficit round robin. In *ACM SIGCOMM Computer Communication Review*, volume 25, pages 231–242. ACM, 1995.

[17] Jon CR Bennett and Hui Zhang. Wf2q: worst-case fair weighted fair queueing. In *INFOCOM'96. Fifteenth Annual Joint Conference of the IEEE Computer Societies. Networking the Next Generation. Proceedings IEEE*, volume 1, pages 120–128. IEEE, 1996.

[18] Alan Shieh, Srikanth Kandula, Albert Greenberg, Changhoon Kim, and Bikas Saha. Sharing the data center network. In *USENIX NSDI*, volume 11, 2011.

[19] Srikanth Kandula, Sudipta Sengupta, Albert Greenberg, Parveen Patel, and Ronnie Chaiken. The nature of data center traffic: measurements & analysis. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, pages 202–208. ACM, 2009.

[20] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Understanding data center traffic characteristics. *ACM SIGCOMM Computer Communication Review*, 40(1):92–99, 2010.

[21] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). *ACM SIGCOMM Computer Communication Review*, 40(4):63–74, 2010.

[22] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less is more: Trading a little bandwidth for ultra-low latency in the data center. In *Proceedings of USENIX NSDI conference*, 2012.

[23] David Zats, Tathagata Das, Prashanth Mohan, Dhruba Borthakur, and Randy Katz. Detail: Reducing the flow completion time tail in datacenter networks. *ACM SIGCOMM Computer Communication Review*, 42(4):139–150, 2012.

[24] Mohammad Alizadeh, Berk Atikoglu, Abdul Kabbani, Ashvin Lakshmikantha, Rong Pan, Balaji Prabhakar, and Mick Seaman. Data center transport mechanisms: Congestion control theory and ieee standardization. In *Communication, Control, and Computing, 2008 46th Annual Allerton Conference on*, pages 1270–1277. IEEE, 2008.

[25] Aleksandar Kuzmanovic and Edward W Knightly. Low-rate tcp-targeted denial of service attacks: the shrew vs. the mice and elephants. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 75–86. ACM, 2003.

[26] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazieres, Balaji Prabhakar, Changhoon Kim, and Windows Azure. Eyeq: practical network performance isolation for the multi-tenant cloud. In *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*, pages 8–8. USENIX Association, 2012.

[27] Nandita Dukkipati and Nick McKeown. Why flow-completion time is the right metric for congestion control. *ACM SIGCOMM Computer Communication Review*, 36(1):59–62, 2006.

[28] Guohan Lu, Chuanxiong Guo, Yulong Li, Zhiqiang Zhou, Tong Yuan, Haitao Wu, Yongqiang Xiong, Rui Gao, and Yongguang Zhang. Serverswitch: A programmable and high performance platform for data center networks. In *Proc. NSDI*, 2011.

[29] Frank Kelly, Gaurav Raina, and Thomas Voice. Stability and fairness of explicit congestion control with small buffers. *ACM SIGCOMM Computer Communication Review*, 38(3):51–62, 2008.

[30] Mohammad Alizadeh, Abdul Kabbani, Berk Atikoglu, and Balaji Prabhakar. Stability analysis of qcn: the averaging principle. *ACM SIGMETRICS Performance Evaluation Review*, 39(1):49–60, 2011.

[31] Guide to linux kernel network scaling. `http://code.google.com/p/kernel/wiki/NetScalingGuide`.

[32] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the outliers in map-reduce clusters using mantri. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, pages 1–16. USENIX Association, 2010.

[33] Internet-scale datacenter economics: Costs and opportunities. `http://mvdirona.com/jrh/TalksAndPapers/JamesHamilton_HPTS2011.pdf`.

[34] Steven McCanne, Sally Floyd, Kevin Fall, Kannan Varadhan, et al. Network simulator ns-2, 1997.

[35] Advait Dixit, Pawan Prakash, and Ramana Rao Kompella. On the efficacy of fine-grained traffic splitting protocols in data center networks. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 430–431. ACM, 2011.

[36] Ion Stoica, Hui Zhang, and TS Ng. *A hierarchical fair service curve algorithm for link-sharing, real-time and priority services*, volume 27. ACM, 1997.

[37] Lixia Zhang, Steve Deering, Deborah Estrin, Scott Shenker, and Daniel Zappala. Rsvp: A new resource reservation protocol. *Network, IEEE*, 7(5):8–18, 1993.

[38] Ion Stoica, Scott Shenker, and Hui Zhang. Core-stateless fair queueing: Achieving approximately fair bandwidth allocations in high speed networks. In *ACM SIGCOMM Computer Communication Review*, volume 28, pages 118–130. ACM, 1998.

[39] Abdul Kabbani, Mohammad Alizadeh, Masato Yasuda, Rong Pan, and Balaji Prabhakar. Af-qcn: Approximate fairness with quantized congestion notification for multi-tenanted data centers. In *High Performance Interconnects (HOTI), 2010 IEEE 18th Annual Symposium on*, pages 58–65. IEEE, 2010.

[40] Lakshminarayanan Subramanian, Ion Stoica, Hari Balakrishnan, and Randy Katz. Overqos: An overlay based architecture for enhancing internet qos. In *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, volume 1, pages 6–21, 2004.

[41] Chuanxiong Guo, Guohan Lu, Helen J Wang, Shuang Yang, Chao Kong, Peng Sun, Wenfei Wu, and Yongguang Zhang. Secondnet: a data center network virtualization architecture with bandwidth guarantees. In *Proceedings of the 6th International COnference*, page 15. ACM, 2010.

[42] Henrique Rodrigues, Jose Renato Santos, Yoshio Turner, Paolo Soares, and Dorgival Guedes. Gatekeeper: Supporting bandwidth guarantees for multi-tenant datacenter networks. *USENIX WIOV*, 2011.

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

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

[45] Jeffrey C Mogul and Lucian Popa. What we talk about when we talk about cloud network performance. *ACM SIGCOMM Computer Communication Review*, 42(5):44–48, 2012.