Aequitas: Admission Control for Performance-Critical RPCs in Datacenters

Yiwen Zhang[†‡], Gautam Kumar[†], Nandita Dukkipati[†], Xian Wu[†], Priyaranjan Jha[†], Mosharaf Chowdhury[‡], Amin Vahdat[†]

[†]Google LLC., [‡]University of Michigan

ABSTRACT

With the increasing popularity of disaggregated storage and microservice architectures, high fan-out and fan-in Remote Procedure Calls (RPCs) now generate most of the traffic in modern datacenters. While the network plays a crucial role in RPC performance, traditional traffic classification categories cannot sufficiently capture their importance due to wide variations in RPC characteristics. As a result, meeting service-level objectives (SLOs), especially for performance-critical (*PC*) RPCs, remains challenging.

We present Aequitas, a distributed sender-driven admission control scheme that uses commodity Weighted-Fair Queuing (WFQ) to guarantee RPC-level SLOs. In the presence of network overloads, it enforces cluster-wide RPC latency SLOs by limiting the amount of traffic admitted into any given QoS and downgrading the rest. We show analytically and empirically that this simple scheme works well. When the network demand spikes beyond provisioned capacity, Aequitas achieves a latency SLO that is 3.8× lower than the state-of-art congestion control at the 99.9^{th}-p and admits up to 2× more *PC* RPCs meeting SLO when compared with pFabric, Qjump, D^3, PDQ, and Homa. Results in our fleetwide production deployment show a 10% latency improvement.

CCS CONCEPTS

• **Networks** → **Network architectures**; **Network management**; **Data center networks**;

KEYWORDS

Quality of Service, RPC Performance, Network Overload

ACM Reference Format:
Yiwen Zhang, Gautam Kumar, Nandita Dukkipati, Xian Wu, Priyaranjan Jha, Mosharaf Chowdhury, Amin Vahdat. 2022. Aequitas: Admission Control for Performance-Critical RPCs in Datacenters. In *ACM SIGCOMM 2022 Conference (SIGCOMM '22), August 22–26, 2022, Amsterdam, Netherlands.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3544216.3544271

SIGCOMM '22, August 22–26, 2022, Amsterdam, Netherlands

ACM ISBN 978-1-4503-9420-8/22/08.
https://doi.org/10.1145/3544216.3544271

1 INTRODUCTION

Modern datacenter applications are composed of many microservices [16, 38, 44, 52] that interact with each other and remote disaggregated storage [4, 58, 59, 66, 68] using Remote Procedure Calls (RPCs). As of 2021, RPCs generate 95%+ of the application traffic in Google production datacenters, of which ~75% is to and from storage systems. To satisfy diverse business requirements of modern applications, RPCs generated by these microservices often have Service Level Objectives (SLOs) that vary widely. While many performance-critical (*PC*) RPCs have microsecond-scale SLOs (e.g., interactive user-facing traffic), some can take much longer (e.g., maps traffic for ride-sharing applications). Non-critical (*NC*) RPCs often constitute bulk storage operations, and there are best-effort (*BE*) RPCs from background analytics and machine learning.

Because of many recent advances in host and cluster networking, most RPCs complete within their SLOs. However, supporting predictable RPC performance under overload scenarios still remains elusive. Datacenter networks are deliberately over-subscribed for statistical multiplexing as it would otherwise be too expensive to provision. Consequently, network overloads—sometimes as high as 8× the average—are inevitable when multiple applications simultaneously surge in their demands. Under sustained network overload, the network component of RPC latency often dominates to the point of making the service effectively unavailable.

Recent research on meeting low-latency application needs in datacenter networks falls into three broad categories. First, many congestion control (CC) schemes [2, 34, 36, 39, 69] perform well in maximally using link capacity, keeping losses and network queues low in times of overload. Yet, CC by itself cannot provide *guarantees* for RPC latency: under overload, CC fair-shares the network bandwidth and causes a slowdown for *all* RPCs. Second, priority-based schemes [3, 40] minimize the average flow completion time by prioritizing smaller flows based on size or strictly apply application-defined priorities. The former does not work well when size and priority are not aligned, while the latter incentivizes applications to mark all their RPC as the highest priority. We observe both trends in production, and both lead to missed SLOs. On top of that, priority-based schemes are not readily deployable in many existing datacenters. Finally, another line of work focuses on providing bandwidth sharing guarantees

[6, 7, 13, 27, 41, 49, 50], but these efforts do not consider application priorities or provide RPC latency guarantees, make restrictive assumptions on where overloads occur [32, 53] or involve centralized entities that are hard to scale in large datacenters [48].

Our goal in this paper is to provide SLOs for *PC* RPCs—regardless of their size—in the network, even at the 99.9th percentile (99.9^{th}-p). We do so by focusing on the network component of RPC latency, which we call *RPC Network-Latency* (RNL). This leads to a design where RPCs become first-class citizens, and hosts make dynamic and local QoS-admission decisions to meet SLOs, leveraging commodity network components with weighted fair queuing (WFQ) QoS capabilities.

We present Aequitas, a simple admission control system anchored in two key conceptual insights. First, WFQ in switches has delay bounds that can be used to provide RNL SLOs in overload situations. Building on network calculus concepts [17], we derive through theory and simulations the *admissible region* based on per-QoS worst-case latency with respect to QoS utilization. Second, by explicitly managing the traffic admitted on a per-QoS basis, we can guarantee a cluster-wide per-QoS RNL SLO for all but the lowest QoS even under traffic overloads. Based on these insights, the design of Aequitas can be summarized as follows:

(1) End-hosts *align* 1:1 the priority classes at the granularity of *RPCs* (*PC*, *NC* and *BE*) to high/medium/low-weight network QoS queues (QoS_h, QoS_m, QoS_l) by encoding the QoS in the packet's DSCP header field. Switches are simple and enforce the standard QoS using WFQ.

(2) Sending hosts employ a distributed *admission control* scheme to manage the traffic mix across QoS levels. Hosts independently measure RNL for each QoS level. When the offered load of QoS_h or QoS_m RPCs is high, hosts adaptively *downgrade* excess traffic to QoS_l such that admitted traffic in higher QoS classes meets SLOs, with no explicit coordination.

We evaluate Aequitas with packet-level simulations and testbed experiments using real application workloads, and we also present early results from production deployment. We find that:

(1) Predictable latency performance can be realized cluster-wide through picking RPC winners and losers explicitly. By measuring RNL for each QoS level and realizing explicitly when the offered load is no longer in profile, hosts can make *local* decisions to admit or downgrade QoS for an RPC—a simple and effective way to ensure that quality network experience is always available for admitted traffic.

(2) With Aequitas, *PC* traffic is SLO-compliant not just at the mean, but also at the 99.9^{th}-p RNL, even when network demand spikes 10× beyond provisioned capacity. In production, Aequitas achieves 10% average reduction in 99^{th}-p RNL across fifty clusters.

(3) There exists a trade-off between how strict the SLO is and amount of traffic which can be admitted at that SLO. Aequitas achieves close to maximal traffic that can be admitted within SLO-compliance.

(4) Judicious management of traffic mix across QoS levels can create lower latency for *all* classes of traffic, including the *BE* class.

This work does not raise any ethical issues.

2 BACKGROUND AND MOTIVATION

We begin with background on RPC workloads in modern datacenters, followed by how network impacts RPCs and what challenges exist in mitigating the network impact.

2.1 RPC Workloads in Modern Datacenters

Modern datacenter applications that use microservice architectures [16, 38, 44, 52] or interact with disaggregated memory and storage systems [4, 58, 59, 66, 68] rely heavily on RPCs. An RPC is a programmatic request for action or information between components of applications, and it can consist of dozens of individual packets. Hundreds of RPCs can be on the critical path to completing an application-level operation. As a result, many datacenter application developers today measure and reason about application performance in terms of RPC performance [56]. Indeed, RPCs generate 95%+ of the application traffic in our production datacenters, of which ~75% is to and from storage devices.

Business Priorities: Typical cluster applications in public clouds—Storage, MapReduce, distributed in-memory file system, web search indexing, query serving, and caching services being the largest few in our production datacenters—classify their traffic into three priority classes:

(1) *Performance-critical (PC)* RPCs have tail latency SLOs. Sometimes they are associated with real-time interactive applications or carry key control traffic.

(2) *Non-critical (NC)* RPCs generally care about sustained rate and their latency SLOs are less stringent on the tail relative to *PC* RPCs.

(3) *Best-effort (BE)* RPCs have the lowest priority, such as background backup traffic which sees no imminent disadvantage to elevated latency as long as it eventually completes. *BE* RPCs have no SLOs and are akin to a scavenger class.

RPC priority classes are used in application-level logic as well as for prioritization under server/client overloads. For storage, *PC* RPCs might constitute small random access reads and metadata exchange; *NC* RPCs might include large sequential reads, and *BE* might be for backups that are most concerned with long-term average throughput. For an online retail tenant running on public cloud, revenue-generating user traffic is *PC*; a ride-sharing tenant may consider real-time maps traffic to be *PC*; and a social networking tenant would classify its user-facing traffic to be *PC*. Machine learning training or analytics workloads may be *BE*. A key goal

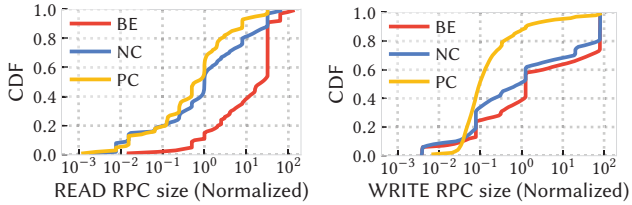


Figure 1: Normalized RPC size distribution of READs and WRITEs.

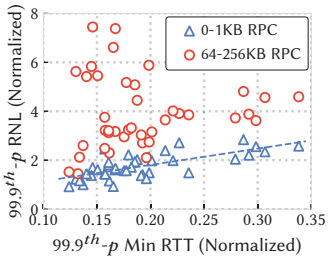


Figure 2: RNL of [0–1KB] and [64–256KB] RPCs versus Min RTT. Each data point is a sampled cluster.

in assigning priorities is to avoid lower priority RPCs from interfering with those with higher priority.

Size Distributions: We find that RPC size is not fully correlated with priority from applications' perspectives. Figure 1 shows the CDF of storage RPC sizes using response payload size for READ RPCs and request payload size for WRITE RPCs collected in our datacenter for three categories — *PC*, *NC*, *BE* — as per their application-level priorities. While it is true that the *PC* RPCs are generally smaller than *NC* or *BE* RPCs, there are high-priority large *PC* RPCs. As such, size-based network prioritization schemes proposed in the existing literature [3, 28, 40] can lead to poor performance induced by priority inversion.

2.2 Network Impact on RPCs

2.2.1 RPC Network Latency (RNL)

While end-to-end RPC latency serves as the primary metric for assessing regressions and triggering production alerts, it consists of two primary components: client/server latency (CPU load, thread scheduling, cache state) and network latency. In this work, we focus on meeting SLOs for the component of RPC latency impacted by network overload, which we refer to as *RPC network-latency* or RNL in this work.

Specifically, we define RNL as the time between the first RPC packet arriving at the transport layer (such as TCP) and the time when the last packet of the RPC is acknowledged at the transport. Appendix A provides a detailed breakdown. RNL captures delays incurred in the host networking stack due to network overload, including queuing delays incurred due to congestion control (CC) backoff.

RNL depends on (i) the bandwidth available to an RPC, (ii) queuing and propagation delays which comprise packet-level RTT, and (iii) congestion control decisions such as congestion window and pacing rate. Figure 2 shows that RTT and

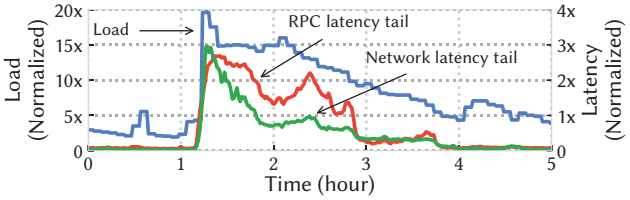


Figure 3: A congestion episode (w/o Aequitas) in production showing that increased load (bits/sec) leads to RPC latency spikes. Higher RNL than RPC latency in some cases is due to sampling differences.

bandwidth do not succinctly capture RNL in a randomly sampled collection of our clusters. While RNL of small (<1 MTU) RPCs correlates well with RTTs, RNL of large RPCs do not demonstrate the correlation. We can see clusters with low RNL for small RPCs (and low RTTs) but showing high RNL for large RPCs. The main reason is that an effective congestion control algorithm will keep packet delays low no matter how large the offered load is, but at high offered loads, RPCs may be queued for long periods at the sending hosts. Thus, providing either packet-level SLOs or bandwidth-SLOs in isolation are not meaningful to application developers — they care about RPC latency SLOs.

2.2.2 Impact of Network Overloads on RNL

Sustained network overloads without admission control lead to queuing and packet drops, which can significantly affect RPC performance as a whole as well as RNL. Figure 3 shows an example of degraded RPC latency when network load surged to 8× the usual load at the ToR uplinks in a production incident. RNL is a major contributing factor to the elevated latency. For many services, such extreme degradation in latency can be tantamount to an unavailable service.

Although several proposals for network isolation [32, 53] assume that oversubscriptions occur only at the edge ToR-to-NIC links, it is not always the case. Because typical workloads use relatively little bandwidth compared to their peaks, it is cost-ineffective to provision peak capacity for *all* workloads across *all* cuts of the network. As a result, overloads can occur *anywhere* in the network along the path that an RPC takes between the client and the server. Kumar et al. [34] made a similar observation.

2.3 Challenges in Mitigating the Network Impact

There are three primary approaches toward enforcing performance isolation between *PC*, *NC*, and *BE* RPC traffic in the network.

(1) *Size-based approaches* — such as Shortest Job First (SJF) and Shortest Remaining Time First (SRTF) — do not work well because size and importance are often misaligned.

(2) *Strict priority queuing (SPQ)*, where RPC priorities are pushed down into the network and used as network priorities, provides a perverse incentive where developers mark all of their RPCs as *PC* to receive good network performance.

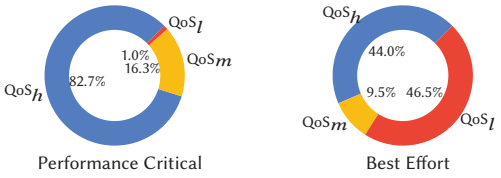


Figure 4: Production data showing high misalignment between RPC priority (left) and network QoS (right).

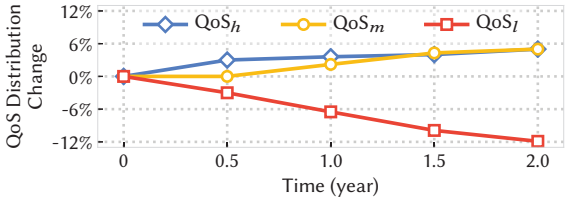


Figure 5: Distribution change of QoS classes over time.

More importantly, strict priority queuing is known to cause starvation if there are traffic surges with improperly configured high-priority traffic, and thus is not widely used in many production networks.

(3) *Weighted fair queuing (WFQ)* is available in commodity switches/NICs and its bandwidth sharing property enables the mapping from traffic priority classes to desired QoS levels.[1] We currently take this approach to map RPC priorities to network QoS levels[2] in our datacenters.

Mapping RPC priorities to WFQ QoS levels, however, comes with two primary constraints. First, such mappings are *coarse*; a common model used in practice by many cloud providers is to map all traffic from an entire application to the same class [4, 6, 31, 55, 58]. For example, if a business-critical application is marked as *PC*, all its traffic (including *NC* and *BE* traffic) is marked as critical. Second, QoS classes are *not associated with guarantees* or SLOs.

The common practice of allowing developers to set coarse-grained application-level priorities leads to a surprising degree of mismatch between actual RPC priority and their supposed importance in the network. In surveying our production traffic before the deployment of Aequitas, shown in Figure 4, we found that 17.3% of *PC* RPCs did not flow on the highest class, while 54.5% of *BE* RPCs used a higher level than necessary. Pervasive priority misalignment can degrade *PC* RPC latency due to *BE* overload even when there is sufficient capacity to meet SLO for *PC* RPCs.

A direct consequence of the above scenario is a ***race to the top***: each time a network overload event occurs, applications that suffer an RNL SLO failure event are often granted a higher priority class. Figure 5 shows how more application traffic moved to higher classes over time for the services in our datacenters before we deployed Aequitas. Two underlying problems contribute to this problem: (1) the granularity of mapping at the application level creates stronger QoS upgrade pressure than necessary; and (2) the absence of a scheme that determines what traffic should get access to limited resources when demand exceeds network bandwidth.

[1]We refer to WFQ [19] as the general scheduling mechanism with Virtual-Time/PGPS [47] and DWRR [54] as different implementations.

[2]NICs/switches support ~10 WFQs per port; buffer space is shared across the ports based on usage.

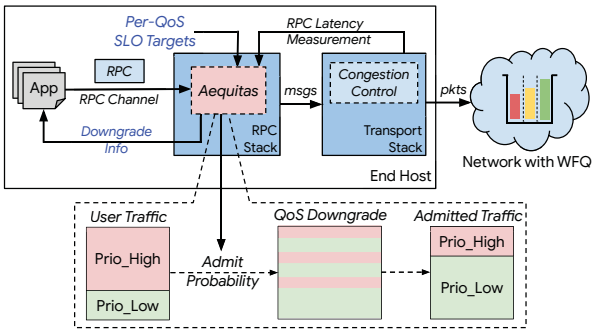


Figure 6: Aequitas system overview.

3 AEQUITAS OVERVIEW

3.1 Objectives and Challenges

Predictable RPC completion is a key performance goal in modern datacenter networks. Our goal in this work is to *provide RNL SLOs for RPC priority classes with performance requirements (PC and NC).* We face three challenges in achieving our goal:

(1) *Expressing SLOs for a diverse set of applications.* Latency degradation of RPCs can result from an overload of compute, storage, or networking. It is key to tease out an SLO that the network can be held accountable for.

(2) *Structural.* Given that overloads can occur anywhere in the network, the solution needs to handle dynamic overloads appearing anywhere along the path an RPC traverses.

(3) *Scale.* There may be tens of thousands of hosts, thousands of tenants, and hundreds of applications in a cluster, all requiring RPC performance at microsecond-scale.

3.2 System Overview

Figure 6 presents a high-level system diagram of Aequitas. Aequitas resides in the RPC layer and communicates with applications above it and network or transport stacks below it. Aequitas works at the RPC level and does not interfere with underlying congestion control, and it leverages weighted fair queuing (WFQ) available in commodity switches without making any modifications to existing hardware.

Applications issue RPCs on *RPC-channels*,[3] annotating their priority class, which maps to a *requested QoS* class. The operator provides the per-QoS RNL SLO targets. Once an RPC completes, Aequitas measures its RNL and feeds it into its admission control algorithm. By comparing the RNL SLO targets and the actual measurements, the algorithm

[3]An RPC-channel maps to one or more transport-layer socket/connection.

ϕ_i	QoS weight of class i
g_i	minimum guaranteed rate of class i
s_i	instantaneous service rate received for class i
r	total link capacity
a_i	instantaneous arrival rate of class i
a	aggregate instantaneous arrival rate at the link
μ	average load: average arrival rate over the period normalized to line rate
ρ	burst load: maximum instantaneous arrival rate normalized to line rate

Table 1: Notation used in Section 4 and beyond.

adjusts the amount of traffic admitted per destination-host for the QoS at which the RPC ran. In this way, Aequitas does not need extra signaling to determine the location of oversubscription points. Admission control is enforced in a fully distributed manner among all the hosts without requiring centralized knowledge. When admitting an RPC, Aequitas adopts a probabilistic approach by maintaining *admit probability* to determine if an RPC should be admitted or *downgraded* to a lower QoS level. Downgrade information is explicitly notified back to the application as a hint to adjust their RPC priorities.

In the next section, we show theoretically why Aequitas' central idea – managing RPC traffic admitted across QoS levels with WFQ – can be a powerful knob for realizing RNL SLOs in oversubscription situations.

4 ANALYTICAL RESULTS

RPC network-latency is primarily dictated by bandwidth and queuing-delay. In this section, we provide a theoretical characterization that motivated how we arrived at Aequitas design—controlling RPC network-latency across priority classes to provide differentiated SLOs by **controlling the amount of traffic admitted** on the respective QoS as realized by **WFQ**.

4.1 WFQ Bandwidth and Queuing-Delay Analysis

We find that WFQ is an excellent building block to help provide RNL SLOs as not only does it guarantee a *minimum bandwidth* for a traffic class, it also provides delay boundedness given its utilization level.

Given N QoS classes with $\phi_1, \phi_2, \cdots, \phi_N$ representing the weights of the WFQs that serve the QoS classes, the *minimum guaranteed rate* g_i for class i with line rate r is given by $g_i = \frac{\phi_i}{\sum_j \phi_j} r$. We assume lower i indicates a higher WFQ weight. WFQ is also *work-conserving*. If the instantaneous demand for a QoS class is lower than the rate above, its traffic is completely isolated from the other QoS classes and observes nearly zero queuing delay. Correspondingly, the bandwidth share of a QoS class may exceed the rate above when other QoS classes have aggregate demands lower than their share.

The seminal work in [47] describes the *delay guarantees* supported by WFQ: 1) the delay of a QoS level can be bounded

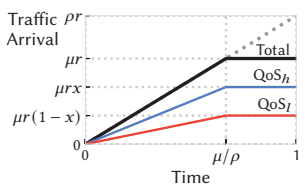


Figure 7: Traffic arrival pattern used in WFQ delay analysis.

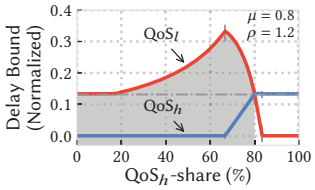


Figure 8: Theoretical worst-case delay with QoS_h:QoS_l weights=4:1.

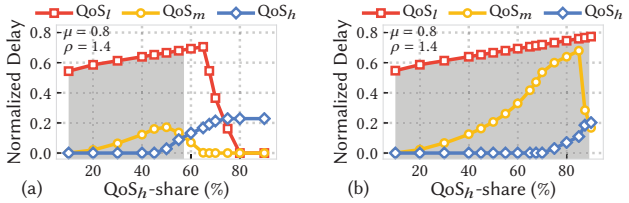


Figure 9: Simulated WFQ worst-case delay with 3 QoS levels under different QoS_h:QoS_m:QoS_l weights: (a) 8:4:1 and (b) 50:4:1. QoS-share of QoS_m and QoS_l is fixed at a ratio of 2:1.

as a function of its own queue length independent of other queues and arrivals of other levels; and that 2) it is feasible to compute the worst-case queuing delay when sources are constrained by leaky-bucket rate limiters. We build upon this work using Network Calculus [17] concepts to calculate delay bounds given different utilization levels in the QoS classes instead of finding the absolute worst-case delay bounds across all possible arrival curves. Formally, if we denote the arrival rate of a class i as a_i with the sum of arrival rates as a, and define **QoS-mix** as the N-tuple $(\frac{a_1}{a}, \frac{a_2}{a}, ..., \frac{a_N}{a})$; our analysis shows *how QoS-mix affects WFQ's per-QoS delay bounds in overload situations*. We refer to the i^{th} element in the QoS-mix as **QoS_i-share**. Compared to prior work, the analysis is more general in that it can provide delay bounds given a QoS-mix, but is less general as the closed-form equations are restricted to only two QoS levels.

Denote x as QoS_h-share of the QoS-mix, which is the ratio of QoS_h traffic to the total arrival rate $\frac{a_h}{a}$ where $0 < x < 1$; QoS_l-share is $(1-x)$. The ratio of QoS weights for QoS_h:QoS_l is ϕ:1. Consider the traffic pattern shown in Figure 7.[4] We define the entire sending period to be one unit of time. Traffic arrives in bursts characterized by a burst parameter, ρ, which is the slope of the black curve in the figure and $\rho > 1$. All notation used for analysis is defined in Table 1. For stability, there is an idle phase such that the average load μ within the period is less than 1.0. Thus, the delay bound can be represented as a fraction of the period; by definition, the arriving traffic can be consumed within a single period. We denote this as *normalized delay bound*.

Different subdomains for x yield different service curves and thus, a different delay-bound representation, e.g., up until a certain value of x, QoS_h experiences zero delay. Also,

[4]This model is similar to a Leaky-Bucket formulation expressed differently to aid closed-form equations, with the interval normalized to a unit of time.

some of the subdomains can be empty depending on the parameter values. Worst-case delay experienced by QoS_h as a function of x ($Delay_h(x)$) is given by (detailed proof is in Appendix B):

$$\begin{cases} 0, & x \le \frac{\phi}{\phi+1}\frac{1}{\rho} \\ \mu(\frac{\phi+1}{\phi}x - \frac{1}{\rho}), & \frac{\phi}{\phi+1}\frac{1}{\rho} < x \le \frac{\phi}{\phi+1} \\ \mu(1-x)(\phi+1-\frac{\phi}{\rho\, x}), & \frac{\phi}{\phi+1} < x \le \min\{1-\frac{1}{\phi+1}\frac{1}{\rho}, \frac{1}{\rho}\} \\ \mu(\frac{1}{\rho}-\frac{1}{\rho^2})\frac{1}{x}, & \min\{1-\frac{1}{\phi+1}\frac{1}{\rho}, \frac{1}{\rho}\} < x \le \frac{1}{\rho} \\ \mu(1-\frac{1}{\rho}), & x > \max\{\frac{\phi}{\phi+1}, \frac{1}{\rho}\} \end{cases} \tag{1}$$

Figure 8 plots the theoretical worst-case delay per QoS level in a 2-QoS scenario. There are two main takeaways from the above formulation. First, QoS-mix affects delay in both QoS classes. As QoS distribution changes, delay from both classes experience different regions. Second, at a certain share of QoS_h, we observe *priority inversion* where delay in QoS_h exceeds that of QoS_l.

Due to the increased complexity of closed-form delay equations, we extend the analysis to more number of QoS classes via empirical analysis in simulation. Figure 9 plots the simulated results of a WFQ with 3 QoS levels. As before, the QoS-mix plays an important role in the delay profile among all the QoS levels. Shaded area represents delay regions with no priority inversion.

4.2 Controlling *QoS*-Mix for RNL SLOs

The above set of equations show that the delay bound depends on a few different parameters including QoS-mix, QoS weights, and burstiness. We characterize two lemmas to conclude that controlling QoS-mix ties directly to delay-bounds, which along with the bandwidth guarantees provided by WFQs is effective in providing RNL SLOs.

Lemma 1. *When the demand for each QoS class (a_i) exceeds its minimum guaranteed rate (g_i), the QoS-share thresholds for priority inversion are a function of QoS weights.* Priority inversion happens when the delay in a higher QoS exceeds the delay in a lower QoS. In the case where the demand for each QoS class (a_i) exceeds its minimum guaranteed rate (g_i), it takes longer to process the traffic in QoS_i versus QoS_{i+1}. The processing time is proportional to a_i/ϕ_i, i.e.,

$$a_1/\phi_1 \le a_2/\phi_2 \le \ldots \le a_N/\phi_N. \tag{2}$$

For the two QoS case, this implies QoS_h and QoS_l,

$$\frac{a_h}{g_h} \le \frac{a_l}{g_l},\ \frac{x}{\phi} \le \frac{1-x}{1} \implies x \le \frac{\phi}{\phi+1}$$

Thus, the values for QoS weights determine the boundary of a region of operation outside which priority inversion occurs. We define it as the *admissible region*, formally as the region where each point satisfies

$$\forall k \in \{1, 2, \ldots, N-1\},\ delay\_bound_k \le delay\_bound_{k+1} \tag{3}$$

with N QoS classes; lower indices indicating higher priority. Figure 9 shows that when we increase the weight of QoS_h to 50, the priority inversion points (and hence the admissible region) move to the right, albeit at the cost of higher delay bounds for QoS_m.

Lemma 2. *While increasing QoS_h weight ϕ helps admit more QoS_h traffic with zero delay, beyond QoS_h-share exceeding $\frac{1}{\rho}$, delay is independent of QoS weights.* As ϕ increases, the domain of case 1 in Eq 1 grows and approaches $\frac{1}{\rho}$ as ϕ keeps increasing. In fact, based on Eq 1, when ϕ goes to infinity, QoS_h delay expression becomes

$$Delay_h(x) = \begin{cases} 0, & x \le \frac{1}{\rho} \\ \mu(x - \frac{1}{\rho}), & \frac{1}{\rho} < x \le 1 \end{cases} \tag{4}$$

Taking the two lemmas together, we observe while QoS weights are an important parameter to increase the amount of traffic permitted at a given delay bound, it is *controlling the QoS-share that is effective in providing the delay bounds in the first place.* This property directly motivates a key part of Aequitas' design – control utilization at a given QoS via admission control. We also note that as ϕ increases, the equations approach the single-QoS scenario where the only way to control delay is to control the amount of admitted traffic. Furthermore, the core idea of controlling QoS-mix enables cloud operators to select SLOs from the profiles of latency versus QoS-mix.

Aligning RPC priority to network QoS with WFQ scheduling and controlling the amount of traffic admitted to individual QoS classes are the core principles of Aequitas design which we describe in the next section.

5 SYSTEM DESIGN

The overall design for Aequitas follows naturally from the analysis above and consists of two sequential phases amenable to incremental deployment. In this paper, we focus on presenting the key design ideas without going into details about how Aequitas is implemented in the RPC stack and the transport stack internally at our clusters, which are beyond the scope of this paper.

Phase 1: Align Network QoS with RPC priority Most datacenter applications have a clear notion of RPCs that are *PC*, *NC*, and *BE*. Classifying an entire application or a job into a single priority class is too coarse-grained; transport level flows can be both coarse and fine grained, e.g., consider the issue we face with multiplexing multiple RPCs onto a single TCP connection; packet level is needlessly fine-grained and loses application-level semantics. As a first step

to providing RNL SLOs, Aequitas maps, at the *granularity of RPCs*, the three priority classes bijectively to three QoS classes served with WFQ-scheduling: *PC* RPCs to QoS_h, *NC* to QoS_m, and *BE* to QoS_l. Aequitas provides SLOs for QoS_h and QoS_m; QoS_l is treated as a scavenger class on which best-effort and downgraded traffic is served and offers no SLOs. The design organically extends to larger numbers of QoS priority classes.

Phase 2: Distributed Admission Control via QoS downgrade to provide RNL SLOs Aequitas uses a distributed algorithm implemented completely at sending hosts to decide, at the RPC granularity, whether to admit a given RPC on the requested QoS by controlling an *admit probability*. This controls the portion of RPCs admitted across QoS levels in order to meet RNL SLOs. In a departure from traditional mechanisms of admission-control that either drop or rate-limit traffic, Aequitas ***downgrades*** the unadmitted RPCs and issues them at the lowest QoS level. The algorithm follows an Additive Increase Multiplicative Decrease (AIMD) control.

5.1 Distributed Admission Control

At its core, Aequitas is a distributed admission control system for RPCs implemented completely at sending hosts utilizing a novel mechanism of QoS-downgrade enabled by WFQs commonly available in commodity switches.

Probabilistic admission of RPCs: Central to Aequitas' distributed algorithm is an *admit probability* denoted by p_{admit} that each RPC channel maintains on a per-*(src-host, dst-host, QoS)* basis; Aequitas probabilistically admits RPCs on a given QoS based on p_{admit}, which Aequitas controls as per Algorithm 1. Note that if an RPC is downgraded, it is **explicitly notified** to the application via an additional field in RPC metadata (lines 10-11). This notification is important for two reasons: (i) the application sees network overload and QoS downgrades directly, and (ii) when not all RPCs can be admitted on the requested QoS, the application has the freedom to control which RPCs are more critical and issue only those at higher QoS to prevent downgrades. How applications exactly use the downgrade information is outside the scope of this paper.

The key idea behind the algorithm is simple. At each source host, for each RPC channel, Aequitas collects measurements of RPC network latencies (RNL as described in §2.2.1) per destination host and QoS level to capture the delays incurred by both network overload and congestion control backoff. These measurements serve as the primary signal to adjust p_{admit}. If the latency is within the target, p_{admit} is increased, otherwise it is decreased. We find that such a probabilistic approach has two main advantages: (i) admit probability translates directly to determine the portion of RPCs that needs to be downgraded to control the amount of admitted traffic, and (ii) it is simple to reason about in terms of a fair and efficient distributed algorithm as we describe below. We note similarities to AQM schemes [22, 46] that also perform probabilistic admission control albeit at the packet level; Aequitas does so at the granularity of RPCs.

Algorithm 1: QoS Downgrade Algorithm

```
1  Notation:
     α: additive increment, β: multiplicative decrement,
     target_pctl: target percentile of tail latency, N: total number
     of QoS levels, priority: RPC priority class specified by the
     application, size: size of an RPC in number of MTUs.
2  Initialization:
     for i ← 1 to N − 1 do
3      p_admit[i] = 1
4      increment_window[i] =
         latency_target[i] · 100/(100−target_pctl[i])
5  On RPC Issue (rpc, priority):
6    QoS_req ← MapPriortyToQoS(priority)
7    if rand() ≤ p_admit[QoS_req] then
8      QoS_run ← QoS_req
9    else
10     QoS_run ← QoS_lowest
11     rpc.is_downgraded ← True
12   RPC_Start(rpc, QoS_run)
13 On RPC Completion (rpc_latency, size, QoS_run):
14   k ← QoS_run
15   if rpc_latency / size < latency_target[k] then
       ▷ Additive Increase
16     if now − t_last_increase[k] > increment_window[k] then
17       p_admit[k] ← min(p_admit[k] + α, 1)
18     t_last_increase[k] ← now
19   else                                  ▷ Multiplicative Decrease
20     p_admit[k] ← max(p_admit[k] − β · size, floor)
```

AIMD on admit probability: AIMD as a feedback control algorithm has been widely used to provide fair and efficient utilization of resources both in theory and in practice [11, 30]. Aequitas' usage of AIMD has several important differences compared to how other systems use AIMD.

Additive increase: Aequitas increases p_{admit} if the observed RNL is below the target, restricted to one update per *increment_window* (lines 15-18). The rationale is that for fairness, the increment in p_{admit} should be agnostic to how many RPCs each channel is sending. The value of *increment_window* depends on the percentile at which the SLO is defined, e.g., if the SLO is defined at the 99.9^{th}-p, the *increment_window* is higher than the case where it is at the 99^{th}-p—the algorithm is more conservative in increasing the admit probability when the SLO is for a higher tail.

Multiplicative decrease: If the RPC misses the specified SLO, p_{admit} is decreased by a constant amount per SLO miss (lines 19-20). Aequitas achieves fairness across RPC-channels. For this, when overload occurs, a channel sending more RPCs

incurs a larger decrease in its p_{admit} versus a channel sending fewer RPCs. We utilize RPC-level clocking to achieve this: the constant decrement in the admit probabilities implies that the overall decrease in a given time interval becomes proportional to the RPCs on the channel that miss the SLO.[5] We set a threshold below which p_{admit} does not further decrease. This is to prevent starvation – when the admit probability drops to zero, no new RPCs get admitted on the requested QoS, resulting in no further latency measurement for the admit probability to grow. Detailed evaluation of how the algorithm achieves fairness and efficiency is in §6.5.

By applying the above AIMD policy on the admit probability, Aequitas is able to control the rates of issuing RPCs in a fair and efficient way, and converge to a stable QoS-mix with which the given set of SLOs are precisely maintained. As we will show in §6.3, this leads to close to maximal admitted traffic while maintaining SLO-compliance.

Handling different RPC sizes: We make two augmentations to the algorithm to handle different RPC sizes. First, the *latency_target* is specified as a normalized SLO on an MTU basis, enabling larger RPCs to have a higher absolute RNL target. Second, the multiplicative decrease is made proportional to the size of the RPC, such that an SLO miss on, say, a 10-packet RPC behaves similarly to SLO misses on ten 1-packet RPCs. In other words, irrespective of the sizes of the RPCs, Aequitas will converge to its fair share.

5.2 SLO Guarantees and Robustness

The trifecta of aligning priorities, providing per-QoS SLOs (except the lowest QoS), and admission control to maintain a QoS-mix enables a systematic use of datacenter QoS and incentivizes applications to be well-behaved when using higher QoS.

A naive way of meeting SLOs is to admit a very small number of RPCs on each QoS. However, Aequitas aims to maximize the traffic that is admitted (performance-criterion) while retaining SLO-compliance (correctness-criterion). Additionally, we can show that in our *theoretical* model, at least $r\frac{\phi_i}{\sum\phi}\frac{\mu}{\rho}$ traffic is admitted in QoS$_i$ except for the lowest QoS To see why this is true, consider the model in §4.1. Denote X_i as the average rate that will at least be admitted in QoS_i under Aequitas. Given X_i, the maximum instantaneous rate on QoS_i can be represented as $X_i\frac{\rho}{\mu}$. When the arrival rate does not exceed its minimum guaranteed rate, there cannot be any delay on QoS_i (Appendix B.1 has a formal proof on this), and all traffic is admitted. Thus, if the maximum instantaneous arrival rate for QoS_i is less than g_i, X_i is guaranteed to be admitted:

$$X_i\frac{\rho}{\mu} \leq g_i = \frac{\phi_i}{\sum\phi}r,\ X_i \leq r\frac{\phi_i}{\sum\phi}\frac{\mu}{\rho}$$

with any additional SLO resulting in a larger value of X_i. Note that the guaranteed share is inversely proportional to the traffic burstiness and we evaluate this aspect in §6.4.

It is important to note that while Aequitas provides latency SLOs for all *admitted* RPCs, it does not guarantee the amount of traffic admitted on a per-application or per-tenant basis–wherein the admitted traffic depends on the number of co-existing applications/tenants, as Aequitas shares the per-QoS bandwidth. One can augment Aequitas to provide application/tenant traffic rate guarantees with a centralized RPC *quota server*, and we leave this for future work.

6 EVALUATION

Our evaluation consists of event-driven simulation, testbed experiments, and results from production deployment. The focus is on two aspects, first is to see if Aequitas remains SLO-compliant by controlling the 99^{th}-p (or 99.9^{th}-p) RNL which is our *correctness* criterion, and second is whether Aequitas admits close to ideal amount of traffic irrespective of the input traffic mix which serves as the main *performance* criterion. We evaluate these and additional aspects such as fairness and convergence over different topologies, RPC size distributions—both synthetic and from production, traffic patterns and burstiness. All results are at 100Gbps link rates.

6.1 Simulator

We use a packet-level simulator[6] built atop YAPS [35], augmenting it with WFQ scheduling in switch queues, Swift [34] congestion control, and an RPC stack where Aequitas is implemented. Our open source simulator also serves as a tool for datacenter operators to help define the admissible region and set the right SLOs. Unless otherwise specified, we use an α value of 0.01 and a β value of 0.01 per MTU (note the size-based adjustment to multiplicative decrease in Algorithm 1).

Validation: We validate the correctness of the simulator by replaying the theoretical 2-QoS scenario that was shown in Figure 8. Congestion control is disabled and the buffer size is set to a large value to closely match the theoretical model. We show the results in Figure 10 and observe that the simulator results precisely track the theory including priority inversion points and delay values barring QoS_l's delay, which is slightly higher in the simulation. We believe that this is a result of the packet nature of the simulator versus the fluid model used in theory.

Two experiment setups are common in our simulator-evaluation: (1) a 3-node setup where two clients send RPCs to the same destination server for microbenchmarks, and (2)

[5] An implication of this is if a channel's rate of RPCs is within its fair share, its admit probability will converge to 1.0.

[6] https://github.com/SymbioticLab/Aequitas

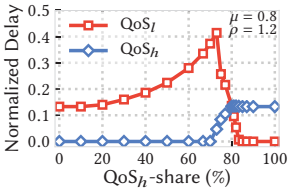


Figure 10: Simulated WFQ delay bounds with QoS_h:QoS_l weights = 4:1.

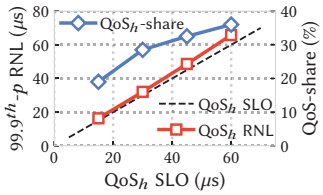


Figure 11: Aequitas provides SLO-compliance: achieved RNL closely tracks SLOs.

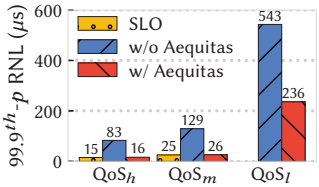


Figure 12: Aequitas significantly improves RNL, closely tracking the SLOs.

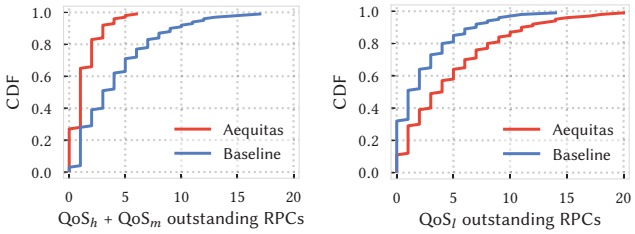


Figure 13: Comparison of number of outstanding RPCs per switch-port before and after Aequitas.

a 33-node or 144-node setup with an all-to-all traffic pattern where each host sends RPCs to the other hosts with an average and burst load of 0.8 and 1.4 respectively (similar to Figure 7) with Poisson arrivals. Setups that differ from above are described for corresponding experiments. QoS weights for experiments with 2-QoS levels are set at 4:1 and with 3-QoS levels are set at 8:4:1.

6.2 SLO Compliance

To evaluate how closely Aequitas' distributed admission control tracks per-QoS RNL SLOs, we start with the 3-node topology where two RPC channels, each running on a different host, issue 32KB WRITE RPCs on QoS_h to a destination server. To make the network persistently overloaded, each host issues RPCs at line rate with 70% of its RPCs at QoS_h and 30% of its RPCs at QoS_l. Figure 11 shows that Aequitas tracks the SLO target in terms of the 99.9th-p RNL for QoS_h extremely well as it is varied from 15μs to 60μs. The tradeoff between SLO targets and admitted traffic is also evident, with stricter SLOs resulting in fewer RPCs admitted on QoS_h.

We move onto the 33-node setup to illustrate that Aequitas preserves its ability to closely track RPC latency SLOs under different communicate patterns at a larger scale. We set the input *QoS-mix* of (QoS_h, QoS_m, QoS_l) to be (0.6, 0.3, 0.1) and show the achieved RNL at the 99.9th-p w/ and w/o Aequitas with the SLOs selected as 15μs and 25μs for QoS_h and QoS_m,

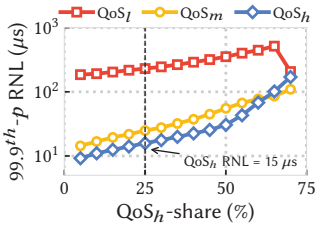


Figure 14: Baseline (w/o Aequitas) 99.9th-p RNL observed as QoS_h-share is varied.

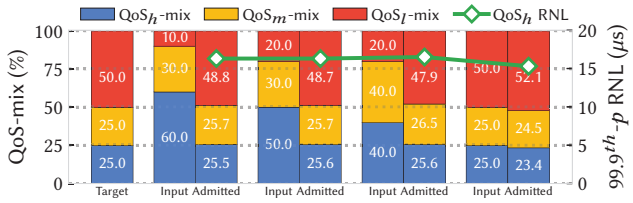


Figure 15: Aequitas admits close to maximal traffic while retaining SLO-compliance irrespective of input QoS-mix.

respectively, in Figure 12. We treat QoS_l as the scavenger class as discussed in §5.

An interesting, and perhaps surprising, observation is that with Aequitas, the RNL of QoS_l reduces as well, i.e., **Aequitas is not a zero-sum game for per-QoS latencies.** The reasoning behind this is similar to the result in Reference [8] where all jobs can improve their performance with right prioritization. Given the improved RNL for QoS_h and QoS_m, RPCs in QoS_l have fewer RPCs to contend with, which as per Little's Law [37] implies that they will finish quicker as well. To verify, we collect the number of outstanding RPCs and show in Figure 13 that the decrease in instantanous outstanding RPCs in QoS_h + QoS_l indeed outweighs the increase in QoS_l outstanding RPCs, especially at the tail.

6.3 Maximizing Admitted Traffic within SLOs

While SLO-compliance is the main correctness criterion, it can be met *trivially* by admitting a tiny amount of traffic. We show that Aequitas admits close to maximal traffic while retaining SLO-compliance, irrespective of the input *QoS-mix*.

To figure out the maximal admissible traffic associated with a given SLO, we measure 99.9th-p RNL in the 33-node setup without Aequitas as we vary QoS_h-share from 5 to 70%, keeping QoS_m at 25% and allotting the rest to QoS_l, as shown in Figure 14. We set the SLO for QoS_h at 15μs which corresponds to QoS_h-share of 25%, and SLO for QoS_m at 25us. Thus, in this setup, 25% is the maximal amount of traffic we can admit as admitting any more traffic will violate the correctness criterion of SLO-compliance. We then vary the input *QoS-mix* and plot both RNL and admitted *QoS-mix* in Figure 15. We can see that Aequitas converges closely to the maximal QoS_h-share while retaining SLO-compliance.

Further, we observe that Aequitas' algorithm is *self-consistent*, i.e., if the input *QoS-mix* is same as target, then

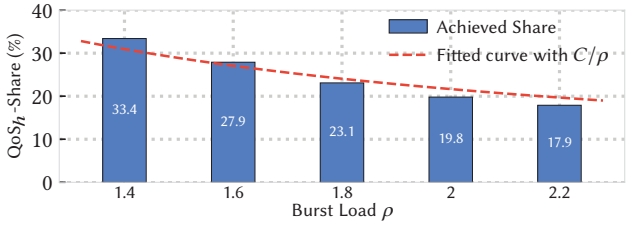


Figure 16: Aequitas adjusts admitted traffic that is inversely proportional to traffic burstiness.

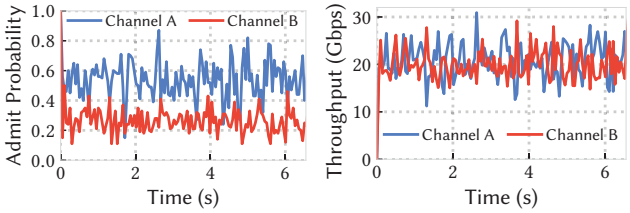


Figure 17: Admit probability and throughput of two RPC channels sending 80Gbps and 40Gbps QoS_h traffic with QoS_h SLO set to 15μs.

very little traffic gets downgraded. The corollary of this result is that Aequitas helps solve the *race to the top* problem defined in §2.3 by effectively controlling the *QoS-mix* independent of the input distribution.

6.4 Impact of Burstiness on Admitted Traffic

As we discussed in in §5.2, for a given set of SLOs, as burstiness of the traffic increases, the amount of traffic for which those SLOs can be provided for decreases. In Figure 16, we vary the burst load, ρ, and plot the QoS_h-share that Aequitas admits. While the simulation differs from the theoretical model in many ways such as packet-level behavior and congestion-control, we can observe, via the fitted curve in the plot, the inverse proportionality of admitted traffic w.r.t. burst load as per the theoretical formulation.

6.5 Fairness

Besides providing SLO guarantees, fairness is also a key goal in Aequitas' admission control. To evaluate if Aequitas ensures fairness across RPC channels, we modify the 3-node experiment in §6.2 such that Channel A issues 40% of its RPCs on QoS_h (equivalent to 40 Gbps worth of RPCs at 100Gbps link as the per-channel load is 1.0) whereas Channel B issues 80% of its RPCs on QoS_h. We set QoS_h SLO to be 15μs such that fairness implies that each channel gets 20% of its RPCs admitted to QoS_h and the rest downgraded. Figure 17 shows that Aequitas achieves fairness by converging to different values of admit probability for each individual channels.

Another important aspect regarding fairness is how Aequitas behaves when a channel is operating within its quota, i.e., its demand for QoS_h is below its fair-share. The expectation is that such a well-behaved RPC channel experiences minimal to no downgrades while continuing to meet the SLOs. We modify the above experiment in that Channel A issues only 10% of its RPCs on QoS_h (lower than its fair share of

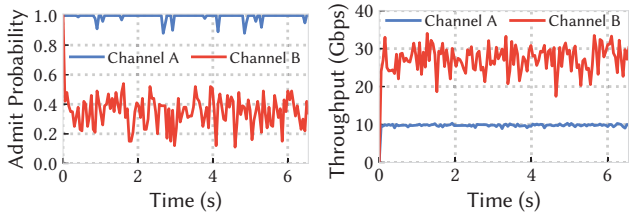


Figure 18: Aequitas maintains a near 1.0 admit probability for in-quota RPC channels that have demand less than fair-share. Excess quota is reclaimed by other channels to provide max-min fairness.

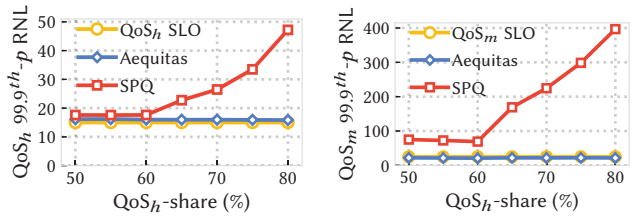


Figure 19: Aequitas compares with Strict Priority Queuing (SPQ) in providing SLO guarantees.

20%) and plot the admit probabilities and achieved throughput in Figure 18. We find that Aequitas not only maintains admit probability of Channel A close to 1.0, it also allows Channel B to reclaim the excess, providing max-min fairness. We observe Channel A is able to consistently achieve a throughput of 10Gbps with 1^{st}-p p_{admit} = 0.82.

α and β are key parameters in that they posit a tradeoff between SLO-compliance and stability. Results of a sensitivity analysis on α and β can be found in Appendix C.

6.6 Convergence Time

Convergence time affects how quickly Aequitas' algorithm achieves max-min fair-sharing amongst channels while ensuring SLO-compliance. Convergence time for both channels is 10ms in Figure 17 and 3ms in Figure 18. As above, α and β are key parameters here and we choose them to favor SLO-compliance.

6.7 Comparison with Strict Priority Queuing

Although strict priority queuing (SPQ) is not widely deployed in many production networks, it is widely used in literature to implement optimal scheduling such as SRPT [3, 40]. We evaluate how using SPQ alone handles network overloads by applying the same setting in our 33-node setup and replacing the underlying WFQs with SPQs. We fix the QoS_m distribution at 20% and increase the percentage of QoS_h traffic as shown in Figure 19. We observe SPQ fails to maintain predictability as more applications mark their RPCs as QoS_h; meaning, it does not resolve the *race to the top* problem.

6.8 Handling Different RPC Sizes

We now evaluate how Aequitas handles different RPC sizes as discussed in §5.1. We conduct an experiment where half the channels continue to issue 32KB RPCs while the other half

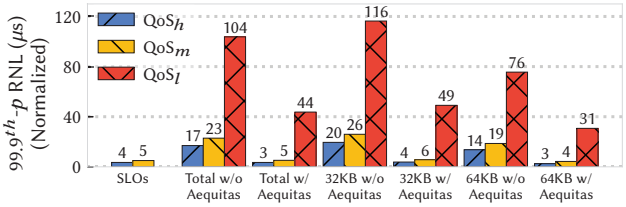


Figure 20: Aequitas uses RPC size to normalize latency with a non-uniform size distribution in a 33-node cluster.

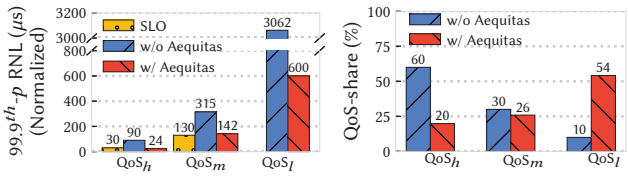


Figure 21: Aequitas' performance in a large (144-node) topology with production RPC sizes.

issue 64KB RPCs. Figure 20 shows that Aequitas continues to meet the *normalized* RNL SLOs with different RPC sizes.

6.9 Large-Scale Eval with Production RPC Sizes

We evaluate Aequitas at a larger scale in a simulated 144-node topology, with RPC sizes taken from production, under extreme overload where we increase the burst load such that the maximum instantaneous load on the link is 25× its capacity. Figure 21 shows that Aequitas improves tail RNL in QoS_h/QoS_m by 3.7×/2.2× in this workload, continuing to meet RNL SLOs even under extreme overloads that can occur in production. We observe 20ms convergence time before the 99.9th-p latency becomes stable.

6.10 Comparison with Related Works

We compare Aequitas with four related systems, pFabric [3], QJump [26], D^3 [62], PDQ [28], and Homa [40], each with a complete implementation in our packet-level simulator. We use our production RPC size distribution with 50%/30%/20% input QoS-mix in the 33-node setup. We start with normalized SLO targets; for D^3 and PDQ, which do not consider RPC sizes in their design, these translate to 250us and 300us deadlines for QoS_h and QoS_m RPCs based on the average of production RPC-size distribution, respectively.

We record per-QoS 99.9th-p RNL and percentage of traffic that meet the SLO targets from their initially assigned QoS levels. We also record network utilization, which we define as the achieved goodput divided by the maximum goodput based on the input arriving rate. Figure 22 summarizes the results and we make the following observations. First, Aequitas achieves the highest amount of traffic meeting SLO targets (QoS_m results, not shown, are similar). Second, Aequitas achieves better 99.9th-p RNL for QoS_h and QoS_m than pFabric and QJump, which are SLO-unaware. D^3 and PDQ observe good performance on RNL, however, a lower percentage of traffic meets the SLO targets / deadlines. This is because both D^3 and PDQ terminate an RPC early when it

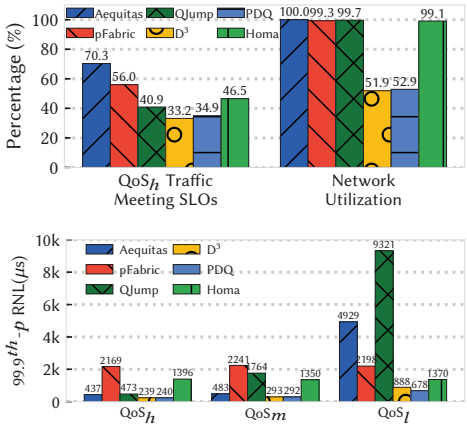


Figure 22: Aequitas compared with related works in the simulated 33-node setup with production RPC size distribution.

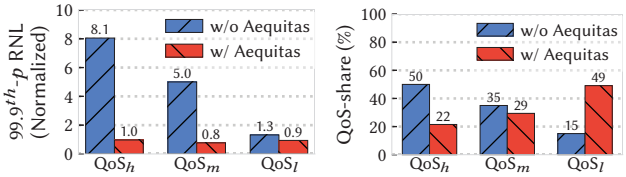


Figure 23: Result from testbed implementation shows that Aequitas maintains SLO compliance and converges to the target QoS-mix.

cannot meet its deadline, which also lowers network utilization down to nearly 50%. pFabric favors short RPCs using SRPT scheduling, but doesn't meet SLO targets for large RPCs, which can be equally important. Homa also adopts SRPT, but its usage of dynamic in-network priorities favors even more applications' small RPCs, leaving more large RPCs' SLO goals to be ignored. QJump provides good performance at the packet level by rate-limiting higher QoS traffic at end-hosts, however at RPC level, Aequitas' performance is better both in terms of RNL and percentage of traffic meeting SLOs.

6.11 Testbed Evaluation

Our prototype implementation of Aequitas is built in a production RPC stack, and it incorporates both Phase 1 and Phase 2 of Aequitas' design. Aequitas' algorithm computes an admit probability per RPC channel, which is mapped to multiple per-QoS TCP sockets. On RPC completion, the RNL measurement is fed into the Aequitas' algorithm.

We deployed the prototype in a 20-machine testbed with 100Gbps NICs connected to a single switch that supports ~10 QoS queues with configurable weights. We set the QoS_h:QoS_m:QoS_l weights to 8:4:1. Each machine issues 32KB WRITE RPCs to other machines in an all-to-all communication pattern. To circumvent the issues in RNL measurements described in §2.2.1, we provision enough CPU such that the elevated network-latency measurement is purely a result of network overload.

Figure 23 shows the RNL SLOs and QoS-mix w/ and w/o Aequitas. The setup is similar to §6.3 with 3 QoS levels and

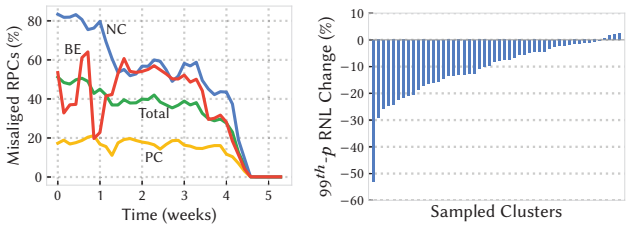


Figure 24: Production deployment of Phase 1 of Aequitas shows improvement in QoS-misalignment and 99th-p RNL.

an input QoS-mix at (0.5, 0.35, 0.15). The SLOs are set as per a QoS-mix of (0.2, 0.3, 0.5). For confidentiality reasons, we show normalized[7] RNL measurements and find that Aequitas meets its promise of achieving SLO targets.

6.12 Results from Production Deployment

We now present results from Aequitas' production deployment with a focus on Phase 1 deployment. Phase 2 results are not ready to be collected as the paper is written. Figure 24 shows the results (collected from a random sampling of 50 clusters) from our fleet-wide deployment of Phase 1, with two important metrics:

1) Misalignment percentage between RPC-priority and QoS, i.e., percentage of *PC* RPCs flowing in QoS_l and QoS_m queues, *NC* RPCs flowing in QoS_h and QoS_l queues, and *BE* RPCs flowing in QoS_h and QoS_m queues. Aequitas brings misalignment down from up to 80% to nearly zero.

2) Aequitas achieves up to 53% reduction in 99^{th}-p RNL for high priority traffic. RNL was measured as described in §2.2.1 for Storage READ and WRITE RPCs. A small number of clusters showed a minor regression primarily because of initial highly skewed QoS distribution, competing traffic that is not yet aligned through Aequitas, and changes in traffic pattern during the measurement window.

7 RELATED WORK

Packet Scheduling: Packet scheduling is a classic networking topic that focuses on different variations of weighted fair queuing (WFQ) in switches, routers, and middleboxes [9, 19, 24, 25, 41, 47, 54, 60]. Aequitas builds upon this pioneering work and extends the analysis by exploring the interactions among QoS levels as well as WFQ's admissible regions to provide SLOs for higher-priority RPCs. Centralized packet arbitration solutions [43, 48] could provide latency guarantees, but are difficult to deploy at scale in datacenters.

Flow and Coflow Scheduling: Flow scheduling algorithms in datacenters focus on minimizing the average or tail flow completion time (FCT), typically by prioritizing short flows [3, 5, 21, 23, 40]. Aequitas works at the RPC granularity, provides guaranteed SLOs for RPC network-latency, and uses WFQ mechanism versus strict priority. Coflow scheduling minimizes the average coflow completion time (CCT) instead of FCT [1, 14, 15, 20], but it does not capture RPC semantics either.

Network Calculus-Based Scheduling: Network calculus has been widely used as a tool to provide worst-case latency guarantees by many related works [18, 31, 51, 57, 66–68]. However, none highlighted the interactions between QoS levels and the possibility of priority inversion. PriorityMeister [68], SNI-Meister [66], and WorkloadCompactor [67] target tail latency SLOs, but they all base their analysis on SPQ and rely on prior knowledge of stationary traffic traces.

AQM and QoS Solutions: Active Queue Management (AQM) performs admission control by probabilistic dropping at the packet layer to prevent congestion under oversubscription while approximating fairness [22, 42, 45, 46, 64]. Aequitas uses a similar probabilistic admission control, but at the layer of RPCs. QoS-based solutions such as IntServ [63] and DiffServ [10] provide applications better service relative to *BE* traffic running on the Internet. Aequitas is similar to DiffServ in that it also differentiates traffic at the edges as per their priority, but focuses on datacenter environments to provide guaranteed latency SLOs for RPCs.

Congestion Control: Datacenter congestion control solutions have focused on quickly achieving max-min fairness with or without hardware support and using edge-based (delay-based) and network-supported (e.g., using ECN or programmable switch) mechanisms [2, 34, 36, 39, 40, 69]. Aequitas, which operates at the RPC layer, relies on a well-functioning congestion control algorithm at the transport layer to keep switch buffer occupancy small, alleviate packet-losses, and fully utilize available bandwidth.

Network Bandwidth Sharing: Over the last decade, another prominent direction of research has been network bandwidth sharing in public and/or private clouds [6, 7, 13, 27, 32, 41, 49, 50, 53]. Similar solutions at the WAN level include, among others, BwE [33] and SWAN [29]. Aequitas differs from these efforts by focusing on providing isolation in terms of RPC latency to critical RPCs.

Server-Side RPC Overload Management: Another line of work is to provide overload management at the layer of RPCs or request/responses but focuses on server-side overload such as CPU contention [12, 61, 65]. Aequitas complements such schemes by focusing on network overload, providing guarantees on RPC network-latency.

8 CONCLUSION

Today, developers running in shared multi-tenant cloud environments have no effective way to provide RPC latency SLOs. In this paper, we take an important step toward this higher-level goal by providing a first solution for SLOs for the network component of RPC latency. We present the

[7] We normalize each QoS level with their observed 99.9^{th}-p RNL when input QoS-mix is same as the target QoS-mix (0.2, 0.3, 0.5).

design, implementation, and evaluation of Aequitas, built around the observation that dynamically mapping RPCs to widely-available network QoS classes can bound RPC network-latency in a shared environment with no centralized control. We employ Network Calculus-based analysis to set the ratio of RPCs admitted into the network at different priority levels with the goal of guaranteeing quantitative tail latency targets for all priority classes except for the lowest (best-effort) class. We hope our work will inspire other techniques to comprehensively deliver RPC latency SLOs as a fundamental building and reasoning block for distributed systems developers. On the theoretical side, we leave an open question: what are the closed-form delay equations for an arbitrary number of QoS levels, if any? Solving this challenge, or proving non-existence of such generalization, will provide valuable insights in designing future QoS-aware systems.

ACKNOWLEDGEMENTS

We would like to thank Neal Cardwell, David Wetherall, the anonymous SIGCOMM reviewers, and our shepherd, Y. Richard Yang, for providing valuable feedback. This work was supported in part by NSF grants CNS-1845853, CNS-1845853, and CNS-2104243.

REFERENCES

[1] Saksham Agarwal, Shijin Rajakrishnan, Akshay Narayan, Rachit Agarwal, David Shmoys, and Amin Vahdat. 2018. Sincronia: Near-optimal network design for coflows. In *SIGCOMM*.

[2] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data Center TCP (DCTCP). In *SIGCOMM*.

[3] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick Mckeown, Balaji Prabhakar, and Scott Shenker. 2013. pFabric: Minimal Near-Optimal Datacenter Transport. In *SIGCOMM*.

[4] Sebastian Angel, Hitesh Ballani, Thomas Karagiannis, Greg O'Shea, and Eno Thereska. 2014. End-to-end Performance Isolation Through Virtual Datacenters. (2014).

[5] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. 2015. Information-Agnostic Flow Scheduling for Commodity Data Centers. In *NSDI*.

[6] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. 2011. Towards predictable datacenter networks. In *SIGCOMM*.

[7] Hitesh Ballani, Keon Jang, Thomas Karagiannis, Changhoon Kim, Dinan Gunawardena, and Greg O'Shea. 2013. Chatty tenants and the cloud network sharing problem. In *NSDI*.

[8] Nikhil Bansal and Mor Harchol-Balter. 2001. Analysis of SRPT Scheduling: Investigating Unfairness. In *Proceedings of the 2001 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '01)*. Association for Computing Machinery, New York, NY, USA, 279–290. https://doi.org/10.1145/378420.378792

[9] J.C.R. Bennett and H. Zhang. 1996. WF^2Q: Worst-case Fair Weighted Fair Queueing. In *INFOCOM*.

[10] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. 1998. RFC2475: An Architecture for Differentiated Service. In *IETF*.

[11] Dah-Ming Chiu and Raj Jain. 1989. Analysis of Increase and Decrease Algorithms for Congestion Avoidance in Computer Networks. In *Computer Networks and ISDN systems*.

[12] Inho Cho, Ahmed Saeed, Joshua Fried, Seo Jin Park, Mohammad Alizadeh, and Adam Belay. 2020. Overload Control for µs-scale RPCs with Breakwater. In *NSDI*.

[13] Mosharaf Chowdhury, Zhenhua Liu, Ali Ghodsi, and Ion Stoica. 2016. HUG: Multi-Resource fairness for correlated and elastic demands. In *NSDI*.

[14] Mosharaf Chowdhury and Ion Stoica. 2015. Efficient Coflow Scheduling Without Prior Knowledge. In *SIGCOMM*.

[15] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. 2014. Efficient Coflow Scheduling with Varys. In *SIGCOMM*.

[16] Jeremy Cloud. 2013. Decomposing twitter: Adventures in serviceoriented architecture. In *n QCon New York*.

[17] Rene L. Cruz. 1991. A calculus for network delay, Part I: Network elements in isolation. In *IEEE/ACM Transactions on Information Theory*.

[18] Rene L Cruz. 1992. Service burstiness and dynamic burstiness measures: A framework. *Journal of High Speed Networks* 1, 2 (1992), 105–127.

[19] A. Demers, S. Keshav, and S. Shenker. 1989. Analysis and Simulation of a Fair Queueing Algorithm. In *SIGCOMM*.

[20] Fahad Dogar, Thomas Karagiannis, Hitesh Ballani, and Ant Rowstron. 2014. Decentralized Task-Aware Scheduling for Data Center Networks. In *SIGCOMM*.

[21] Abdullah Bin Faisal, Hafiz Mohsin Bashir, Ihsan Ayyub Qazi, Zartash Uzmi, and Fahad R. Dogar. 2018. Workload Adaptive Flow Scheduling. In *CoNEXT*.

[22] Sally Floyd and Van Jacobson. 1993. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Transaction on Networking* 1, 4 (1993), 397–1413.

[23] Peter X. Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2015. pHost: Distributed Near-Optimal Datacenter Transport Over Commodity Network Fabric. In *CoNEXT*.

[24] S. Jamaloddin Golestani. 1995. Network delay analysis of a class of fair queueing algorithms. *IEEE JSAC* 13, 6 (1995), 1057–1070.

[25] Pawan Goyal, Harrick M Vin, and Haichen Chen. 1996. Start-time fair queueing: A scheduling algorithm for integrated services packet switching networks. In *SIGCOMM*.

[26] Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert N. M. Watson, Andrew W. Moore, Steven Hand, and Jon Crowcroft. 2015. Queues don't matter when you can JUMP them!. In *NSDI*.

[27] 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 *CoNEXT*.

[28] Chi-Yao Hong, Matthew Caesar, and P. Brighten Godfrey. 2012. Finishing Flows Quickly with Preemptive Scheduling. In *SIGCOMM*.

[29] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. 2013. Achieving high utilization with software-driven WAN. In *SIGCOMM*.

[30] Van Jacobson and Michael J. Karels. 1988. Congestion Avoidance and Control. In *SIGCOMM*.

[31] Keon Jang, Justine Sherry, Hitesh Ballani, and Toby Moncaster. 2015. Silo: Predictable message latency in the cloud. In *SIGCOMM*.

[32] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazieres, Balaji Prabhakar, Changhoon Kim, and Albert Greenberg. 2013. EyeQ: Practical Network Performance Isolation at the Edge. In *NSDI*.

[33] Alok Kumar, Sushant Jain, Uday Naik, Nikhil Kasinadhuni, Enrique Cauich Zermeno, C. Stephen Gunn, Jing Ai, Björn Carlin, Mihai Amarandei-Stavila, Mathieu Robin, Aspi Siganporia, Stephen Stuart, and Amin Vahdat. 2015. BwE: Flexible, Hierarchical Bandwidth Allocation for WAN Distributed Computing. In *SIGCOMM*.

[34] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan M. G Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn,

Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. 2020. Swift: Delay is Simple and Effective for Congestion Control in the Datacenter. In *SIGCOMM*.

[35] Gautam Kumar, Akshay Narayan, and Peter Gao. 2016. YAPS: Yet Another Packet Simulator. https://github.com/NetSys/simulator. (2016).

[36] Yuliang LI, Harry Hongqiang Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. 2019. HPCC: High Precision Congestion Control. In *SIGCOMM*.

[37] John D. C. Little. 1961. A proof for the queuing formula: L=λW. In *Operations Research*.

[38] Tony Mauro. 2015. Adopting Microservices at Netflix: Lessons for Architectural Design. https://tinyurl.com/htfezlj. (2015).

[39] Radhika Mittal, 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. In *SIGCOMM*.

[40] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. 2018. Homa: A Receiver-Driven Low-Latency Transport Protocol Using Network Priorities. In *SIGCOMM*.

[41] Kanthi Nagaraj, Dinesh Bharadia, Hongzi Mao, Sandeep Chinchali, Mohammad Alizadeh, and Sachin Katti. 2016. NUMFabric: Fast and Flexible Bandwidth Allocation in Datacenters. In *SIGCOMM*.

[42] Kathleen Nichols and Van Jacobson. 2012. Controlling Queue Delay: A modern AQM is just one piece of the solution to bufferbloat. In *ACM Queue*.

[43] Amy Ousterhout, Jonathan Perry, Hari Balakrishnan, and Petr Lapukhov. 2017. Flexplane: An experimentation platform for resource management in datacenters. In *NSDI*.

[44] Dan Paik. 2016. Adapt or Die: A microservices story at Google. https://www.slideshare.net/apigee/adapt-or-die-a-microservices-story-at-google. (2016).

[45] Rong Pan, Lee Breslau, Balaji Prabhakar, and Scott Shenker. 2003. Approximate Fairness through Differential Dropping. In *SIGCOMM*.

[46] Rong Pan, Balaji Prabhakar, and Konstantinos Psounis. 2000. CHOKe: A stateless active queue management scheme for approximating fair bandwidth allocation. In *INFOCOM*.

[47] Abhay K. Parekh and Robert G. Gallager. 1993. A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Single-Node Case. In *IEEE/ACM Transactions on Networking*.

[48] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. 2014. Fastpass: A centralized zero-queue datacenter network. (2014).

[49] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica. 2012. FairCloud: Sharing the Network in Cloud Computing. In *SIGCOMM*.

[50] 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 *SIGCOMM*.

[51] Hanrijanto Sariowan, Rene L. Cruz, and George C. Polyzos. 1999. SCED: A Generalized Scheduling Policy for Guaranteeing Quality-of-Service. In *IEEE/ACM Transactions on Networking*.

[52] Cristian Satnic. 2016. Amazon, Microservices and the birth of AWS cloud computing. https://www.linkedin.com/pulse/amazon-microservices-birth-aws-cloud-computing-cristian-satnic/. (2016).

[53] Alan Shieh, Srikanth Kandula, Albert Greenberg, and Changhoon Kim. 2011. Sharing the Data Center Network. In *NSDI*.

[54] Madhavapeddi Shreedhar and George Varghese. 1995. Efficient fair queueing using deficit round robin. In *SIGCOMM*.

[55] David Shue, Michael J. Freedman, and Anees Shaikh. 2012. Performance Isolation and Fairness for Multi-Tenant Cloud Storage. (2012).

[56] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*. Technical Report. Google, Inc. https://research.google.com/archive/papers/dapper-2010-1.pdf

[57] Ion Stoica, Hui Zhang, and TS Ng. 1997. A hierarchical fair service curve algorithm for link-sharing, real-time, and priority services. In *SIGCOMM*.

[58] Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Ant Rowstron, Tom Talepy, Richard Black, and Timothy Zhu. 2013. IOFlow: A Software-Defined Storage Architecture. In *SOSP*.

[59] Andrew Wang, Shivaram Venkataraman, Sara Alspaugh, Randy Katz, and Ion Stoica. 2012. Cake: enabling high-level SLOs on shared storage systems. In *SoCC*.

[60] Xiao-Dong Wang, Xiao Chen, Jie Min, and Yu Zhou. 2012. A Priority-Based Weighted Fair Queueing Algorithm in Wireless Sensor Network. In *WiCom*.

[61] Matt Welsh and David Culler. 2002. Overload Management as a Fundamental Service Design Primitive. In *SIGOPS*.

[62] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowstron. 2011. Better Never than Late: Meeting Deadlines in Datacenter Networks. In *SIGCOMM*.

[63] J. Wroclawski. 1997. RFC 2210: The Use of RSVP with IETF Integrated Services. In *IETF*.

[64] David Zats, Anand Padmanabha Iyer, Ganesh Anantharayanan, Rachit Agarwal, Randy Katz, Ion Stoica, and Amin Vahdat. 2015. FastLane: Making Short Flows Shorter with Agile Drop Notification. In *SOCC*.

[65] Hao Zhou, Ming Chen, Qian Lin, Yong Wang, Xiaobin She, Sifan Liu, Rui Gu, Beng Chin Ooi, and Junf. 2018. Overload Control for Scaling WeChat Microservices. In *SoCC*.

[66] Timothy Zhu, Daniel S. Berger, and Mor Harchol-Balter. 2016. SNC-Meister: Admitting More Tenants with Tail Latency SLOs. In *SoCC*.

[67] Timothy Zhu, Michael A. Kozuch, and Mor Harchol-Balter. 2017. WorkloadCompactor: Reducing datacenter cost while providing tail latency SLO guarantees. In *SoCC*.

[68] Timothy Zhu, Alexey Tumanov, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. 2014. PriorityMeister: Tail Latency QoS for Shared Networked Storage. In *SoCC*.

[69] 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. In *SIGCOMM*.

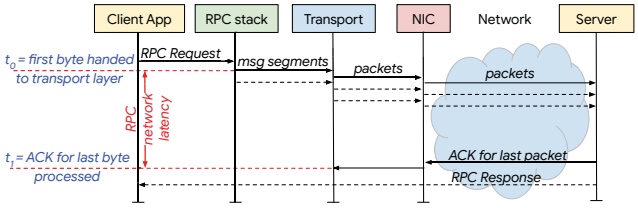


Figure 25: Latency breakdown of a WRITE RPC. The RPC network latency (RNL) measures the difference between the time when the first RPC packet arrives at the L4 layer and the time when the last RPC packet is acknowledged.

A MEASURING RNL

RNL is the portion of RPC latency impacted by network overloads. Figure 25 shows the life of a Storage WRITE RPC; the discussion applies similarly to READ RPCs. A complete Storage RPC operation consists of a request followed by a response. As Figure 25 shows, it constitutes a significant portion of overall RPC latency, especially when network bandwidth is constrained. RNL is defined by $t_1 - t_0$ where t_0 is the time when the first RPC packet arrives at the L4 transport layer, e.g., TCP and t_1 is the time when the last packet of the RPC is acknowledged at the transport. We focus on the payload portion of the RPC as we observe that total data transmitted in a complete RPC operation (a request followed by a response) is dominated by the side which contains the actual payload of the RPC—the response of a READ RPC is much larger than the request (200:1 on average in our clusters), and the request of a WRITE RPC is much larger than its response (400:1 on average).

There are two main challenges in precisely measuring RNL in production stacks: (i) RPC boundaries may not be known precisely at the transport layer, as is the case with Linux kernel TCP, and (ii) RNL can still include delays unrelated to network overload, such as delays due to insufficient CPU, interrupts, flow control or kernel scheduler. One approach is to measure RNL t_0 and t_1 at the *sendmsg* boundary. As Aequitas incorporates RPC size in the SLO, this works well even if an RPC consists of multiple *sendmsg* calls.

B WFQ DELAY ANALYSIS

B.1 When Delay Occurs in WFQ

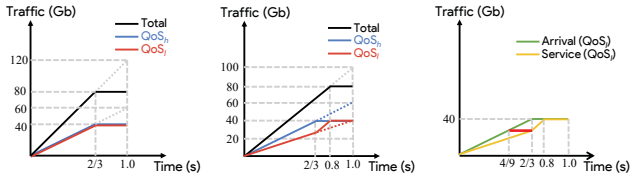
Delay occurs in a WFQ class *iff* (1) the total instantaneous input arrival rate on the link is greater than the link capacity (i.e., in the presence of overload), and (2) the arrival rate of the class is greater than the service rate of the class:

$$\sum a_i > r \text{ and } a_i > s_i \tag{5}$$

where $a_i \geq s_i \geq min(a_i, g_i)$. Further, note that when the total arrival rate is greater than the link rate r, the total amount of service rate which WFQ can provide is fixed at r by the definition of work conservation:

$$\sum s_i = r, \ if \sum a_i > r \tag{6}$$

The expression for s_i is not immediately given as one needs to consider both QoS_i's own guaranteed rate g_i as well as a share of unused rate from other QoS classes due to the work conserving nature of WFQ.



(a) Arrival Curve **(b) Service Curve** **(c) Delay Bound**

Figure 26: Applying network calculus on WFQ with 2 QoS levels.

Thus, the value s_i changes dynamically as the QoS-mix varies (e.g., when all the traffic from some QoS class has been consumed). In other words, QoS-mix in a WFQ can affect the delay profile of each QoS class. If the number of classes is 2, QoS_h and QoS_l, we can show that delay occurs on QoS_i if the following condition holds true:

$$\sum a_i > r \text{ and } a_i > g_i \tag{7}$$

We will show this for QoS_h, and QoS_l follows from symmetry. To prove this, we will show that in the two QoS case, if $a_h > s_h$, then $a_h > g_h$. We use the following equations for the proof:

(1) *Overload Condition*: $a_h + a_l > r$
(2) *Work Conservation Condition*: $s_h + s_l = r$
(3) *WFQ Condition*: $g_h + g_l = r$
(4) *Guaranteed-bandwidth Condition*: If $a_i \geq g_i$, then $s_i \geq g_i$
(5) *Excess-bandwidth Condition*: If $s_h > g_h$, then $(a_h > g_h)$ and $(a_l < g_l)$

Consider the two possible scenarios depending on whether the arrival rate on QoS_l is below or above its guaranteed share.

Case 1: $a_l \leq g_l$ In this case, all of QoS_l traffic gets instantaneously served, i.e. $s_l = a_l$. Above equations give us: $a_h > r - a_l > r - g_l = g_h$.

Case 2: $a_l \geq g_l$ In this case, QoS_l at least gets its guaranteed rate, i.e., $s_l \geq g_l$. However, since there is no excess in QoS_h (as $a_h > s_h$), $s_l = g_l$, therefore $a_h > s_h = r - s_l = r - g_l = g_h$.

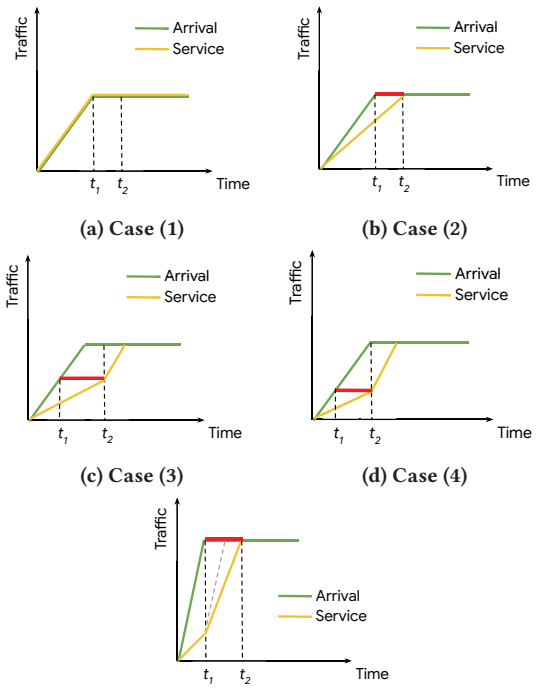
The same proof applies to class l due to symmetry.

B.2 Derivation of Delay Equations in 2-QoS WFQ

We first demonstrate how we apply Network Calculus [17] to find closed-form equations for worst-case delay in WFQ for the 2-QoS case.

Network Calculus

Similar to previous work [47], our analysis is based on Network Calculus. Network Calculus furnishes a simple result:



(a) Case (1) (b) Case (2)

(c) Case (3) (d) Case (4)

(e) Case (5)

Figure 27: Service curve changes as QoS-share of QoS_h changes.

given an arrival cumulative curve as per the traffic arrival pattern and a service curve defining how traffic is served in the system, then the maximum horizontal distance between the arrival and the service curves gives the theoretical delay bound of the queuing system.

Toy Example

Consider a simple example with a single bottleneck with 2 QoS levels, QoS_h and QoS_l, under the same traffic pattern described in §4.1 with a weight ratio of 4:1, guaranteed rate for QoS_h is 80Gbps and is 20Gbps for QoS_l. The switch has a processing rate of 100 Gbps. The total Traffic is split into 50%/50% on QoS_h/QoS_l. Traffic arrives at a burst of 120Gbps for some time (>100Gbps), and then stays idle to achieve an average of 80% load (=80Gbps). Since 50% of the traffic is on QoS_h with a burst load of 120%, the switch immediately processes all QoS_h traffic incoming at a rate of 60Gbps (as its lower than the guaranteed rate of 80Gbps), and the remaining 40 Gbps goes to QoS_l, resulting in QoS_l's queue size growing at 20Gbps. Once all QoS_h traffic has been processed, the link is now able to process QoS_l outstanding traffic at 100Gbsp and QoS_l's queue stops growing. In other words, at this moment QoS_l experiences the maximum delay at this moment – when all of QoS_h traffic has been processed. Figure 26b summarizes the service curve for QoS_l traffic, and the maximum horizontal distance between these 2 curves gives the delay bound (shown as the red bar in Figure 26c), which is represented as a fraction of the data sending period (including the burst and idle stage).

Closed-form Equations for 2-QoS case

With this explanation, we can start providing derivation of worst-case delay for QoS_h, $Delay_h$ in Equation 1, under the model depicted in Figure 7. The central idea is to determine the different service curves as QoS-mix changes, a and formulate equations for the delay bound based on the maximum horizontal distance between the arrival and the service curve.

The following notation in addition to Table 1 will be frequently used in the derivation.

$$a_h = \rho r x$$
$$a_l = \rho r (1 - x)$$
$$g_h = \frac{\phi}{\phi + 1} r$$
$$g_l = \frac{1}{\phi + 1} r$$

where x is the QoS_h-share, and the ratio of QoS weights for QoS_h: QoS_l is $\phi : 1$.

As the value of QoS_h-share (x in the equations) varies from 0 to 1, we enter different domains with different service curves and we explain the cases below.

Case (1) When QoS_h-share just starts to increase from 0, QoS_h's instantaneous input arrival rate can be immediately processed by its minimum guaranteed rate, and thus, QoS_h experiences no delay, i.e., $Delay_h(x) = 0$. In other words, the arrival and service curves overlap under this case, $s_h = a_h$ (as depicted in Figure 27a). The following conditions characterize this domain:

$$a_h \leq g_h$$
$$\implies x \leq \frac{\phi}{\phi + 1} \frac{1}{\rho}$$

Case (2) As QoS_h-share continues to increase, QoS_h starts to experience delay as well but it still finishes earlier than QoS_l. Therefore, priority inversion will not happen yet and we are still in the admissible region. This domain implies:

$$\begin{cases} a_h > g_h \\ a_l > g_l \\ \frac{\mu x}{g_h} < \frac{\mu(1-x)}{g_l} \end{cases}$$
$$\implies \frac{\phi}{\phi + 1} \frac{1}{\rho} < x \leq \frac{\phi}{\phi + 1}$$

We draw the arrival and service curve of QoS_h for Case (2) in Figure 27b. The delay bound can be achieved by taking the maximum horizontal distance between the two curves, which is the difference between t_1 and t_2 in the figure. In this

case, t_1 is as per the model in Figure 7, and t_2 is calculated as the time it takes to consume QoS_h's incoming traffic with $s_h = g_h$.

$$t_1 = \frac{\mu}{\rho}$$
$$t_2 = \frac{\mu r x}{g_h} = \mu x \frac{\phi + 1}{\phi}$$
$$Delay_h(x) = t_2 - t_1 = \mu(\frac{\phi + 1}{\phi} x - \frac{1}{\rho})$$

Case (3) QoS_h delay keeps growing as QoS_h-share increases, and eventually QoS_h finishes processing all the incoming traffic later than QoS_l, causing priority inversion. To solve for the domain in this case, we have:

$$\begin{cases} a_h > g_h \\ a_l > g_l \\ \frac{\mu x}{g_h} >= \frac{\mu(1-x)}{g_l} \end{cases}$$
$$\implies \frac{\phi}{\phi + 1} < x \leq 1 - \frac{1}{\phi + 1}\frac{1}{\rho}$$

The arrival and service curve of QoS_h in Case (3) is depicted in Figure 27c. Similar to the last case, the delay is calculated by the difference of t_1 and t_2. By definition, t_2 in this case is the time QoS_l finishes processing its input traffic using g_l in the current period. In the interval $[0, t_2]$, $s_h = g_h$. One can find t_1 by matching the y-value of the arrival and the service curves in Figure 27c as:

$$a_h t_1 = g_h t_2$$
$$t_2 = \frac{\mu r(1-x)}{g_l} = \mu(1-x)(\phi + 1)$$
$$t_1 = \frac{g_h t_2}{a_h} = \frac{\mu(1-x)\phi}{\rho x}$$
$$Delay_h(x) = t_2 - t_1 = \mu(1-x)(\phi + 1 - \frac{\phi}{\rho\, x})$$

Case (4) As QoS_l-share keeps decreasing with increasing x, QoS_l eventually experiences no delay, however QoS_h continues to experience delay. This is because we are in the overload situation where $\rho > 1$, there exists at least one QoS level that must experience delay. This implies:

$$a_l < g_l$$
$$\implies x > 1 - \frac{1}{\phi + 1}\frac{1}{\rho}$$

Figure 27d describes the arrival and service curve of QoS_h in this case. t_2 marks the time when QoS_l finishes servicing its traffic. t_1 can be calculated in a similar way as Case (3) by matching the y-values:

$$t_2 = \frac{\mu}{\rho}$$
$$t_1 = \frac{(r - a_l)t_2}{a_h} = \frac{\mu}{\rho^2}\frac{1 - \rho(1-x)}{x}$$
$$Delay_h(x) = \mu(\frac{1}{\rho} - \frac{1}{\rho^2})\frac{1}{x}$$

Case (5) As QoS_h-share keeps increasing, its arriving rate eventually exceeds the line rate. This creates the final arrival and service curve profile as shown in Figure 27e. The domain in this case is represented by:

$$a_h > r$$
$$\implies x > \frac{1}{\rho}$$

Note that t_1 in Figure 27e is when QoS_l finishes. There are multiple ways to obtain the delay bound in this case. A simple one is to recognize that t_2 is the time it takes to consume all the incoming traffic. To solve for the delay bound, we have:

$$t_1 = \frac{\mu}{\rho}$$
$$t_2 = \frac{\mu r}{r} = \mu$$
$$Delay_h(x) = t_2 - t_1 = \mu(1 - \frac{1}{\rho})$$

In summary, the five cases above have the following delay characteristics:

(1) QoS_h has no delay but QoS_l has delay
(2) both have delay but QoS_h finishes earlier
(3) both have delay but QoS_l finishes earlier
(4) QoS_h has delay but QoS_l does not
(5) QoS_h's arrival rate is above the link rate

By combining the above 5 cases, QoS_h worst-case delay ($Delay_h(x)$) (Equation 1) is obtained as:

$$\begin{cases} 0, & x \leq \frac{\phi}{\phi+1}\frac{1}{\rho} \\ \mu(\frac{\phi+1}{\phi}x - \frac{1}{\rho}), & \frac{\phi}{\phi+1}\frac{1}{\rho} < x \leq \frac{\phi}{\phi+1} \\ \mu(1-x)(\phi + 1 - \frac{\phi}{\rho\, x}), & \frac{\phi}{\phi+1} < x \leq \min\{1 - \frac{1}{\phi+1}\frac{1}{\rho}, \frac{1}{\rho}\} \\ \mu(\frac{1}{\rho} - \frac{1}{\rho^2})\frac{1}{x}, & \min\{1 - \frac{1}{\phi+1}\frac{1}{\rho}, \frac{1}{\rho}\} < x \leq \frac{1}{\rho} \\ \mu(1 - \frac{1}{\rho}), & x > \max\{\frac{\phi}{\phi+1}, \frac{1}{\rho}\} \end{cases}$$

Note that depending on the value of ρ and ϕ, it is possible that some of the domains are empty, however the equations remain valid with different domain boundaries.

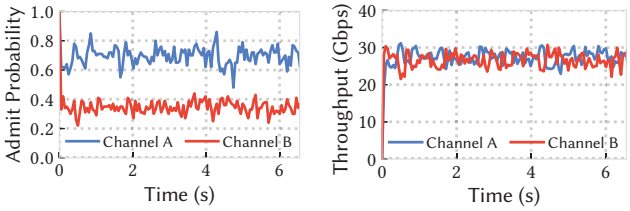


Figure 28: Same experiment as in Figure 17 with smaller β value ($\beta = 0.0015$).

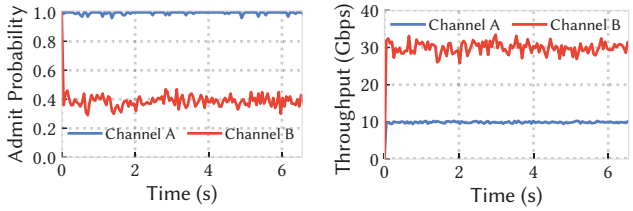


Figure 29: Same experiment as in Figure 18 with smaller β value ($\beta = 0.0015$).

For example, when $\phi = 4$, $\rho = 2$ and $\mu = 0.8$, QoS_h worst-case delay becomes:

$$Delay_h(x) = \begin{cases} 0, & x \leq 0.4 \\ x - 0.4, & 0.4 < x \leq 0.8 \\ 0.4, & x > 0.8 \end{cases}$$

Following a similar analysis, the closed-form worst-case delay for QoS_l ($Delay_l(x)$) is given by:

$$\begin{cases} \mu(1-\frac{1}{\rho}), & x \leq \min\{1-\frac{1}{\rho}, \frac{\phi}{\phi+1}\} \\ \mu(\frac{1}{\rho}-\frac{1}{\rho^2})\frac{1}{(1-x)}, & 1-\frac{1}{\rho} < x \leq \max\{\frac{\phi}{\phi+1}\frac{1}{\rho}, 1-\frac{1}{\rho}\} \\ \mu\frac{x}{\phi}(\phi+1-\frac{1}{\rho(1-x)}), & \max\{\frac{\phi}{\phi+1}\frac{1}{\rho}, 1-\frac{1}{\rho}\} < x \leq \frac{\phi}{\phi+1} \\ \mu((\phi+1)(1-x)-\frac{1}{\rho}), & \frac{\phi}{\phi+1} < x \leq 1-\frac{1}{\phi+1}\frac{1}{\rho} \\ 0, & x > 1-\frac{1}{\phi+1}\frac{1}{\rho} \end{cases} \quad (8)$$

C SENSITIVITY ANALYSIS

α and β are key parameters in that they posit a tradeoff between SLO-compliance and stability in achieved shares. Under adversarial patterns, Aequitas favors SLO-compliance over work-conservation by the virtue of being an admission-control system.

We repeat the experiment in §6.5 with a lower β value of 0.0015 per MTU (Algorithm 1) compared to the original setting of 0.01. A lower β value implies a smaller decrease in p_{admit} whenever a latency miss is detected. While this provides excellent stability around fair-shares, it is less suited towards SLO-compliance. We show the equivalent of Figure 17 with smaller β value in Figure 28, and the equivalent of Figure 18 in Figure 29. The 1^{st}-p p_{admit} for RPC Channel A in Figure 29 is 0.96, an improvement over 0.82 in Figure 18.

The parameter α has a similar tradeoff, a lower value is averse to increasing admit probability and favors SLO-compliance, whereas a higher value can provide better stability but with worse SLO-compliance.

D ARTIFACT APPENDIX

We provide a brief appendix of our artifact for interested readers who want to try out Aequitas. You can find more information about our artifact evaluation on the *artifact-eval* branch of our GitHub repository (https://github.com/SymbioticLab/Aequitas).

D.1 Abstract

This artifact is designed to reproduce major results in our paper using the simulator we wrote. It contains the source code of the simulator and the scripts used to launch experiments that reproduce our evaluation results.

D.2 Scope

The artifact is used to validate major evaluation results including theoretical 2-QoS worst-case delay analysis, SLO compliance, QoS-mix convergence, and fairness aspects of Aequitas. Readers are also encouraged to use the simulator or build on top of it to study other research problems in datacenter networking.

D.3 Contents

The artifact contains the following items:

(1) An README describing the artifact including how to build the simulator code, how to launch the selected experiments, and what expected results are after running those experiments.

(2) Source code of the simulator, which is used to simulate Aequitas and other systems in related work.

(3) Scripts to launch all the provided experiments.

The experiments include (a) a theoretical verification of the 2-QoS worst case delay, (b) evaluating SLO-compliance for both 2-QoS and 3-QoS cases, (c) evaluating QoS-mix convergence in a 3-QoS setting, and (d) evaluating fairness aspect in the same setting as the one we have in the paper. We also includes the configuration files for all the systems in our related work comparison. Users can try out the simulator with their own RPC size distribution.

D.4 Hosting

To obtain the artifact, go to our GitHub repository at https://github.com/SymbioticLab/Aequitas and switch to the *artifact-eval* branch with the latest commit.

D.5 Requirements

The artifact requires a build environment to compile the simulator, which is written in C++. We developed the simulator in Linux. The code should work as long as it can be compiled correctly with automake. The compilation is very straightforward.