Cloud Control with Distributed Rate Limiting

Barath Raghavan, Kashi Vishwanath, Sriram Ramabhadran,
Kenneth Yocum, and Alex C. Snoeren

Department of Computer Science and Engineering
University of California, San Diego

ABSTRACT

Today's cloud-based services integrate globally distributed resources into seamless computing platforms. Provisioning and accounting for the resource usage of these Internet-scale applications presents a challenging technical problem. This paper presents the design and implementation of distributed rate limiters, which work together to enforce a global rate limit across traffic aggregates at multiple sites, enabling the coordinated policing of a cloud-based service's network traffic. Our abstraction not only enforces a global limit, but also ensures that congestion-responsive transport-layer flows behave as if they traversed a single, shared limiter. We present two designs—one general purpose, and one optimized for TCP—that allow service operators to explicitly trade off between communication costs and system accuracy, efficiency, and scalability. Both designs are capable of rate limiting thousands of flows with negligible overhead (less than 3% in the tested configuration). We demonstrate that our TCP-centric design is scalable to hundreds of nodes while robust to both loss and communication delay, making it practical for deployment in nationwide service providers.

Categories and Subject Descriptors

C.2.3 [**Computer Communication Networks**]: Network management

General Terms

Algorithms, Management, Performance

Keywords

Rate Limiting, Token Bucket, CDN

1. INTRODUCTION

Yesterday's version of distributed computing was a self-contained, co-located server farm. Today, applications are increasingly deployed on third-party resources hosted across the Internet. Indeed, the rapid spread of open protocols and standards like Web 2.0 has fueled an explosion of compound services that script together third-party components to deliver a sophisticated service [27, 29]. These specialized services are just the beginning: flagship consumer and enterprise applications are increasingly being delivered in the software-as-a-service model [9]. For example, Google Documents, Groove Office, and Windows Live are early examples of desktop applications provided in a hosted environment, and represent the beginning of a much larger trend.

Aside from the functionality and management benefits Web-based services afford the end user, hosted platforms present significant benefits to the service provider as well. Rather than deploy each component of a multi-tiered application within a particular data center, so-called "cloud-based services" can transparently leverage widely distributed computing infrastructures. Google's service, for example, reportedly runs on nearly half-a-million servers distributed around the world [8]. Potential world-wide scale need not be limited to a few large corporations, however. Recent offerings like Amazon's Elastic Compute Cloud (EC2) promise to provide practically infinite resources to anyone willing to pay [3].

One of the key barriers to moving traditional applications to the cloud, however, is the loss of cost control [17]. In the cloud-based services model, cost recovery is typically accomplished through metered pricing. Indeed, Amazon's EC2 charges incrementally per gigabyte of traffic consumed [3]. Experience has shown, however, that ISP customers prefer flat fees to usage-based pricing [30]. Similarly, at a corporate level, IT expenditures are generally managed as fixed-cost overheads, not incremental expenses [17]. A flat-fee model requires the ability for a provider to limit consumption to control costs. Limiting global resource consumption in a distributed environment, however, presents a significant technical challenge. Ideally, resource providers would not require services to specify the resource demands of each distributed component *a priori*; such fine-grained measurement and modeling can be challenging for rapidly evolving services. Instead, they should provide a fixed price for an aggregate, global usage, and allow services to consume resources dynamically across various locations, subject to the specified aggregate limit.

In this paper, we focus on a specific instance of this problem: controlling the aggregate network bandwidth used by a cloud-based service, or distributed rate limiting (DRL). Our goal is to allow a set of distributed traffic rate limiters to collaborate to subject a class of network traffic (for example, the traffic of a particular cloud-based service) to a single, aggregate global limit. While traffic policing is common in data centers and widespread in today's networks, such limiters typically enforce policy independently at each location [1]. For example, a resource provider with 10 hosting centers may wish to limit the total amount of traffic it carries for a particular service to 100 Mbps. Its current options are to either limit the service to 100 Mbps at each hosting center (running the risk that they may all use this limit simultaneously, resulting in 1 Gbps of total traffic), or to limit each center to a fixed portion (i.e., 10 Mbps) which overconstrains the service traffic aggregate and is unlikely to allow the service to consume its allocated budget unless traffic is perfectly balanced across the cloud.

The key challenge of distributed rate limiting is to allow individual flows to compete dynamically for bandwidth not only with flows traversing the same limiter, but with flows traversing other limiters as well. Thus, flows arriving at different limiters should achieve the same rates as they would if they all were traversing a single, shared rate limiter. Fairness between flows inside a traffic aggregate depends critically on accurate limiter assignments, which in turn depend upon the local packet arrival rates, numbers of flows, and up/down-stream bottleneck capacities. We address this issue by presenting the illusion of passing all of the traffic through a single token-bucket rate limiter, allowing flows to compete against each other for bandwidth in the manner prescribed by the transport protocol(s) in use. For example, TCP flows in a traffic aggregate will share bandwidth in a flow-fair manner [6]. The key technical challenge to providing this abstraction is measuring the demand of the aggregate at each limiter, and apportioning capacity in proportion to that demand. This paper makes three primary contributions:

Rate Limiting Cloud-based Services. We identify a key challenge facing the practical deployment of cloud-based services and identify the chief engineering difficulties: how to effectively balance *accuracy* (how well the system bounds demand to the aggregate rate limit), *responsiveness* (how quickly limiters respond to varying traffic demands), and *communication* between the limiters. A distributed limiter cannot be simultaneously perfectly accurate and responsive; the communication latency between limiters bounds how quickly one limiter can adapt to changing demand at another.

Distributed Rate Limiter Design. We present the design and implementation of two distributed rate limiting algorithms. First, we consider an approach, *global random drop* (GRD), that approximates the number, but not the precise timing, of packet drops. Second, we observe that applications deployed using Web services will almost exclusively use TCP. By incorporating knowledge about TCP's congestion control behavior, we design another mechanism, *flow proportional share* (FPS), that provides improved scalability.

Evaluation and Methodology. We develop a methodology to evaluate distributed rate limiters under a variety of traffic demands and deployment scenarios using both a local-area testbed and an Internet testbed, PlanetLab. We demonstrate that both GRD and FPS exhibit long-term behavior similar to a centralized limiter for both mixed and homogeneous TCP traffic in low-loss environments. Furthermore, we show that FPS scales well, maintaining near-ideal 50-Mbps rate enforcement and fairness up to 490 limiters with a modest communication budget of 23 Kbps per limiter.

2. CLASSES OF CLOUDS

Cloud-based services come in varying degrees of complexity; as the constituent services become more numerous and dynamic, resource provisioning becomes more challenging. We observe that the distributed rate limiting problem arises in any service composed of geographically distributed sites. In this section we describe three increasingly mundane applications, each illustrating how DRL empowers service providers to enforce heretofore unrealizable traffic policies, and how it offers a new service model to customers.

2.1 Limiting cloud-based services

Cloud-based services promise a "utility" computing abstraction in which clients see a unified service and are unaware that the system stitches together independent physical sites to provide cycles, bandwidth, and storage for a uniform purpose. In this context, we are interested in rate-based resources that clients source from a single provider across many sites or hosting centers.

For clouds, distributed rate limiting provides the critical ability for resource providers to control the use of network bandwith as if it were all sourced from a single site. A provider runs DRL across its sites, setting global limits on the total network usage of particular traffic classes or clients. Providers are no longer required to migrate requests to accomodate static bandwidth limits. Instead, the available bandwidth gravitates towards the sites with the most demand. Alternatively, clients may deploy DRL to control aggregate usage across providers as they see fit. DRL removes the artificial separation of access metering and geography that results in excess cost for the client and/or wasted resources for service providers.

2.2 Content distribution networks

While full-scale cloud-based computing is in its infancy, simple cloud-based services such as content-distribution networks (CDNs) are prevalent today and can benefit from distributed rate limiting. CDNs such as Akamai provide content replication services to third-party Web sites. By serving Web content from numerous geographically diverse locations, CDNs improve the performance, scalability, and reliability of Web sites. In many scenarios, CDN operators may wish to limit resource usage either based on the content served or the requesting identity. Independent rate limiters are insufficient, however, as content can be served from any of numerous mirrors around the world according to fluctuating demand.

Using DRL, a content distribution network can set per-customer limits based upon service-level agreements. The CDN provides service to all sites as before, but simply applies DRL to all out-bound traffic for each site. In this way, the bandwidth consumed by a customer is constrained, as is the budget required to fund it, avoiding the need for CDNs to remove content for customers who cannot pay for their popularity.[1] Alternatively, the CDN can use DRL as a protective mechanism. For instance, the CoDeeN content distribution network was forced to deploy an ad-hoc approach to rate limit nefarious users across proxies [37]. DRL makes it simple to limit the damage on the CDN due to such behavior by rate limiting traffic from an identified set of users. In summary, DRL provides CDNs with a powerful tool for managing access to their clients' content.

2.3 Internet testbeds

Planetlab supports the deployment of long-lived service prototypes. Each Planetlab service runs in a *slice*—essentially a fraction of the entire global testbed consisting of $1/N$ of each machine. Currently Planetlab provides work-conserving bandwidth limits at each individual site, but the system cannot coordinate bandwidth demands across multiple machines [18].

DRL dynamically apportions bandwidth based upon demand, allowing Planetlab administrators to set bandwidth limits on a per-slice granularity, independent of which nodes a slice uses. In the context of a single Planetlab service, the service administrator may limit service to a particular user. In Section 5.7 we show that DRL provides effective limits for a Planetlab service distributed across North America. In addition, while we focus upon network rate limiting in this paper, we have begun to apply our techniques to control other important rate-based resources such as CPU.

[1]For example, Akamai customers are typically not rate limited and billed in arrears for actual aggregate usage, leaving them open to potentially large bills. If demand dramatically exceeds expectation and/or their willingness to pay, manual intervention is required [2].

2.4 Assumptions and scope

Like centralized rate limiting mechanisms, distributed rate limiting does not provide QoS guarantees. Thus, when customers pay for a given level of service, providers must ensure the availability of adequate resources for the customer to attain its given limit. In the extreme, a provider may need to provision each limiter with enough capacity to support a service's entire aggregate limit. Nevertheless, we expect many deployments to enjoy considerable benefits from statistical multiplexing. Determining the most effective provisioning strategy, however, is outside the scope of this paper.

Furthermore, we assume that mechanisms are already in place to quickly and easily identify traffic belonging to a particular service [25]. In many cases such facilities, such as simple address or protocol-based classifiers, already exist and can be readily adopted for use in a distributed rate limiter. In others, we can leverage recent work on network capabilities [32, 39] to provide unforgeable means of attribution. Finally, without loss of generality, we discuss our solutions in the context of a single service; multiple services can be limited in the same fashion.

3. LIMITER DESIGN

We are concerned with coordinating a set of topologically distributed limiters to enforce an aggregate traffic limit while retaining the behavior of a centralized limiter. That is, a receiver should not be able to tell whether the rate limit is enforced at one or many locations simultaneously. Specifically, we aim to approximate a centralized token-bucket traffic-policing mechanism. We choose a token bucket as a reference mechanism for a number of reasons: it is simple, reasonably well understood, and commonly deployed in Internet routers. Most importantly, it makes instantaneous decisions about a packet's fate—packets are either forwarded or dropped—and so does not introduce any additional queuing.

We do not assume anything about the distribution of traffic across limiters. Thus, traffic may arrive at any or all of the limiters at any time. We use a peer-to-peer limiter architecture: each limiter is functionally identical and operates independently. The task of a limiter can be split into three separate subtasks: estimation, communication, and allocation. Every limiter collects periodic measurements of the local traffic arrival rate and disseminates them to the other limiters. Upon receipt of updates from other limiters, each limiter computes its own estimate of the global aggregate arrival rate that it uses to determine how to best service its local demand to enforce the global rate limit.

3.1 Estimation and communication

We measure traffic demand in terms of bytes per unit time. Each limiter maintains an estimate of both local and global demand. Estimating local arrival rates is well-studied [15, 34]; we employ a strategy that computes the average arrival rate over fixed time intervals and applies a standard exponentially-weighted moving average (EWMA) filter to these rates to smooth out short-term fluctuations. The estimate interval length and EWMA smoothing parameter directly affect the ability of a limiter to quickly track and communicate local rate changes; we determine appropriate settings in Section 5.

At the end of each estimate interval, local changes are merged with the current global estimate. In addition, each limiter must disseminate changes in local arrival rate to the other limiters. The simplest form of communication fabric is a broadcast mesh. While fast and robust, a full mesh is also extremely bandwidth-intensive (requiring $O(N^2)$ update messages per estimate interval). Instead, we implement a gossip protocol inspired by Kempe *et al.* [22]. Such "epidemic" protocols have been widely studied for distributed coordination; they require little to no communication structure, and are robust to link and node failures [10]. At the end of each estimate interval, limiters select a fixed number of randomly chosen limiters to update; limiters use any received updates to update their global demand estimates. The number of limiters contacted—the gossip branching factor—is a parameter of the system. We communicate updates via a UDP-based protocol that is resilient to loss and reordering; for now we ignore failures in the communication fabric and revisit the issue in Section 5.6. Each update packet is 48 bytes, including IP and UDP headers. More sophisticated communication fabrics may reduce coordination costs using structured approaches [16]; we defer an investigation to future work.

```
GRD-HANDLE-PACKET($P$: Packet)
1  demand ← $\sum_i^n r_i$
2  bytecount ← bytecount+LENGTH($P$)
3  **if** demand > limit **then**
4      dropprob ← (demand − limit) / demand
5      **if** RAND() < dropprob **then**
6          DROP($P$)
7          **return**
8  FORWARD($P$)
```

Figure 1: Pseudocode for GRD. Each value r_i corresponds to the current estimate of the rate at limiter i.

3.2 Allocation

Having addressed estimation and communication mechanisms, we now consider how each limiter can combine local measurements with global estimates to determine an appropriate local limit to enforce. A natural approach is to build a global token bucket (GTB) limiter that emulates the fine-grained behavior of a centralized token bucket. Recall that arriving bytes require tokens to be allowed passage; if there are insufficient tokens, the token bucket drops packets. The rate at which the bucket regenerates tokens dictates the traffic limit. In GTB, each limiter maintains its own global estimate and uses reported arrival demands at other limiters to estimate the rate of drain of tokens due to competing traffic.

Specifically, each limiter's token bucket refreshes tokens at the global rate limit, but removes tokens both when bytes arrive locally and to account for expected arrivals at other limiters. At the end of every estimate interval, each limiter computes its local arrival rate and sends this value to other limiters via the communication fabric. Each limiter sums the most recent values it has received for the other limiters and removes tokens from its own bucket at this "global" rate until a new update arrives. As shown in Section 4, however, GTB is highly sensitive to stale observations that continue to remove tokens at an outdated rate, making it impractical to implement at large scale or in lossy networks.

3.2.1 *Global random drop*

Instead of emulating the precise behavior of a centralized token bucket, we observe that one may instead emulate the higher-order behavior of a central limiter. For example, we can ensure the rate of drops over some period of time is the same as in the centralized case, as opposed to capturing the burstiness of packet drops—in this way, we emulate the rate enforcement of a token bucket but not its burst limiting. Figure 1 presents the pseudocode for a global random drop (GRD) limiter that takes this approach. Like GTB, GRD monitors the aggregate global input demand, but uses it to calculate a packet drop probability. GRD drops packets with a probability proportional to the excess global traffic demand in the previous interval (line 4). Thus, the number of drops is expected to be the same as in a single token bucket; the aggregate forwarding rate should be no greater than the global limit.

GRD somewhat resembles RED queuing in that it increases its drop probability as the input demand exceeds some threshold [14]. Because there are no queues in our limiter, however, GRD requires no tuning parameters of its own (besides the estimator's EWMA and estimate interval length). In contrast to GTB, which attempts to reproduce the packet-level behavior of a centralized limiter, GRD tries to achieve accuracy by reproducing the number of losses over longer periods of time. It does not, however, capture short-term effects. For inherently bursty protocols like TCP, we can improve short-term fairness and responsiveness by exploiting information about the protocol's congestion control behavior.

3.2.2 *Flow proportional share*

One of the key properties of a centralized token bucket is that it retains inter-flow fairness inherent to transport protocols such as TCP. Given the prevalence of TCP in the Internet, and especially in modern cloud-based services, we design a flow proportional share (FPS) limiter that uses domain-specific knowledge about TCP to emulate a centralized limiter without maintaining detailed packet arrival rates. Each FPS limiter uses a token bucket for rate limiting—thus, each limiter has a *local* rate limit. Unlike GTB, which renews tokens at the global rate, FPS dynamically adjusts its local rate limit in proportion to a set of weights computed every estimate interval. These weights are based upon the number of live flows at each limiter and serve as a proxy for demand; the weights are then used to enforce max-min fairness between congestion-responsive flows [6].

The primary challenge in FPS is estimating TCP demand. In the previous designs, each rate limiter estimates demand by measuring packets' sizes and the rate at which it receives them; this accurately reflects the byte-level demand of the traffic sources. In contrast, FPS must determine demand in terms of the number of TCP flows present, which is independent of arrival rate. Furthermore, since TCP always attempts to increase its rate, a single flow consuming all of a limiter's rate is nearly indistinguishable from 10 flows doing the same.[2] However, we would like that a 10-flow aggregate receive 10 times the weight of a single flow.

Our approach to demand estimation in FPS is shown in Figure 2. Flow aggregates are in one of two states. If the aggregate under-utilizes the allotted rate (local limit) at a limiter, then all constituent flows must be *bottlenecked*. In other words, the flows are all constrained elsewhere. On the other hand, if the aggregate either meets or exceeds the local limit, we say that one or more of the constituent flows is *unbottlenecked*—for these flows the limiter is the bottleneck. We calculate flow weights with the function FPS-ESTIMATE. If flows were max-min fair, then each unbottlenecked flow would receive approximately the same rate. We therefore count a weight of 1 for every unbottlenecked flow at every limiter. Thus, if all flows were unbottlenecked, then the demand at each limiter is directly proportional to its current flow count. Setting the local weight to this number results in max-min fair allocations. We use the computed weight on line 10 of FPS-ESTIMATE to proportionally set the local rate limit.

[2]There is a slight difference between these scenarios: larger flow aggregates have smaller demand oscillations when desynchronized [4]. Since TCP is periodic, we considered distinguishing TCP flow aggregates based upon the component frequencies in the aggregate via the FFT. However, we found that the signal produced by TCP demands is not sufficiently stationary.

```
FPS-ESTIMATE()
1   for each flow $f$ in sample set
2       ESTIMATE($f$)
3   localdemand $\leftarrow r_i$
4   if localdemand $\geq$ locallimit then
5       maxflowrate $\leftarrow$ MAXRATE(sample set)
6       idealweight $\leftarrow$ locallimit / maxflowrate
7   else
8       remoteweights $\leftarrow \sum_{j \neq i}^{n} w_j$
9       idealweight $\leftarrow \frac{\text{localdemand} \cdot \text{remoteweights}}{\text{limit} - \text{localdemand}}$
10  locallimit $\leftarrow \frac{\text{idealweight} \cdot \text{limit}}{\text{remoteweights} + \text{idealweight}}$
11  PROPAGATE(idealweight)

FPS-HANDLE-PACKET($P$: Packet)
1   if RAND() < resampleprob then
2       add FLOW($P$) to sample set
3   TOKEN-BUCKET-LIMIT($P$)
```

Figure 2: Pseudocode for FPS. w_i corresponds to the weight at each limiter i that represents the normalized flow count (as opposed to rates r_i as in GRD).

A seemingly natural approach to weight computation is to count TCP flows at each limiter. However, flow counting fails to account for the demands of TCP flows that are bottlenecked: 10 bottlenecked flows that share a modem do not exert the same demands upon a limiter as a single flow on an OC-3. Thus, FPS must compute the equivalent number of unbottlenecked TCP flows that an aggregate demand represents. Our primary insight is that we can use TCP itself to estimate demand: in an aggregate of TCP flows, each flow will eventually converge to its fair-share transmission rate. This approach leads to the first of two operating regimes:

Local arrival rate $\geq$ local rate limit. When there is at least one unbottlenecked flow at the limiter, the aggregate input rate is equal to (or slightly more than) the local rate limit. In this case, we compute the weight by dividing the local rate limit by the sending rate of an unbottlenecked flow, as shown on lines 5 and 6 of FPS-ESTIMATE. Intuitively, this allows us to use a TCP flow's knowledge of congestion to determine the amount of competing demand. In particular, if all the flows at the provider are unbottlenecked, this yields a flow count without actual flow counting.

Thus, to compute the weight, a limiter must estimate an unbottlenecked flow rate. We can avoid per-flow state by sampling packets at a limiter and maintaining byte counters for a constant-size flow set. We assume that the flow with the maximum sending rate is unbottlenecked. However, it is possible that our sample set will contain only bottlenecked flows. Thus, we continuously resample and discard small flows from our set, thereby ensuring that the sample set contains an unbottlenecked flow. It is likely that we will select an unbottlenecked flow in the long run for two reasons. First, since we uniformly sample packets, an unbottlenecked flow is more likely to be picked than a bottlenecked flow. Second, a sample set that contains only bottlenecked flows results in the weight being overestimated, which increases the local rate limit, causes unbottlenecked flows to grow, and makes them more likely to be chosen subsequently.

To account for bottlenecked flows, FPS implicitly normalizes the weight by scaling down the contribution of such flows proportional to their sending rates. A bottlenecked flow only contributes a fraction equal to its sending rate divided by that of an unbottlenecked flow. For example, if a bottlenecked flow sends at 10 Kbps, and the fair share of an unbottlenecked flow is 20 Kbps, the bottlenecked flow counts for half the weight of an unbottlenecked flow.

Local arrival rate $<$ local rate limit. When all flows at the limiter are bottlenecked, there is no unbottlenecked flow whose rate can be used to compute the weight. Since the flow aggregate is unable to use all the rate available at the limiter, we compute a weight that, based on current information, sets the local rate limit to be equal to the local demand (line 9 of FPS-ESTIMATE).

A limiter may oscillate between the two regimes: entering the second typically returns the system to the first, since the aggregate may become unbottlenecked due to the change in local rate limit. As a result, the local rate limit is increased during the next allocation, and the cycle repeats. We note that this oscillation is necessary to allow bottlenecked flows to become unbottlenecked should additional capacity become available elsewhere in the network; like the estimator, we apply an EWMA to smooth this oscillation. We have proved that FPS is stable—given stable input demands, FPS remains at the correct allocation of weights among limiters once it arrives in that state. (We include the proof in the Appendix.) It remains an open question, however, whether FPS converges under all conditions, and if so, how quickly.

Finally, TCP's slow start behavior complicates demand estimation. Consider the arrival of a flow at a limiter that has a current rate limit of zero. Without buffering, the flow's SYN will be lost and the flow cannot establish its demand. Thus, we allow bursting of the token bucket when the local rate limit is zero to allow a TCP flow in slow start to send a few packets before losses occur. When the allocator detects nonzero input demand, it treats the demand as a bottlenecked flow for the first estimate interval. As a result, FPS allocates rate to the flow equivalent to its instantaneous rate during the beginning of slow start, thus allowing it to continue to grow.

4. EVALUATION METHODOLOGY

Our notion of a good distributed rate limiter is one that accurately replicates centralized limiter behavior. Traffic policing mechanisms can affect packets and flows on several time scales; particularly, we can aim to emulate packet-level behavior or flow-level behavior. However, packet-level behavior is non-intuitive, since applications typically operate at the flow level. Even in a single limiter, any one measure of packet-level behavior fluctuates due to randomness in the physical system, though transport-layer flows may achieve the same relative fairness and throughput. This implies a weaker, but tractable goal of functionally equivalent behavior. To this end, we measure limiter performance using aggregate metrics over real transport-layer protocols.

4.1 Metrics

We study three metrics to determine the fidelity of limiter designs: utilization, flow fairness, and responsiveness. The basic goal of a distributed rate limiter is to hold aggregate throughput across all limiters below a specified global limit. To establish fidelity we need to consider utilization over different time scales. Achievable throughput in the centralized case depends critically on the traffic mix. Different flow arrivals, durations, round trip times, and protocols imply that aggregate throughput will vary on many time scales. For example, TCP's burstiness causes its instantaneous throughput over small time scales to vary greatly. A limiter's long-term behavior may yield equivalent aggregate throughput, but may burst on short time scales. Note that, since our limiters do not queue packets, some short-term excess is unavoidable to maintain long-term throughput. Particularly, we aim to achieve fairness equal to or better than that of a centralized token bucket limiter.

Fairness describes the distribution of rate across flows. We employ Jain's fairness index to quantify the fairness across a flow set [20]. The index considers k flows where the throughput of flow i is x_i. The fairness index f is between 0 and 1, where 1 is completely fair (all flows share bandwidth equally):

$$f = \frac{\left(\sum_{i=1}^{k} x_i\right)^2}{k\left(\sum_{i=1}^{k} x_i^2\right)}$$

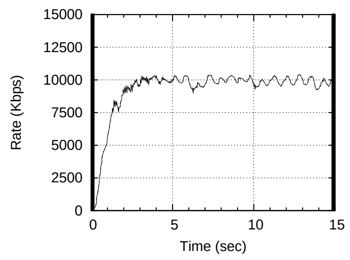
We must be careful when using this metric to ascertain flow-level fidelity. Consider a set of identical TCP flows traversing a single limiter. Between runs, the fairness index will show considerable variation; establishing the flow-level behavior for one or more limiters requires us to measure the distribution of the index across multiple experiments. Additional care must be taken when measuring Jain's index across multiple limiters. Though the index approaches 1 as flows receive their fair share, skewed throughput distributions can yield seemingly high indices. For example, consider 10 flows where 9 achieve similar throughput while 1 gets nothing; this results in the seemingly high fairness index of 0.9. If we consider the distribution of flows across limiters—the 9 flows go through one limiter and the 1 flow through another—the fairness index does not capture the poor behavior of the algorithm. Nevertheless, such a metric is necessary to help establish the flow-level behavior of our limiters, and therefore we use it as a standard measure of fairness with the above caveat. We point out discrepancies when they arise.

4.2 Implementation

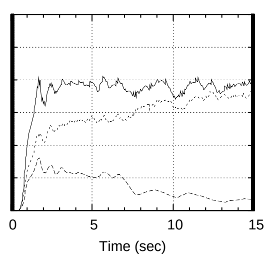
To perform rate limiting on real flows without proxying, we use user-space queuing in `iptables` on Linux to capture full IP packets and pass them to the designated rate limiter without allowing them to proceed through kernel packet processing. Each rate limiter either drops the packet or forwards it on to the destination through a raw socket. We use similar, but more restricted functionality for VNET raw sockets in PlanetLab to capture and transmit full IP packets. Rate limiters communicate with each other via UDP. Each gossip message sent over the communication fabric contains a sequence number in addition to rate updates; the receiving limiter uses the sequence number to determine if an update is lost, and if so, compensates by scaling the value and weight of the newest update by the number of lost packets. Note that all of our experiments rate limit traffic in one direction; limiters forward returning TCP ACKs irrespective of any rate limits.

4.3 Evaluation framework

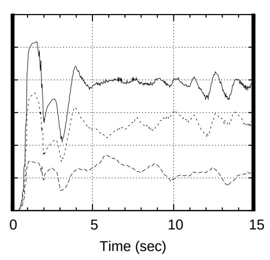
We evaluate our limiters primarily on a local-area emulation testbed using ModelNet [35], which we use only to emulate link latencies. A ModelNet emulation tests real, deployable prototypes over unmodified, commodity operating systems and network stacks, while providing a level of repeatability unachievable in an Internet experiment. Running our experiments in a controlled environment helps us gain intuition, ensures that transient network congestion, failures, and unknown intermediate bottleneck links do not confuse our results, and allows direct comparison across experiments. We run the rate limiters, traffic sources, and traffic sinks on separate endpoints in our ModelNet network topology. All source, sink, and rate limiter machines run Linux 2.6.9. TCP sources use New Reno with SACK enabled. We use a simple mesh topology to connect limiters and route each source and sink pair through a single limiter. The virtual topology connects all nodes using 100-Mbps links.



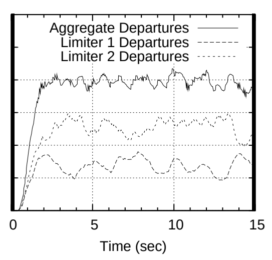
(a) Centralized token bucket.



(b) Global token bucket.

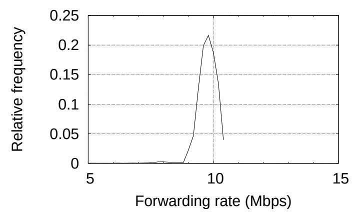


(c) Global random drop.

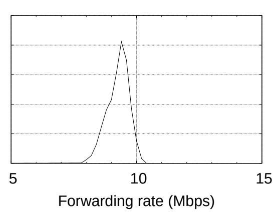


(d) Flow proportional share.

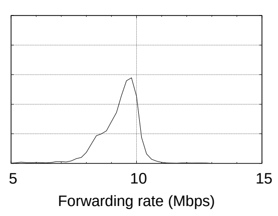
Figure 3: Time series of forwarding rate for a centralized limiter and our three limiting algorithms in the baseline experiment—3 TCP flows traverse limiter 1 and 7 TCP flows traverse limiter 2.



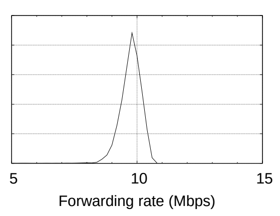
(a) CTB 1 sec.



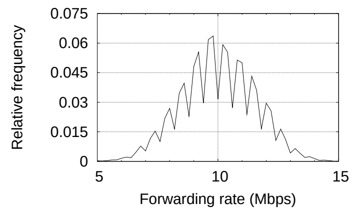
(b) GTB 1 sec.



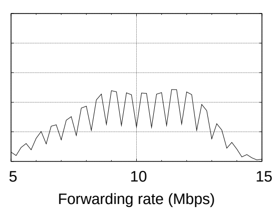
(c) GRD 1 sec.



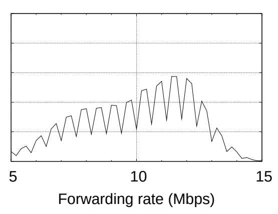
(d) FPS 1 sec.



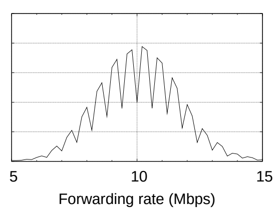
(e) CTB 0.1 sec.



(f) GTB 0.1 sec.



(g) GRD 0.1 sec.



(h) FPS 0.1 sec.

Figure 4: Delivered forwarding rate for the aggregate at different time scales—each row represents one run of the baseline experiment across two limiters with the "instantaneous" forwarding rate computed over the stated time period.

5. EVALUATION

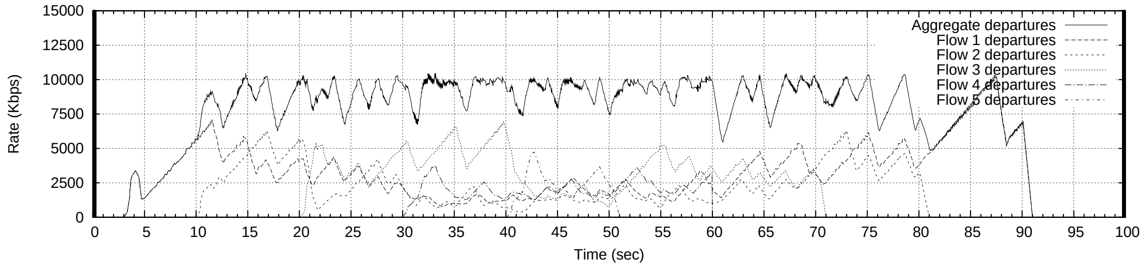
Our evaluation has two goals. The first is to establish the ability of our algorithms to reproduce the behavior of a single limiter in meeting the global limit and delivering flow-level fairness. These experiments use only 2 limiters and a set of homogeneous TCP flows. Next we relax this idealized workload to establish fidelity in more realistic settings. These experiments help achieve our second goal: to determine the effective operating regimes for each design. For each system we consider responsiveness, performance across various traffic compositions, and scaling, and vary the distribution of flows across limiters, flow start times, protocol mix, and traffic characteristics. Finally, as a proof of concept, we deploy our limiters across PlanetLab to control a mock-up of a simple cloud-based service.

5.1 Baseline

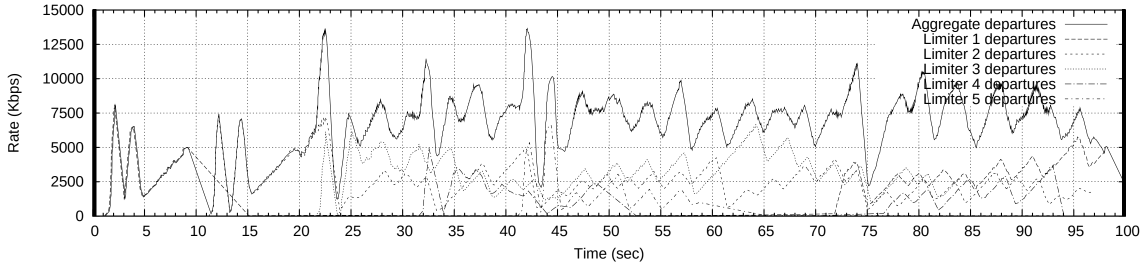
The baseline experiment consists of two limiters configured to enforce a 10-Mbps global limit. We load the limiters with 10 unbottlenecked TCP flows; 3 flows arrive at one limiter while 7 arrive at the other. We choose a 3-to-7 flow skew to avoid scenarios that would result in apparent fairness even if the algorithm fails. The reference point is a centralized token-bucket limiter (CTB) servicing all 10 flows. We fix flow and inter-limiter round trip times (RTTs) to 40 ms, and token bucket depth to 75,000 bytes—slightly greater than the bandwidth-delay product, and, for now, use a loss-free communication fabric. Each experiment lasts 60 seconds (enough time for TCP to stabilize), the estimate interval is 50 ms, and the 1-second EWMA parameter is 0.1; we consider alternative values in the next section.

Figure 3 plots the packet forwarding rate at each limiter as well as the achieved throughput of the flow aggregate. In all cases, the aggregate utilization is approximately 10 Mbps. We look at smaller time scales to determine the extent to which the limit is enforced. Figure 4 shows histograms of delivered "instantaneous" forwarding rates computed over two different time periods, thus showing whether a limiter is bursty or consistent in its limiting. All designs deliver the global limit over 1-second intervals; both GTB and GRD, however, are bursty in the short term. By contrast, FPS closely matches the rates of CTB at both time scales. We believe this is because FPS uses a token bucket to enforce local limits. It appears that when enforcing the same aggregate limit, the forwarding rate of multiple token buckets approximates that of a single token bucket even at short time scales.

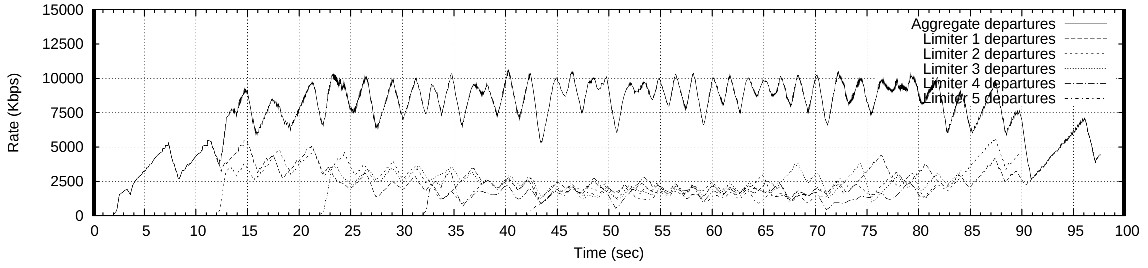
Returning to Figure 3, the aggregate forwarding rate should be apportioned between limiters in about a 3-to-7 split. GTB clearly fails to deliver in this regard, but both GRD and FPS appear approximately correct upon visual inspection. We use Jain's fairness index to quantify the fairness of the allocation. For each run of an experiment, we compute one fairness value across all flows, irrespective of the limiter at which they arrive. Repeating this experiment 10 times yields a distribution of fairness values. We use quantile-quantile plots to compare the fairness distribution of each



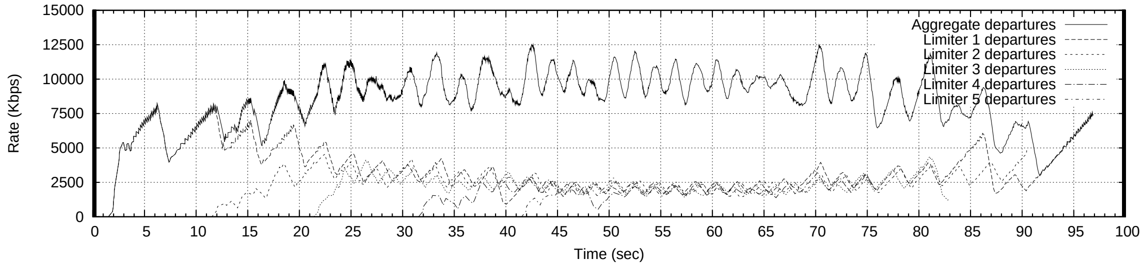
(a) Central token bucket



(b) Global random drop with 500-ms estimate interval



(c) Global random drop with 50-ms estimate interval



(d) Flow proportional share 500-ms estimate interval

Figure 6: Time series of forwarding rate for a flow join experiment. Every 10 seconds, a flow joins at an unused limiter.

limiter to the centralized token bucket (CTB). Recall that an important benchmark of our designs is their ability to reproduce a distribution of flow fairness at least as good as that of CTB. If they do, their points will closely follow the $x = y$ line; points below the line are less fair, indicating poor limiter behavior and points above the line indicate that the rate limiting algorithm produced better fairness than CTB.

Figure 5 compares distributions for all algorithms in our baseline experiment. GTB has fairness values around 0.7, which corresponds to the 7-flow aggregate unfairly dominating the 3-flow aggregate. This behavior is clearly visible in Figure 3(b), where the 7-flow limiter receives almost all the bandwidth. GRD and FPS, on the other hand, exhibit distributions that are at or above that of CTB. GRD, in fact, has a fairness index close to 1.0—much better than CTB. We verify this counter-intuitive result by comparing the performance of CTB with that of a single GRD limiter (labeled "Central Random Drop" in the figure). It is not surprising, then, that FPS is less fair than GRD, since it uses a token bucket at each limiter to enforce the local rate limit.[3]

Additionally, with homogeneous flows across a wide range of parameters—estimate intervals from 10 ms to 500 ms and EWMA from 0 to 0.75—we find that GTB and GRD are sensitive to estimate intervals, as they attempt to track packet-level behaviors (we omit the details for space). In general, GTB exhibits poor fairness for almost all choices of EWMA and estimate interval, and performs well only when the estimate interval is small and the EWMA is set to 0 (no filter). We conjecture that GTB needs to sample the short-term behavior of TCP in congestion avoidance, since considering solely aggregate demand over long time intervals fails to capture the increased aggressiveness of a larger flow aggregate. We

[3]In future work, we plan to experiment with a local GRD-like random drop mechanism instead of a token bucket in FPS; this will improve the fairness of FPS in many scenarios.

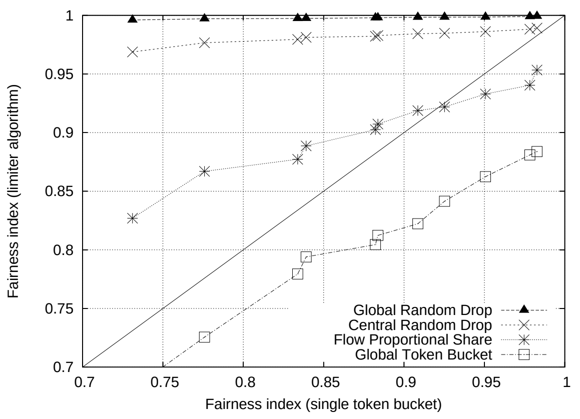


Figure 5: Quantile-quantile plots of a single token bucket vs. distributed limiter implementations. For each point (x, y), x represents a quantile value for fairness with a single token bucket and y represents the same quantile value for fairness for the limiter algorithm.

verified that GTB provides better fairness if we lengthen TCP's periodic behavior by growing its RTT. Since all results show that GTB fails with anything but the smallest estimate interval, we do not consider it further.

GRD is sensitive to the estimate interval, but in terms of short-term utilization, not flow fairness, since it maintains the same drop probability until it receives new updates. Thus, it occasionally drops at a higher-than-desired rate, causing congestion-responsive flows to back off significantly. While its long-term fairness remains high even for 500-ms estimate intervals, short-term utilization becomes exceedingly poor. By contrast, for homogeneous flows, FPS appears insensitive to the estimate interval, since flow-level demand is constant. Both GRD and FPS require an EWMA to smooth input demand to avoid over-reacting to short-term burstiness.[4]

5.2 Flow dynamics

We now investigate responsiveness (time to convergence and stability) by observing the system as flows arrive and depart. We sequentially add flows to a system of 5 limiters and observe the convergence to fair share of each flow. Figure 6(a) shows the reference time-series behavior for a centralized token bucket. Note that even through a single token bucket, the system is not completely fair or stable as flows arrive or depart due to TCP's burstiness. With a 500-ms estimate interval, GRD (Figure 6(b)) fails to capture the behavior of the central token bucket. Only with an order-of-magnitude smaller estimate interval (Figure 6(c)) is GRD able to approximate the central token bucket, albeit with increased fairness. FPS (Figure 6(d)), on the other hand, exhibits the least amount of variation in forwarded rate even with a 500-ms estimate interval, since flow-level demand is sufficiently constant over half-second intervals. This experiment illustrates that the behavior GRD must observe occurs at a packet-level time scale: large estimate intervals cause GRD to lose track of the global demand, resulting in chaotic packet drops. FPS, on the other hand, only requires updates as flows arrive, depart, or change their behavior.

[4]Though neither are particularly sensitive to EWMA, we empirically determined that a reasonable setting of the 1-second EWMA is 0.1. We use this value unless otherwise noted.

	CTB	GRD	FPS
Goodput (bulk mean)	6900.90	7257.87	6989.76
(stddev)	125.45	75.87	219.55
Goodput (web mean)	1796.06	1974.35	2090.25
(stddev)	104.32	93.90	57.98
Web rate (h-mean) [0,5000)	28.17	25.84	25.71
[5000, 50000)	276.18	342.96	335.80
[50000, 500000)	472.09	612.08	571.40
[500000, ∞)	695.40	751.98	765.26
Fairness (bulk mean)	0.971	0.997	0.962

Table 1: Goodput and delivered rates (Kbps), and fairness for bulk flows over 10 runs of the Web flow experiment. We use mean values for goodput across experiments and use the harmonic mean of rates (Kbps) delivered to Web flows of size (in bytes) within the specified ranges.

5.3 Traffic distributions

While TCP dominates cloud-based service traffic, the flows themselves are far from regular in their size, distribution, and duration. Here we evaluate the effects of varying traffic demands by considering Web requests that contend with long-running TCP flows for limiter bandwidth. To see whether our rate limiting algorithms can detect and react to Web-service demand, we assign 10 long-lived (bulk) flows to one limiter and the service requests to the other; this represents the effective worst-case for DRL since short and long flows cannot exert ordinary congestive pressures upon each other when isolated. We are interested in the ability of both traffic pools to attain the correct aggregate utilization, the long-term fairness of the stable flows, and the service rates for the Web flows.

Since we do not have access to traffic traces from deployed cloud-based services, we use a prior technique to derive a distribution of Web object sizes from a CAIDA Web trace for a high-speed OC-48 MFN (Metropolitan Fiber Network) Backbone 1 link (San Jose to Seattle) that follows a heavy-tailed distribution [36]. We fetch objects in parallel from an Apache Web server using `httperf` via a limiter. We distribute requests uniformly over objects in the trace distribution. Requests arrive according to a Poisson process with average μ of 15.

Table 1 gives the delivered rates for the Web flows of different sizes and the delivered rates for the 10-flow aggregates in each scenario across 10 runs. This shows that the 10-flow aggregate achieved a comparable allocation in each scenario. When seen in conjunction with the Web download service rates, it also indicates that the Web traffic aggregate in the other limiter received the correct allocation. Considering the Web flow service rates alone, we see that both GRD and FPS exhibit service rates close to that of a single token bucket, even for flows of significantly different sizes. The fairness index of the long-lived flows once again shows that GRD exhibits higher fairness than either CTB or FPS. FPS does not benefit from the fact that it samples flow-level behavior, which, in this context, is no more stable than the packet-level behavior observed by GRD.

5.4 Bottlenecked TCP flows

So far, the limiters represent the bottleneck link for each TCP flow. Here we demonstrate the ability of FPS to correctly allocate rate across aggregates of bottlenecked and unbottlenecked flows. The experiment in Figure 7 begins as our baseline 3-to-7 flow skew experiment where 2 limiters enforce a 10 Mbps limit. Around 15 seconds, the 7-flow aggregate experiences an upstream 2-Mbps bot-

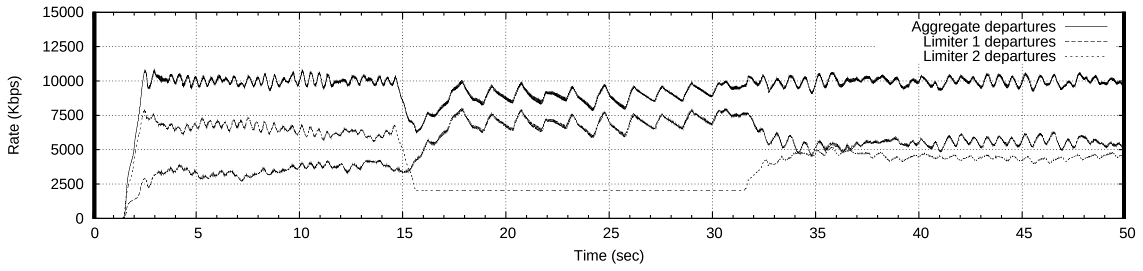


Figure 7: FPS rate limiting correctly adjusting to the arrival of bottlenecked flows.

	CTB	GRD	FPS
Aggregate (Mbps)	10.57	10.63	10.43
Short RTT (Mbps)	1.41	1.35	0.92
(stddev)	0.16	0.71	0.15
Long RTT (Mbps)	0.10	0.16	0.57
(stddev)	0.01	0.03	0.05

Table 2: Average throughput for 7 short (10-ms RTT) flows and 3 long (100 ms) RTT flows distributed across 2 limiters.

tleneck, and FPS quickly re-apportions the remaining 8 Mbps of rate across the 3 flows at limiter 1. Then, at time 31, a single unbottlenecked flow arrives at limiter 2. FPS realizes that an unbottlenecked flow exists at limiter 2, and increases the allocation for the (7+1)-flow aggregate. In a single pipe, the 4 unbottlenecked flows would now share the remaining 8 Mbps. Thus, limiter 2 should get 40% of the global limit, 2 Mbps from the 7 bottlenecked flows, and 2 Mbps from the single unbottlenecked flow. By time 39, FPS apportions the rate in this ratio.

5.5 Mixed TCP flow round-trip times

TCP is known to be unfair to long-RTT flows. In particular, short-RTT flows tend to dominate flows with longer RTTs when competing at the same bottleneck, as their tighter control loops allow them to more quickly increase their transmission rates. FPS, on the other hand, makes no attempt to model this bias. Thus, when the distribution of flow RTTs across limiters is highly skewed, one might be concerned that limiters with short-RTT flows would artificially throttle them to the rate achieved by longer-RTT flows at other limiters. We conduct a slight variant of the baseline experiment, with two limiters and a 3-to-7 flow split. In this instance, however, all 7 flows traversing limiter 2 are “short” (10-ms RTT), and the 3 flows traversing limiter 1 are “long” (100-ms RTT), representing a worst-case scenario. Table 2 shows the aggregate delivered throughput, as well as the average throughput for short and long-RTT flows for the different allocators. As expected, FPS provides a higher degree of fairness between RTTs, but all three limiters deliver equivalent aggregate rates.

5.6 Scaling

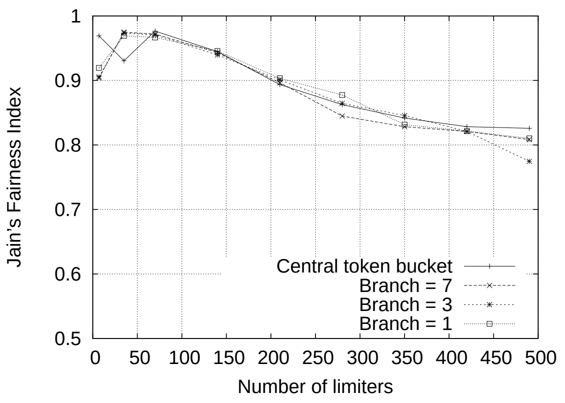
We explore scalability along two primary dimensions: the number of flows, and the number of limiters. First we consider a 2-limiter setup similar to the baseline experiment, but with a global rate limit of 50 Mbps. We send 5000 flows to the two limiters in a 3-7 ratio: 1500 flows to the first and 3500 to the second. GRD and FPS produce utilization of 53 and 46 Mbps and flow fairness of 0.44 and 0.33 respectively. This is roughly equal to that of a single token bucket with 5000 flows (which yielded 51 Mbps and 0.34). This poor fairness is not surprising, as each flow has only 10 Kbps, and prior work has shown that TCP is unfair under such conditions [28]. Nevertheless, our limiters continue to perform well with many flows.

Next, we investigate rate limiting with a large number of limiters and different inter-limiter communication budgets, in an environment in which gossip updates can be lost. We consider a topology with up to 490 limiters; our testbed contains 7 physical machines with 70 limiters each. Flows travel from the source through different limiter nodes, which then forward the traffic to the sink. (We consider TCP flows here and use symmetric paths for the forward and reverse directions of a flow.) We set the global rate limit to 50 Mbps and the inter-limiter and source-sink RTTs to 40 ms. Our experiment setup has the number of flows arriving at each limiter chosen uniformly at random from 0 to 5. For experiments with the same number of limiters, the distribution and number of flows is the same. We start 1 random flow from the above distribution every 100 ms; each flow lives for 60 seconds.

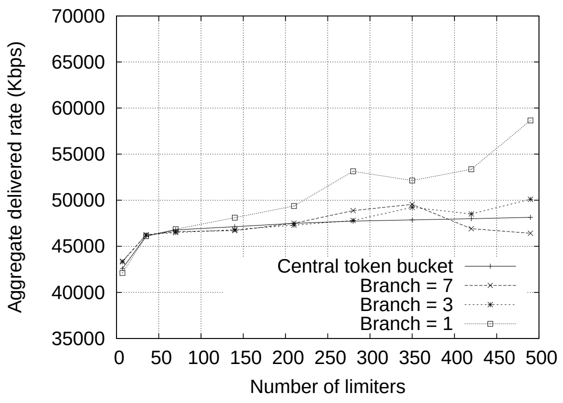
To explore the effect of communication budget, we vary the branching factor (*br*) of the gossip protocol from 1 to 7; for a given value, each extra limiter incurs a fixed communication cost. Figure 8 shows the behavior of FPS in this scaling experiment. At its extreme there are 1249 flows traversing 490 limiters. (We stop at 490 not due to a limitation of FPS, but due to a lack of testbed resources.) When *br* = 3, each extra limiter consumes $48 \times 20 \times 3 = 2.88$ Kbps. Thus, at 490 limiters, the entire system consumes a total of 1.4 Mbps of bandwidth for control communication—less than 3% overhead relative to the global limit.

We find that beyond a branching factor of 3, there is little benefit either in fairness or utilization. Indeed, extremely high branching factors lead to message and ultimately information loss. Beyond 50 limiters, GRD fails to limit the aggregate rate (not shown), but this is not assuaged by an increasing communication budget (increasing *br*). Instead it indicates GRD’s dependence on swiftly converging global arrival rate estimates. In contrast, FPS, because it depends on more slowly moving estimates of the number of flows at each limiter, maintains the limit even at 490 limiters.

This experiment shows that limiters rely upon up-to-date summaries of global information, and these summaries may become stale when delayed or dropped by the network. In particular, our concern lies with stale under-estimates that cause the system to overshoot the global rate; a completely disconnected system—due to either congestion, failure, or attack—could over-subscribe the global limit by a factor of N. We can avoid these scenarios by initializing limiters with the number of peers, N, and running a light-weight membership protocol [24] to monitor the current number of connected peers. For each disconnected peer, the limiter can reduce the global limit by $\frac{1}{N}$, and set each stale estimate to zero. This conservative policy drives the limiters toward a $\frac{1}{N}$ limiter (where each limiter enforces an equal fraction of the global aggregate) as disconnections occur. More generally, though, we defer analysis of DRL under adversarial or Byzantine conditions to future work.



(a) FPS fairness.



(b) FPS delivered rate.

Figure 8: Fairness and delivered rate vs. number of limiters in the scaling experiment.

5.7 Limiting cloud-based services

Finally, we subject FPS to inter-limiter delays, losses, and TCP arrivals and flow lengths similar to those experienced by a cloud-based service. As in Section 5.3, we are not concerned with the actual service being provided by the cloud or its computational load—we are only interested in its traffic demands. Hence, we emulate a cloud-based service by using generic Web requests as a stand-in for actual service calls. We co-locate distributed rate limiters with 10 PlanetLab nodes distributed across North America configured to act as component servers. Without loss of generality, we focus on limiting only out-bound traffic from the servers; we could just as easily limit in-bound traffic as well, but that would complicate our experimental infrastructure. Each PlanetLab node runs Apache and serves a series of Web objects; an off-test-bed client machine generates requests for these objects using `wget`. The rate limiters enforce an aggregate global rate limit of 5 Mbps on the response traffic using a 100-ms estimate interval and a gossip branching factor of 4, resulting in a total control bandwidth of 38.4 Kbps. The inter-limiter control traffic experienced 0.47% loss during the course of the experiment.

Figure 9 shows the resulting time-series plot. Initially each content server has demands to serve 3 requests simultaneously for 30 seconds, and then the total system load shifts to focus on only 4 servers for 30 seconds, emulating a change in the service's request load, perhaps due to a phase transition in the service, or a flash crowd of user demand. Figure 9(a) shows the base case, where a static $\frac{1}{N}$ limiting policy cannot take advantage of unused capacity at the other 6 sites. In contrast, FPS, while occasionally bursting above the limit, accommodates the demand swing and delivers the full rate limit.

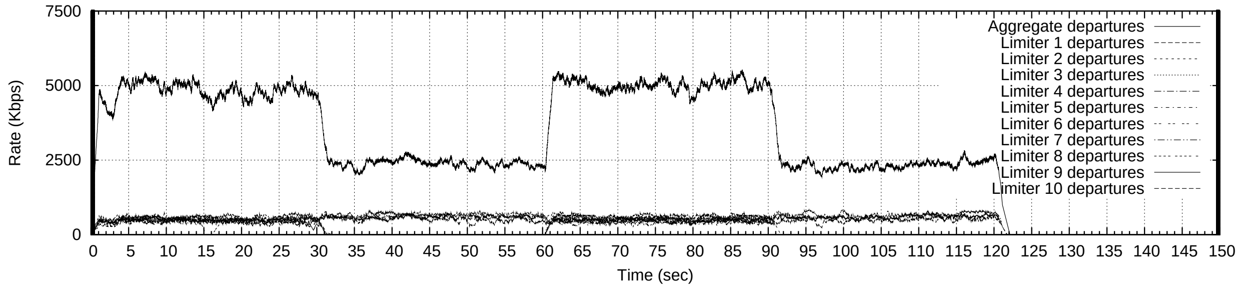
6. RELATED WORK

The problem of online, distributed resource allocation is not a new one, but to our knowledge we are the first to present a concrete realization of distributed traffic rate limiting. While there has been considerable work to determine the optimal allocation of bandwidth between end-point pairs in virtual private networks (VPNs), the goal is fundamentally different. In the VPN hose model [23], the challenge is to meet various quality-of-service guarantees by provisioning links in the network to support any traffic distribution that does not exceed the bandwidth guarantees at each end point in the VPN. Conversely, the distributed rate limiting problem is to control the aggregate bandwidth utilization at all limiters in the network, regardless of the available capacity at the ingress or egress points.

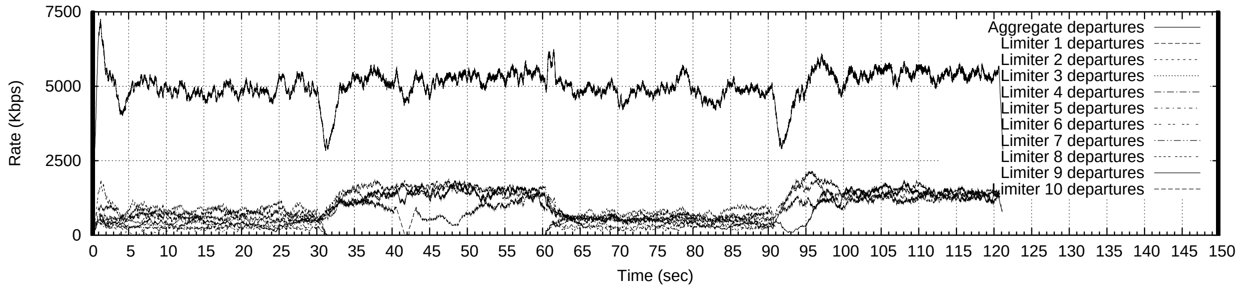
Distributed rate limiting can be viewed as a continuous form of distributed admission control. Distributed admission control allows participants to test for and acquire capacity across a set of network paths [21, 40]; each edge router performs flow-admission tests to ensure that no shared hop is over-committed. While our limiters similarly "admit" traffic until the virtual limiter has reached capacity, they do so in an instantaneous, reservation-free fashion.

Ensuring fairness across limiters can be viewed as a distributed instance of the link-sharing problem [15]. A number of packet scheduling techniques have been developed to enforce link-sharing policies, which provide bandwidth guarantees for different classes of traffic sharing a single link. These techniques, such as weighted fair queuing [11], apportion link capacity among traffic classes according to some fixed weights. These approaches differ from ours in two key respects. First, by approximating generalized processor sharing [31], they allocate excess bandwidth across backlogged classes in a max-min fair manner; we avoid enforcing any explicit type of fairness between limiters, though FPS tries to ensure max-min fairness between flows. Second, most class-based fair-queuing schemes aim to provide isolation between packets of different classes. In contrast, we expose traffic at each limiter to all other traffic in the system, preserving whatever implicit notion of fairness would have existed in the single-limiter case. As discussed in Section 3, we use a token bucket to define the reference behavior of a single limiter. There are a broad range of active queue management schemes that could serve equally well as a centralized reference [13, 14]. Determining whether similar distributed versions of these sophisticated AQM schemes exist is a subject of future work.

The general problem of using and efficiently computing aggregates across a distributed set of nodes has been studied in a number of other contexts. These include distributed monitoring [12], triggering [19], counting [33, 38], and data stream querying [5, 26]. Two systems in particular also estimate aggregate demand to apportion shared resources at multiple points in a network. The first is a token-based admission architecture that considers the problem of parallel flow admissions across edge routers [7]. Their goal is to divide the total capacity fairly across allocations at edge routers by setting an edge router's local allocation quota in proportion to its share of the request load. However they must revert to a first-come first-served allocation model if ever forced to "revoke" bandwidth to maintain the right shares. Zhao *et al.* use a similar protocol to enforce service level agreements between server clusters [41]. A



(a) $\frac{1}{N}$ limiting.



(b) Flow Proportional Share

Figure 9: A time-series graph rate limiting at 10 PlanetLab sites across North America. Each site is a Web server, fronted by a rate limiter. Every 30 seconds total demand shifts to four servers and then back to all 10 nodes. The top line represents aggregate throughput; other lines represent the served rates at each limiter.

set of layer-7 switches employ a "coordinated" request queuing algorithm to distribute service requests in proportion to the aggregate sum of switch queue lengths.

7. CONCLUSION

As cloud-based services transition from marketing vaporware to real, deployed systems, the demands on traditional Web-hosting and Internet service providers are likely to shift dramatically. In particular, current models of resource provisioning and accounting lack the flexibility to effectively support the dynamic composition and rapidly shifting load enabled by the software as a service paradigm. We have identified one key aspect of this problem, namely the need to rate limit network traffic in a distributed fashion, and provided two novel algorithms to address this pressing need.

Our experiments show that naive implementations based on packet arrival information are unable to deliver adequate levels of fairness, and, furthermore, are unable to cope with the latency and loss present in the wide area. We presented the design and implementation of two limiters, a protocol-agnostic global random drop algorithm and a flow proportional share algorithm appropriate for deployment in TCP-based Web-services environments that is robust to long delays and lossy inter-limiter communication. By translating local arrival rate into a flow weight, FPS communicates a unit of demand that is inherently more stable than packet arrivals. Thus, it is possible for the local arrival rates to fluctuate, but for the flow weight to remain unchanged.

Our results demonstrate that it is possible to recreate, at distributed points in the network, the flow behavior that end users and network operators expect from a single centralized rate limiter. Moreover, it is possible to leverage knowledge of TCP's congestion avoidance algorithm to do so using little bandwidth, hundreds of limiters, thousands of flows, and realistic Internet delays and losses. While our experience with GRD indicates it may be difficult to develop a robust protocol-agnostic limiter, it is likely that UDP-based protocols deployed in a cloud will have their own congestion-control algorithms. Hence, FPS could be extended to calculate a flow weight for these as well.

Acknowledgements

We are grateful to Alberto Blanc, Jay Chen, Yu-Chung Cheng, Stefan Savage, Scott Shenker, Amin Vahdat, George Varghese, and Geoff Voelker for their valuable comments. This work is supported by the National Science Foundation through grant CNS-627167.

8. REFERENCES

[1] Packeteer. `http://www.packeteer.com`.

[2] Akamai Technologies. Personal communication, June 2007.

[3] Amazon. Elastic compute cloud. `http://aws.amazon.com/ec2`.

[4] G. Appenzeller, I. Keslassy, and N. McKeown. Sizing router buffers. In *Proceedings of ACM SIGCOMM*, 2004.

[5] B. Babcock and C. Olston. Distributed top-k monitoring. In *Proceedings of ACM SIGMOD*, 2003.

[6] D. Bertsekas and R. Gallager. *Data Networks*. Prentice Hall, 1987.

[7] S. Bhatnagar and B. Nath. Distributed admission control to support guaranteed services in core-stateless networks. In *Proceedings of IEEE INFOCOM*, 2003.

[8] D. F. Carr. How Google works. *Baseline Magazine*, July 2006.

[9] G. Carraro and F. Chong. Software as a service (SaaS): An enterprise perspective. *MSDN Solution Architecture Center*, Oct. 2006.

[10] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of ACM PODC*, 1987.

[11] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *Proceedings of ACM SIGCOMM*, 1989.

[12] M. Dilman and D. Raz. Efficient reactive monitoring. In *Proceedings of IEEE INFOCOM*, 2001.

[13] W. Feng, K. Shin, D. Kandlur, and D. Saha. The blue active queue management algorithms. *IEEE/ACM Transactions on Networking*, 10(4), 2002.

[14] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4), 1993.

[15] S. Floyd and V. Jacobson. Link-sharing and resource management models for packet networks. *IEEE/ACM Transactions on Networking*, 3(4), 1995.

[16] I. Gupta, A.-M. Kermarrec, and A. J. Ganesh. Efficient epidemic-style protocols for reliable and scalable multicast. In *Proceedings of IEEE SRDS*, 2002.

[17] D. Hinchcliffe. 2007: The year enterprises open thier SOAs to the Internet? *Enterprise Web 2.0*, Jan. 2007.

[18] M. Huang. Planetlab bandwidth limits. `http://www.planet-lab.org/doc/BandwidthLimits.php`.

[19] A. Jain, J. M. Hellerstein, S. Ratnasamy, and D. Wetherall. A wakeup call for internet monitoring systems: The case for distributed triggers. In *Proceedings of HotNets-III*, 2004.

[20] R. Jain, D. M. Chiu, and W. Hawe. A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. Technical report, DEC Research Report TR-301, 1984.

[21] S. Jamin, P. Danzig, S. Shenker, and L. Zhang. A measurement-based admission control algorithm for integrated services packet networks. In *Proceedings of ACM SIGCOMM*, 1995.

[22] D. Kempe, A. Dobra, and J. Gehrke. Gossip-based computation of aggregate information. In *Proceedings of IEEE FOCS*, 2003.

[23] A. Kumar, R. Rastogi, A. Siberschatz, and B. Yener. Algorithms for provisioning virtual private networks in the hose model. *IEEE/ACM Transactions on Networking*, 10(4), 2002.

[24] J. Liang, S. Y. Ko, I. Gupta, and K. Nahrstedt. MON: On-demand overlays for distributed system management. In *Proceedings of USENIX WORLDS*, 2005.

[25] J. Ma, K. Levchenko, C. Kriebich, S. Savage, and G. M. Voelker. Automated protocol inference: Unexpected means of identifying protocols. In *Proceedings of ACM/USENIX IMC*, 2006.

[26] A. Manjhi, V. Shkapenyuk, K. Dhamdhere, and C. Olston. Finding (recently) frequent items in distributed data streams. In *Proceedings of IEEE ICDE*, 2005.

[27] P. Marks. Mashup' websites are a hacker's dream come true. *New Scientist magazine*, 2551:28, May 2006.

[28] R. Morris. TCP behavior with many flows. In *Proceedings of IEEE ICNP*, 1997.

[29] J. Musser. Programmable web. `http://www.programmableweb.com`.

[30] A. M. Odlyzko. Internet pricing and the history of communications. *Computer Networks*, 36:493–517, 2001.

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

[32] B. Raghavan and A. C. Snoeren. A system for authenticated policy-compliant routing. In *Proceedings of ACM SIGCOMM*, 2004.

[33] N. Shavit and A. Zemach. Diffracting trees. *ACM Transactions on Computer Systems*, 14(4), 1996.

[34] I. Stoica, S. Shenker, and H. Zhang. Core-stateless fair queueing: a scalable architecture to approximate fair bandwidth allocations in high speed networks. In *Proceedings of ACM SIGCOMM*, 1998.

[35] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker. Scalability and accuracy in a large-scale network emulator. In *Proceedings of USENIX OSDI*, 2002.

[36] K. Vishwanath and A. Vahdat. Realistic and responsive network traffic generation. In *Proceedings of ACM SIGCOMM*, 2006.

[37] L. Wang, K. Park, R. Pang, V. S. Pai, and L. Peterson. Reliability and security in the CoDeeN content distribution network. In *Proceedings of USENIX*, 2004.

[38] R. Wattenhofer and P. Widmayer. An inherent bottleneck in distributed counting. In *Proceedings of ACM PODC*, 1997.

[39] X. Yang, D. Wetherall, and T. Anderson. A DoS-limiting network architecture. In *Proceedings of ACM SIGCOMM*, 2005.

[40] Z.-L. Zhang, Z. Duan, and Y. T. Hou. On scalable design of bandwidth brokers. *IEICE Transactions on Communications*, E84-B(8), 2001.

[41] T. Zhao and V. Karamcheti. Enforcing resource sharing agreements among distributed server clusters. In *Proceedings of IEEE IPDPS*, 2002.

APPENDIX

Here we show that FPS correctly stabilizes to the "correct" allocations at all limiters in the presence of both unbottlenecked and bottlenecked flows. First, we present a model of TCP estimation over n limiters. Let $a_1, a_2, \ldots, a_n$ be the number of unbottlenecked flows at limiters 1 to n respectively. Similarly, let $B_1, B_2, \ldots, B_n$ be the local bottlenecked flow rates (which may include multiple flows). At the ith limiter, there exists a local rate limit, l_i. These limits are subject to the constraint that $l_1 + l_2 + \cdots + l_n = L$, where L is the global rate limit. Let $U = L - \sum_i B_i$ represent the total amount of rate available for unbottlenecked flows. Let $A = \sum_i a_i$ represent the total number of unbottlenecked flows across all limiters. Given these values, a TCP estimator outputs a tuple of weights $(w_1, w_2, \ldots, w_n)$ that are used by FPS to assign rate limits at all limiters. Suppose we are given perfect global knowledge and are tasked to compute the correct allocations at all limiters. The allocation would be

$$I = (U \cdot \frac{a_1}{A} + B_1, U \cdot \frac{a_2}{A} + B_2, \ldots, U \cdot \frac{a_n}{A} + B_n).$$

Note that these weights are also equal to the actual rate limits assigned at each node. This corresponds to an allocation which would result for each limiter's flow aggregate had all flows (globally) been forced through a single pipe of capacity L.

FPS first estimates the rate of a single unbottlenecked flow at each limiter. Once stabilized, such a flow at limiter number i will receive a rate f (where l_i is the *current* rate limit at limiter i):

$$f = \frac{l_i - B_i}{a_i}.$$

Given these flow rates, FPS will compute a new weight w_i at each limiter:

$$w_i = \frac{l_i \cdot a_i}{l_i - B_i}.$$

Once FPS arrives at the ideal allocation, it will remain at the ideal allocation in the absence of any demand changes. That is, suppose $(l_1, \ldots, l_n) = (I_1, \ldots, I_n)$. We claim that the newly computed weights $(w_1, \ldots, w_n)$ result in the same allocation; equivalently,

$$\frac{w_1}{w_1 + \cdots + w_n} = \frac{I_1}{I_1 + \cdots + I_n}.$$

The weights computed given this starting state are, for each i,

$$w_i = \frac{(U \cdot \frac{a_i}{A} + B_i) \cdot a_i}{(U \cdot \frac{a_i}{A} + B_i) - B_i}.$$

Thus, considering the allocation at limiter 1,

$$\frac{w_1}{w_1 + \cdots + w_n} = \frac{\frac{Ua_1 + AB_1}{U}}{\frac{Ua_1 + AB_1}{U} + \cdots + \frac{Ua_n + AB_n}{U}},$$

which is equal to

$$\frac{Ua_1 + AB_1}{Ua_1 + AB_1 + \cdots + Ua_n + AB_n} = \frac{I_1}{I_1 + \cdots + I_n},$$

the ideal allocation fraction for limiter 1. The allocations at other limiters are analogous.