Sharing the Data Center Network

Alan Shieh[‡†], Srikanth Kandula[‡], Albert Greenberg[∨], Changhoon Kim[∨], Bikas Saha[⋆]
Microsoft Research[‡], Cornell University[†], Windows Azure[∨], Microsoft Bing[⋆]
ashieh@cs.cornell.edu {srikanth,albert,chakim,bikas}@microsoft.com

Abstract– While today's data centers are multiplexed across many non-cooperating applications, they lack effective means to share their network. Relying on TCP's congestion control, as we show from experiments in production data centers, opens up the network to denial of service attacks and performance interference. We present Seawall, a network bandwidth allocation scheme that divides network capacity based on an administrator-specified policy. Seawall computes and enforces allocations by tunneling traffic through congestion controlled, point to multipoint, edge to edge tunnels. The resulting allocations remain stable regardless of the number of flows, protocols, or destinations in the application's traffic mix. Unlike alternate proposals, Seawall easily supports dynamic policy changes and scales to the number of applications and churn of today's data centers. Through evaluation of a prototype, we show that Seawall adds little overhead and achieves strong performance isolation.

1. INTRODUCTION

Data centers are crucial to provide the large volumes of compute and storage resources needed by today's Internet businesses including web search, content distribution and social networking. To achieve cost efficiencies and on-demand scaling, cloud data centers [5, 28] are highly-multiplexed shared environments, with VMs and tasks from multiple tenants coexisting in the same cluster. Since these applications come from unrelated customers, they are largely uncoordinated and mutually untrusting. Thus, the potential for network performance interference and denial of service attacks is high, and so performance predictability remains a key concern [8] for customers evaluating a move to cloud datacenters.

While data centers provide many mechanisms to schedule local compute, memory, and disk resources [10, 15], existing mechanisms for apportioning network resources fall short. End host mechanisms such as TCP congestion control (or variants such as TFRC and DCCP) are widely deployed, scale to existing traffic loads, and, to a large extent, determine network sharing today via a notion of flow-based fairness. However, TCP does little to isolate tenants from one another: poorly-designed or malicious applications can consume network capacity, to the detriment of other applications, by opening more flows or using non-compliant protocol implementations that ignore congestion control. Thus, while resource allocation using TCP is scalable and achieves high network utilization, it does not provide robust performance isolation.

Switch and router mechanisms (e.g., CoS tags, Weighted Fair Queuing, reservations, QCN [29]) are better decoupled from tenant misbehavior. However, these features, inherited from enterprise networks and the Internet, are of limited use when applied to the demanding cloud data center environment, since they cannot keep up with the scale and the churn observed in datacenters (e.g., numbers of tenants, arrival rate of new VMs), can only obtain isolation at the cost of network utilization, or might require new hardware.

For a better solution, we propose Seawall, an edge based mechanism that lets administrators prescribe how their network is shared. Seawall works irrespective of traffic characteristics such as the number of flows, protocols or participants. Seawall provides a simple abstraction: given a *network weight* for each local entity that serves as a traffic source (VM, process, etc.), Seawall ensures that along all network links, the share of bandwidth obtained by the entity is proportional to its weight. To achieve efficiency, Seawall is work-conserving, proportionally redistributing unused shares to currently active sources.

Beyond simply improving security by mitigating DoS attacks from malicious tenants and generalizing existing *use-what-you-pay-for* provisioning models, per-entity weights also enable better control over infrastructure services. Data centers often mix latency- and throughput-sensitive tasks with background infrastructure services. For instance, customer-generated web traffic contends with the demands of VM deployment and migration tasks. Per-entity weights obviate the need to hand-craft every individual service.

Further, per-entity weights also enable better control over application-level goals. Network allocation decisions can have significant impact on end-to-end metrics such as completion time or throughput. For example, in a map-reduce cluster, a reduce task with a high fan-in can open up many more flows than map tasks sharing the same bottleneck. Flow-based fairness prioritizes high fan-in reduce tasks over other tasks, resulting in imbalanced progress that leaves CPU resources idle and degrades cluster throughput. By contrast, Seawall decouples network allocation from communications patterns.

Seawall achieves scalable resource allocation by reducing the network sharing problem to an instance of distributed congestion control. The ubiquity of TCP shows that such algorithms can scale to large numbers of participants, adapt quickly to change, and can be implemented strictly at the edge. Though Seawall borrows from TCP, Seawall's architecture and control loop ensure robustness against tenant misbehavior. Seawall uses a shim layer at the sender that makes policy compliance mandatory by forcing all traffic into congestion-controlled tunnels. To prevent tenants from bypassing Seawall, the shim runs in the virtualization or platform network stack, where it is well-isolated from tenant code.

Simply enforcing a separate TCP-like tunnel to every destination would permit each source to achieve higher rate by communicating with more destinations. Since this does not achieve the desired policy based on per-entity weights, Seawall instead uses a novel control loop that combines feedback from multiple destinations.

Overall, we make three contributions. First, we identify problems and missed opportunities caused by poor network resource allocation. Second, we explore at length the tradeoffs in building network allocation mechanisms for cloud data centers. Finally, we design and implement an architecture and control loop that are robust against malicious, selfish, or buggy tenant behavior. We have built a prototype of Seawall as a Windows NDIS filter. From experiments in a large server cluster, we show that Seawall achieves proportional sharing of the network while remaining agnostic to tenant protocols and traffic patterns and protects against UDP- and TCP-based DoS attacks. Seawall provides these benefits while achieving line rate with low CPU overhead.

2. PROBLEMS WITH NETWORK SHARING IN DATACENTERS

To understand the problems with existing network allocation schemes, we examine two types of clusters that consist of several thousands of servers and are used in production. The first type is that of public infrastructure cloud services that rent virtual machines along with other shared services such as storage and load balancers. In these datacenters, clients can submit arbitrary VM images and choose which applications to run, who to talk to, how much traffic to send, when to send that traffic, and what protocols to use to exchange that traffic (TCP, UDP, # of flows). The second type is that of platform cloud services that support map-reduce workloads. Consider a map-reduce cluster that supports a search engine. It is used to analyze logs and improve query and advertisement relevance. Though this cluster is shared across many users and business groups, the execution platform (i.e., the job compiler and runtime) is proprietary code controlled by the datacenter provider.

Through case studies on these datacenters we observe how the network is shared today, the problems that arise from such sharing and the requirements for an improved sharing mechanism.

In all datacenters, the servers have multiple cores, multiple disks, and tens of GBs of RAM. The network is a tree like topology [26] with 20–40 servers in a rack and a small over-subscription factor on the upstream links of the racks.

2.1 Performance interference in infrastructure cloud services

Recent measurements demonstrate considerable variation in network performance metrics – medium instances in EC2 experience throughput that can vary by 66% [25, 43]. We conjecture, based on anecdotal evidence, that a primary reason for the variation is the inability to control the network traffic share of a VM.

Unlike CPU and memory, network usage is harder to control because it is a distributed resource. For example, consider the straw man where each VM's network share is statically limited to a portion of the host's NIC rate (the equivalent of assigning the VM a fixed number of cores or a static memory size). A tenant with many VMs can cumulatively send enough traffic to overflow the receiver, some network link en route to that host, or other network bottlenecks. Some recent work [33] shows how to co-locate a trojan VM with a target VM. Using this, a malicious tenant can degrade the network performance of targeted victims. Finally, a selfish client, by using variable numbers of flows, or higher rate UDP flows, can hog network bandwidth.

We note that out-of-band mechanisms to mitigate these problems exist. Commercial cloud providers employ a combination of such mechanisms. First, the provider can account for the network usage of tenants (and VMs) and quarantine or ban the misbehavers. Second, cloud providers might provide even less visibility into their clusters to make it harder for a malicious client to co-locate with target VMs. However, neither approach is foolproof. Selfish or malicious traffic can mimic legitimate traffic, making it hard to distinguish. Further, obfuscation schemes may not stop a determined adversary.

Our position, instead, is to get at the root of the problem. The reason existing solutions fail is that they primarily rely on TCP flows. But VMs are free to choose their number of flows, congestion control variant, and even whether they respond to congestion, allowing a small number of VMs to disproportionately impact the network. Hence, we seek alternative ways to share the network that are independent of the clients' traffic matrices and implementations.

2.2 Poorly-performing schedules in Cosmos

We shift focus to Cosmos [9], a dedicated internal cluster that supports map-reduce workloads. We obtained detailed logs over several days from a production cluster with thousands of servers that supports the Bing search engine. The logs document the begin and end times of

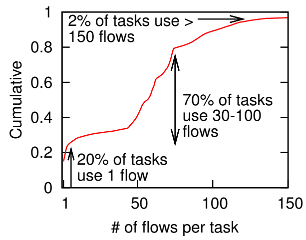


Figure 1: Distribution of the number of flows per task in Cosmos.

Task Type	#flows per task	% of net tasks
Aggregate	56.1	94.9
Partition	1.2	3.7
Extract	8.8	.2
Combine	2.3	1.0
other	1.0	.2

Figure 2: Variation in number of flows per task is due to the role of the task

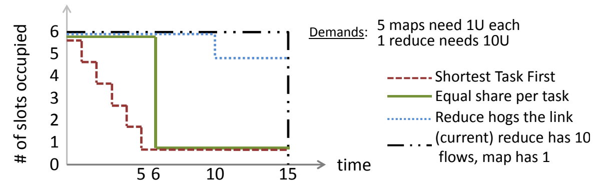


Figure 3: Poor sharing of the network leads to poor performance and wasted resources

jobs, tasks and flows in this cluster.

Performance interference happens here as well. Instances of high network load are common. A few entities (jobs, background services) contribute a substantial share of the traffic [22]. Tasks that move data over congested links suffer collateral damage – they are more likely to experience failures and become stragglers at the job level [6, 22].

Uniquely, however, we find that the de facto way of sharing the network leads to poor schedules. This is because schedulers for map-reduce platforms [27, 45] explicitly allocate local resources such as compute slots and memory. But, the underlying network primitives prevent them from exerting control over how tasks share the network. Map-reduce tasks naturally vary in the number of flows and the volume of data moved – a map task may have to read from just one location but a reduce task has to read data from all the map tasks in the preceding stage. Figure 1 shows that of the tasks that read data across racks, 20% of the tasks use just one flow, another 70% of the tasks vary between 30 and 100 flows, and 2% of the tasks use more than 150 flows. Figure 2 shows that this variation is due to the role of the task.

Because reduce tasks use a large number of flows, they starve other tasks that share the same paths. Even if the scheduler is tuned to assign a large number of compute slots for map tasks, just a few reduce tasks will cause these map tasks to be bottlenecked on the network. Thus, the compute slots held by the maps make little progress.

In principle, such unexpectedly idle slots could be put to better use on compute-heavy tasks or tasks that use less loaded network paths. However, current map-reduce schedulers do not support such load redistribution.[1]

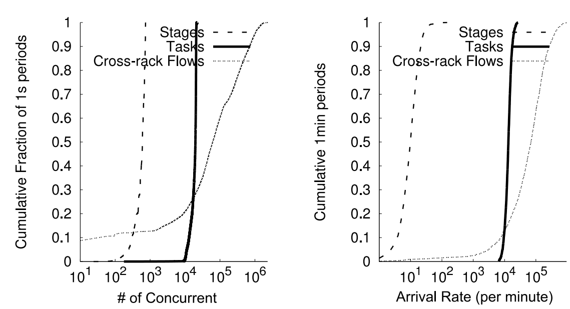
A simple example illustrates this problem. Figure 3 examines different ways of scheduling six tasks, five maps that each want to move 1 unit of data across a link of unit capacity and one reduce that wants to move 10 units of data from ten different locations over the same link. If the reduce uses 10 flows and each map uses 1 flow, as they do today, each of the flows obtains $\frac{1}{15}$'th of the link bandwidth and all six tasks finish at $t = 15$ (the schedule shown in black). The total activity period, since each task use local resources that no one else can use during the period it is active, is $6 * 15 = 90$.

If each task gets an even share of the link, it is easy to see that the map tasks will finish at $t = 6$ and the reduce task finishes at $t = 15$. In this case, the total activity period is $5 * 6 + 1 * 15 = 45$, or a 50% reduction in resource usage (the green solid line in Fig. 3). These spare resources can be used for other jobs or subsequent tasks within the same job.

The preceding example shows how the inherent variation in the way applications use the network can lead to poor schedules in the absence of control over how the network is shared. Our goal is to design ways of sharing the network that are efficient (no link goes idle if pent-up demand exists) and are independent of the traffic mix (UDP, #'s of TCP flows).

We note that prescribing the optimal bandwidth shares is a non-goal for this paper. In fact, evenly allocating bandwidth across tasks is not optimal for some metrics. If the provider has perfect knowledge about demands, scheduling the shortest remaining transfer first will minimize the activity period [18]. Going back to the example, this means that the five map tasks get exclusive access to the link and finish one after the other resulting in an activity period of 30 (the red dashed line in Fig. 3). However, this scheme has the side-effect of starving all the waiting transfers and requires perfect knowledge about client demands, which is hard to obtain in practice.

2.3 Magnitude of scale and churn



(a) Cosmos: Scale (b) Cosmos: Churn

Figure 4: Scale and churn seen in the observed datacenter.

We attempt to understand the nature of the sharing problem in production datacenters. We find that the number of classes to share bandwidth among is large and varies frequently. Figure 4(a) shows the distribution of the number of concurrent entities that share the examined Cosmos cluster. Note that the x-axis is in log scale. We see that at median, there are 500 stages (e.g., map, reduce, join), 10^4 tasks and 10^5 flows in the cluster. The number of traffic classes required is at least two orders of magnitude larger than is feasible with current CoS tags or the number of WFQ/DRR queues that switches can handle per port.

Figure 4(b) shows the distribution of the number of new arrivals in the observed cluster. Note that the x-axis is again in log scale. At median, 10 new stages, 10^4 new tasks and $5 * 10^4$ new flows arrive in the cluster every minute. Anecdotal analysis of EC2, based on decoding the instance identifiers, concluded that O(10^4) new VM instances are requested each day [34]. Updating VLANs or re-configuring switches whenever a VM arrives is several orders of magnitude more frequent than is achievable in today's enterprise networks.

Each of the observed data centers is large, with up to tens of thousands of servers, thousands of ToR switches, several tens of aggregation switches, load balancers, etc. Predicting traffic is easier in platform datacenters (e.g., Cosmos) wherein high level descriptions of the jobs are available. However, the scale and churn numbers indicate that obtaining up-to-date information (e.g., within a minute) may be a practical challenge. In cloud datacenters (e.g., EC2) traffic is even harder to predict because customer's traffic is unconstrained and privacy concerns limit instrumentation.

3. REQUIREMENTS

From the above case studies and from interviews with operators of production clusters, we identify these requirements for sharing the datacenter network.

An ideal network sharing solution for datacenters has to scale, keep up with churn and retain high network utilization. It must do so without assuming well-behaved or TCP-compliant tenants. Since changes to the NICs and switches are expensive, take some time to standardize and deploy, and are hard to customize once deployed, edge- and software- based solutions are preferable.

- **Traffic Agnostic, Simple Service Interface:** Tenants cannot be expected to know or curtail the nature of their traffic. It is good business sense to accommodate diverse applications. While it is tempting to design sharing mechanisms that require tenants to specify a traffic matrix, i.e., the pattern and volume of traffic between the tenant's VMs, we find this to be an unrealistic burden. Changes in demands from the tenant's customers and dynamics of their workload (e.g., map-reduce) will change the requirements. Hence, it is preferable to keep a thin service interface, e.g., have tenants choose a class of network service.
- **Require no changes to network topology or hardware:** Recently, many data center network topologies have been proposed [2, 3, 16, 21]. Cost benefit tradeoffs indicate that the choice of topology depends on the intended usage. For example, EC2 recently introduced a full bisection bandwidth network for high performance computing (HPC); less expensive EC2 service levels continue to use the over-subscribed tree topology. To be widely applicable, mechanisms to share the network should be agnostic to network topology.
- **Scale to large numbers of tenants and high churn:** To have practical benefit, any network sharing mechanism would need to scale to support the large workloads seen in real datacenters.
- **Enforce sharing without sacrificing efficiency:** Statically apportioning fractions of the bandwidth improves sharing at the cost of efficiency and can result in bandwidth fragmentation that makes it harder to accommodate new tenants. At the same time, a tenant with pent up demand can use no more than its reservation even if the network is idle.

To meet these requirements, Seawall relies on *congestion-controlled tunnels* implemented in the host but requires no per-flow state within switches. In this way, Seawall is independent of the physical data center network. Seawall does benefit from measurements at switches, if they are available. Seawall scales to large numbers of tenants and handles high churn, because provisioning new VMs or tasks is entirely transparent to the physical network. As tenants, VMs, or tasks come and go, there is no change to the physical network through signaling or configuration. Seawall's design exploits the homogeneity of the data center environment, where end host software is easy to change and topology is predictable. These properties enable Seawall to use a system architecture and algorithms that are impractical on the Internet yet well-suited for data centers.

4. Seawall DESIGN

Seawall exposes the following abstraction. A *network weight* is associated with each entity that is sharing the network. The entity can be any traffic source that is confined to a single node, such as a VM, process, or collection of port numbers, but not a tenant or set of VMs. On each link in the network, Seawall provides the entity with a bandwidth share that is proportional to its weight; i.e., an entity k with weight w_k sending traffic over link l obtains this share of the total capacity of that link $Share(k, l) = \frac{w_k}{\Sigma_{i \in Active(l)} w_i}$. Here, $Active(l)$ is the set of entities actively sending traffic across l. The allocation is end-to-end, i.e., traffic to a destination will be limited by the smallest $Share(k, l)$ over links on the path to that destination. The allocation is also work-conserving: bandwidth that is unused because the entity needs less than its share or because its traffic is bottlenecked elsewhere is re-apportioned among other users of the link in

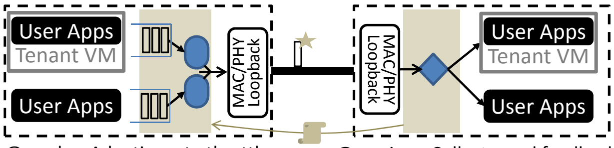


Figure 5: Seawall's division of functionality. (New components are shaded gray.)

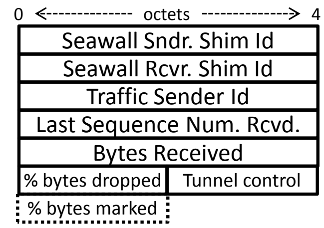


Figure 6: Content of Seawall's feedback packet

proportion to their weights. Here, we present a distributed technique that holds entities to these allocations while meeting our design requirements.

Weights can be adjusted dynamically and allocations re-converge rapidly. The special case of assigning the same weight to all entities divides bandwidth in a max-min fair fashion. By specifying equal weights to VMs, a public cloud provider can avoid performance interference from misbehaving or selfish VMs (§2.1). We defer describing further ways to configure weights and enforcing global allocations, such as over a set of VMs belonging to the same tenant, to §4.6.

4.1 Data path

To achieve the desired sharing of the network, Seawall sends traffic through congestion-controlled logical tunnels. As shown in Figure 5, these tunnels are implemented within a shim layer that intercepts all packets entering and leaving the server. At the sender, each tunnel is associated with an allowed rate for traffic on that tunnel, implemented as a rate limiter. The receive end of the tunnel monitors traffic and sends congestion feedback back to the sender. A bandwidth allocator corresponding to each entity uses feedback from all of the entity's tunnels to adapt the allowed rate on each tunnel. The bandwidth allocators take the network weights as parameters, work independently of each other, and together ensure that the network allocations converge to their desired values.

The Seawall shim layer is deployed to all servers in the data center by the management software that is responsible for provisioning and monitoring these servers (e.g., Autopilot, Azure Fabric). To ensure that only traffic controlled by Seawall enters the network, a provider can use attestation-based 802.1x authentication to disallow servers without the shim from connecting to the network.

The feedback to the control loop is returned at regular intervals, spaced T apart. It includes both explicit control signals from the receivers as well as congestion feedback about the path. Using the former, a receiver can explicitly block or rate-limit unwanted traffic. Using the latter, the bandwidth allocators adapt allowed rate on the tunnels. To help the receiver prepare congestion feedback, the shim at the sender maintains a byte sequence number per tunnel (i.e., per (sending entity, destination) pair). The sender shim stamps outgoing packets with the corresponding tunnel's current sequence number. The receiver detects losses in the same way as TCP, by looking for gaps in the received sequence number space. At the end of an interval, the receiver issues feedback that reports the number of bytes received and the percentage of bytes deemed to be lost (Figure 6). Optionally, if ECN is enabled along the network path, the feedback also relays the fraction of packets received with congestion marks.

We show efficient ways of stamping packets without adding a header and implementing queues and rate limiters in §5. Here, we describe the bandwidth allocator.

.**Begin** (weight W)
{ rate $r \leftarrow I$, weight $w \leftarrow W$ } ▷ Initialize
.**TakeFeedback** (feedback f, proportion p)
{
if feedback f indicates loss **then**
  $r \leftarrow r - r * \alpha * p$ ▷ Multiplicative Decrease
else
  $r \leftarrow r + w * p$ ▷ Weighted Additive Increase
end if
}

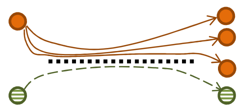
Class 1: A Strawman Bandwidth Allocator: an instance of this class is associated with each (entity, tunnel) pair.

4.2 Strawman

Consider the strawman bandwidth allocator in Class 1. Recall that the goal of the bandwidth allocator is to control the entity's network allocation as per the entity's network weight. Apart from the *proportion* variable, which we'll ignore for now, Class 1 is akin to weighted additive increase, multiplicative decrease (AIMD). It works as follows: when feedback indicates loss, it multiplicatively decreases the allowed rate by α. Otherwise, the rate increases by an additive constant.

This simple strawman satisfies some of our requirements. By making the additive increase step size a function of the entity's weight, the equilibrium rate achieved by an entity will be proportional to its weight. Unused shares are allocated to tunnels that have pent up demand, favoring efficiency over strict reservations. Global coordination is not needed. Further, when weights change, rates re-converge quickly (within one sawtooth period).

We derive the distributed control loop in Class 1 from TCP-Reno though any other flow-oriented protocol [4, 1, 29, 32] can be used, so long as it can extend to provide weighted allocations, as in MulTCP or MPAT [11, 39]. Distributed control loops are sensitive to variation in RTT. However, Seawall avoids this by using a constant feedback



Allocation Strategy	Share of Bottleneck for orange entity
Pair-wise	∝ to # of destinations
Seawall	0.5

Figure 7: When entities talk to different numbers of destinations, pair-wise allocation of bandwidth is not sufficient. Reduce tasks behave like the orange entity while maps resemble the green. (Assume that both orange and green entities have the same weight.)

period T, chosen to be larger than the largest RTT of the intra datacenter paths controlled by Seawall. Conservatively, Seawall considers no feedback within a period of T as if a feedback indicating loss was received.

Simply applying AIMD, or any other distributed control loop, on a per-tunnel basis does not achieve the desired per-link bandwidth distribution. Suppose a tenant has N VMs and opens flows between every pair of VMs. This results in a tunnel between each VM; with one AIMD loop per tunnel, thus each VM achieves $O(N)$ times its allocation at the bottleneck link. Large tenants can overwhelm smaller tenants, as shown in Figure 7.

Seawall improves on this simple strawman in three ways. First, it has a unique technique to combine feedback from multiple destinations. By doing so, an entity's share of the network is governed by its network weight and is independent of the number of tunnels it uses (§4.3). The resulting policy is consistent with how cloud providers allocate other resources, such as compute and memory, to a tenant, yet is a significant departure from prior approaches to network scheduling. Second, the sawtooth behavior of AIMD leads to poor convergence on paths with high bandwidth-delay product. To mitigate this, Seawall modifies the adaptation logic to converge quickly and stay at equilibrium longer (§4.4). Third, we show how to nest traffic with different levels of responsiveness to congestion signals (e.g., TCP vs. UDP) within Seawall (§4.5).

4.3 Seawall's Bandwidth Allocator

The bandwidth allocator, associated with each entity, takes as input the network weight of that entity, the congestion feedback from all the receivers that the entity is communicating with and generates the allowed rate on each of the entity's tunnels. It has two parts: a distributed congestion control loop that computes the entity's cumulative share on each link and a local scheduler that divides that share among the various tunnels.

Step 1: Use distributed control loops to determine per-link, per-entity share. The ideal feedback would be per-link. It would include the cumulative usage of the entity across all the tunnels on this link, the total load on the link, and the network weights of all the entities using that link. Such feedback is possible if switches implement explicit feedback (e.g., XCP, QCN) or from programmable switch sampling (e.g., SideCar [38]). Lacking these, the baseline Seawall relies only on existing congestion signals such as end-to-end losses or ECN marks. These signals identify congested paths, rather than links.

To approximate link-level congestion information using path-level congestion signals, Seawall uses a heuristic based on the observation that a congested link causes losses in many tunnels using that link. The logic is described in Class 2. One instance of this class is associated with each entity and maintains separate per-link instances of the distributed control loop (rc_l). Assume for now that rc is implemented as per the strawman Class 1, though we will replace it with Class 3. The sender shim stores the feedback from each destination, and once every period T, applies all the feedback cumulatively (lines 8–10). The heuristic scales the impact of feedback from a given destination in proportion to the volume of traffic sent to that destination by the shim in the last period (line 7, 10).

To understand how this helps, consider the example in Figure 7. An instance of class 2, corresponding to the orange entity, cumulatively applies the feedback from all three destinations accessed via the bottleneck link to the single distributed control loop object representing that link. Since the proportions sum up to 1 across all destinations, the share of the orange entity will increase by only so much as that of the green entity.

A simplification follows because the shim at the receiver reports the fraction of bytes lost or marked. Hence, rather than invoking the distributed control loop once per destination, Class 2 computes just three numbers per link – the proportions of total feedback indicating loss, ECN marks, and neither, and invokes the distributed control loop once with each.

1: .**Begin** (weight W)
2: { rc_l.Begin(W) $\forall$ links l used by sender } ▷ Initialize
3: .**TakeFeedback** (feedback f_{dest})
4: { store feedback }
5: .**Periodically** ()
6: {
7: proportion of traffic to d, $p_d = \frac{f_d.bytesRcvd}{\sum f_i.bytesRcvd}$
8: **for all** destinations d **do**
9: **for all** links $l \in PathTo(d)$ **do**
10: rc_l.TakeFeedback(f_d, p_d)
11: **end for**
12: **end for**
13: ▷ rc_l now contains per-link share for this entity
14: $n_l \leftarrow$ count of dest with paths through link l
15: ▷ r_d is allowed rate to d
16: $r_d \leftarrow \min_{l \in PathTo(d)} \left(\left(\beta p_d + \frac{1-\beta}{n_l} \right) rc_l.rate \right)$
17: }

Class 2: Seawall's bandwidth allocator: A separate instance of this class is associated with each entity. It combines per-link distributed control loops (invoked in lines 2, 10) with a local scheduler (line 16).

Step 2: Convert per-link, per-entity shares to per-link, per-tunnel shares. Next, Seawall runs a local allocator to assign rate limits to each tunnel that respects the entity's per-link rate constraints. A naïve approach divides each link's allowed rate evenly across all downstream destinations. For the example in Fig. 7, this leads to a $\frac{1}{3}'rd$ share of the bottleneck link to the three destinations of the orange entity. This leads to wasted bandwidth if the demands across destinations vary. For example, if the orange entity has demands $(2x, x, x)$ to the three destinations and the bottleneck's share for this entity is $4x$, dividing evenly causes the first destination to get no more than $\frac{4x}{3}$ while bandwidth goes wasted. Hence, Seawall apportions link bandwidth to destinations as shown in line 16, Class 2. The intuition is to adapt the allocations to match the demands. Seawall uses an exponential moving average that allocates β fraction of the link bandwidth proportional to current usage and the rest evenly across destinations. By default, we use $\beta = .9$. Revisiting the $(2x, x, x)$ example, note that while the first destination uses up all of its allowed share, the other two destinations do not, causing the first to get a larger share in the next period. In fact, the allowed share of the first destination converges to within 20% of its demand in four iterations.

Finally, Seawall converts these per-link, per-destination rate limits to a tunnel (i.e., per-path) rate limit by computing the minimum of the allowed rate on each link on the path. Note that Class 2 converges to a lower bound on the per-link allowed rate. At bottleneck links, this is tight. At other links, such as those used by the green flow in Figure 7 that are not the bottleneck, Class 2 can under-estimate their usable rate. Only when the green entity uses these other links on paths that do not overlap with the bottleneck, will the usable rate on those links increase. This behavior is the best that can be done using just path congestion signals and is harmless since the rate along each tunnel, computed as the minimum along each link on that path, is governed by the bottleneck.

4.4 Improving the Rate Adaptation Logic

Weighted AIMD suffers from inefficiencies as adaptation periods increase, especially for paths with high bandwidth-delay product [23] such as those in datacenters. Seawall uses control laws from CUBIC [32] to achieve faster convergence, longer dwell time at the equilibrium point, and higher utilization than AIMD. As with weighted AIMD, Seawall modifies the control laws to support weights and to incorporate feedback from multiple destinations. If switches support ECN, Seawall also incorporates the control laws from DCTCP [4] to further smooth out the sawtooth and reduce queue utilization at the bottleneck, resulting in reduced latency, less packet loss, and improved resistance against incast collapse.

The resulting control loop is shown in Class 3; the stability follows from that of CUBIC and DCTCP. Though we describe a rate-based variant, the equivalent window based versions are feasible and we defer those to future

1: .**Begin** (weight W)
2: { rate $r \leftarrow I$, weight $w \leftarrow W$, $c \leftarrow 0$, $inc \leftarrow 0$ } ▷ Init
3: .**TakeFeedback** (feedback f, proportion p)
4: {
5: $c \leftarrow c + \gamma * p * (f.bytesMarked - c)$
6: ▷ maintain smoothed estimate of congestion
7: **if** $f.bytesMarked > 0$ **then**
8: $r_{new} \leftarrow r - r * \alpha * p * c$ ▷ Smoothed mult. decrease
9: $inc \leftarrow 0$
10: $t_{lastdrop} \leftarrow now$
11: $r_{goal} \leftarrow (r > r_{goal})?r : \frac{r+r_{new}}{2}$
12: **else** ▷ Increase rate
13: **if** $r < r_{goal}$ **then** ▷ Less than goal, concave increase
14: $\Delta t = \min\left(\frac{now - t_{lastdrop}}{T_s}, .9\right)$
15: $\Delta r = \delta * (r_{goal} - r) * (1 - \Delta t)^3$
16: $r \leftarrow r + w * p * \Delta r$
17: **else** ▷ Above goal, convex increase
18: $r \leftarrow r + p * inc$
19: $inc \leftarrow inc + w * p$
20: **end if**
21: **end if**
22: }

Class 3: Seawall's distributed control loop: an instance of this class is associated with each (link, entity) pair. Note that Class 2 invokes this loop (lines 2, 10).

work. We elaborate on parameter choices in §4.6. Lines 14-17 cause the rate to increase along a concave curve, i.e., quickly initially and then slower as rate nears r_{goal}. After that, lines 18-19 implement convex increase to rapidly probe for a new rate. Line 5 maintains a smoothed estimate of congestion, allowing multiplicative decreases to be modulated accordingly (line 8) so that the average queue size at the bottleneck stays small.

4.5 Nesting Traffic Within Seawall

Nesting traffic of different types within Seawall's congestion-controlled tunnels leads to some special cases. If a sender always sends less than the rate allowed by Seawall, she may never see any loss causing her allowed rate to increase to infinity. This can happen if her flows are low rate (e.g., web traffic) or are limited by send or receive windows (flow control). Such a sender can launch a short overwhelming burst of traffic. Hence, Seawall clamps the rate allowed to a sender to a multiple of the largest rate she has used in the recent past. Clamping rates is common in many control loops, such as XCP [23], for similar reasons. The specific choice of clamp value does not matter as long as it is larger than the largest possible bandwidth increase during a Seawall change period.

UDP and TCP flows behave differently under Seawall. While a full burst UDP flow immediately uses all the rate that a Seawall tunnel allows, a set of TCP flows can take several RTTs to ramp up; the more flows, the faster the ramp-up. Slower ramp up results in lower shares on average. Hence, Seawall modifies the network stack to defer congestion control to Seawall's shim layer. All other

TCP functionality, such as flow control, loss recovery and in order delivery remain as before.

The mechanics of re-factoring are similar to Congestion Manager (CM) [7]. Each TCP flow queries the appropriate rate limiter in the shim (e.g., using shared memory) to see whether a send is allowed. Flows that have a backlog of packets register callbacks with the shim to be notified when they can next send a packet. In virtualized settings, the TCP stack defers congestion control to the shim by expanding the paravirtualized NIC interface. Even for tenants that bring their own OSes, the performance gain from refactoring the stack incentivizes adoption. Some recent advances in designing device drivers [36] reduce the overhead of signaling across the VM boundary. However, Seawall uses this simplification that requires less signaling: using hypervisor IPCs, the shim periodically reports a maximum congestion window to each VM to use for all its flows. The max congestion window is chosen large enough that each VM will pass packets to the shim yet small enough to not overflow the queues in front of the rate limiters in the shim.

We believe that deferring congestion control to the Seawall shim is necessary in the datacenter context. Enforcing network shares at the granularity of a flow no longer suffices (see §2). Though similar in spirit to Congestion Manager, Seawall refactors congestion control for different purposes. While CM does so to share congestion information among flows sharing a path, Seawall uses it to ensure that the network allocation policy holds regardless of the traffic mix. In addition, this approach allows for transparent changes to the datacenter transport.

4.6 Discussion

Here, we discuss details deferred from the preceding description of Seawall.

Handling WAN traffic: Traffic entering and leaving the datacenter is subject to more stringent DoS scrubbing at pre-defined chokepoints and, because WAN bandwidth is a scarce resource, is carefully rate-limited, metered and billed. We do not expect Seawall to be used for such traffic. However, if required, edge elements in the datacenter, such as load balancers or gateways, can funnel all incoming traffic into Seawall tunnels; the traffic then traverses a shim within the edge element. Traffic leaving the data center is handled analogously.

Mapping paths to links: To run Seawall, each sender requires path-to-link mapping for the paths that it is sending traffic on (line 10, Class 2). A sender can acquire this information independently, for example via a few traceroutes. In practice, however, this is much easier. Data center networks are automatically managed by software that monitors and pushes images, software and configuration to every node [19, 28]. Topology changes (e.g., due to failures and reconfiguration) are rare and can be disseminated automatically by these systems. Many pieces of today's datacenter ecosystem use topology information (e.g., Map-Reduce schedulers [27] and VM placement algorithms). Note that Seawall does work with a partial mapping (e.g., a high level mapping of each server to its rack, container, VLAN and aggregation switch) and does not need to identify bottleneck links. However, path-to-link mapping is a key enabler; it lets Seawall run over any datacenter network topology.

Choosing network weights: Seawall provides several ways to define the sending entity and the corresponding network weight. The precise choices depend on the datacenter type and application. When VMs are spun up in a cloud datacenter, the fabric sets the network weight of that VM alongside weights for CPU and memory. The fabric can change the VMs weight, if necessary, and Seawall re-converges rapidly. However, a VM cannot change its own weight. The administrator of a cloud datacenter can assign equal weights to all VMs, thereby avoiding performance interference, or assign weights in proportion to the size or price of the VM.

In contrast, the administrator of a platform datacenter can empower trusted applications to adjust their weights at run-time (e.g., via setsockopt()). Here, Seawall can also be used to specify weights per executable (e.g., background block replicator) or per process or per port ranges. The choice of weights could be based on information that the cluster schedulers have. For example, a map-reduce scheduler can assign the weight of each sender feeding a task in inverse proportion to the aggregation fan-in of that task, which he knows before hand. This ensures that each task obtains the same bandwidth (§2.2). Similarly, the scheduler can boost the weight of outlier tasks that are starved or are blocking many other tasks [6], thereby improving job completion times.

Enforcing global allocations: Seawall has so far focused on enforcing the network share of a local entity (VM, task etc.). This is complementary to prior work on Distributed Rate Limiters (DRL) [31] that controls the aggregate rate achieved by a collection of entities. Controlling just the aggregate rate is vulnerable to DoS: a tenant might focus the traffic of all of its VMs on a shared service (such as storage) or link (e.g., ToR containing victim tenant's servers), thereby interfering with the performance of other tenants while remaining under its global bandwidth cap. Combining Seawall with a global allocator such as DRL is simple. The Seawall shim reports each entity's usage to the global controller in DRL, which employs its global policy on the collection of entities and determines what each entity is allowed to send. The shim then caps the rate allowed to that entity to the minimum of the rate allowed by Seawall and the rate allowed by DRL's global policy. Further, the combination lets DRL scale better, since with Seawall, DRL need only track per-entity usage and not

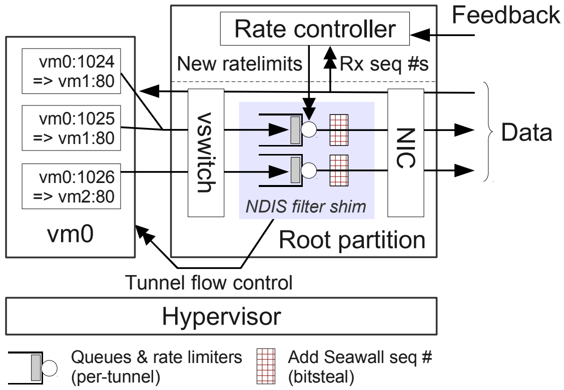


Figure 8: The Seawall prototype is split into an in-kernel NDIS filter shim (shaded gray), which implements the rate limiting datapath, and a userspace rate adapter, which implements the control loop. Configuration shown is for infrastructure data centers.

per-flow state that it would otherwise have to.

Choosing parameters: Whenever we adapt past work, we follow their guidance for parameters. Of the parameters unique to Seawall, their specific values have the following impact. We defer a formal analysis to future work. Reducing the feedback period T makes Seawall's adaptation logic more responsive at the cost of more overhead. We recommend choosing $T \in [10, 50]\ ms$. The multiplicative factor α controls the decrease rate. With the CUBIC/DCTCP control loop (see Class 3), Seawall is less sensitive to α than the AIMD control loop, since the former ramps back up more aggressively. In Class 2, β controls how much link rate is apportioned evenly versus based on current usage. With a larger β, the control loop reacts more quickly to changing demands but delays apportioning unused rate to destinations that need it. We recommend $\beta > .8$.

5. Seawall PROTOTYPE

The shim layer of our prototype is built as an NDIS packet filter (Figure 8). It interposes new code between the TCP/IP stack and the NIC driver. In virtualized settings, the shim augments the vswitch in the root partition. Our prototype is compatible with deployments that use the Windows 7 kernel as the server OS or as the root partition of Hyper-V. The shim can be adapted to other OSes and virtualization technologies, e.g., to support Linux and Xen, one can reimplement it as a Linux network queuing discipline module. For ease of experimentation, the logic to adapt rates is built in user space whereas the filters on the send side and the packet processing on the receive side are implemented in kernel space.

Clocking rate limiters: The prototype uses software-based token bucket filters to limit the rate of each tunnel. Implementing software rate limiters that work correctly and efficiently at high rates (e.g., 100s of Mbps) requires high precision interrupts; which are not widely available to drivers. Instead, we built a simple high precision clock. One core, per rack of servers, stays in a busy loop, and broadcasts a UDP heartbeat packet with the current time to all the servers within that rack once every 0.1ms; the shim layers use these packets to clock their rate limiters. We built a roughly equivalent window-based version of the Seawall shim as proof-of-concept. Windowing is easier to engineer, since it is self-clocking and does not require high precision timers, but incurs the expense of more frequent feedback packets (e.g., once every 10 packets).

Bit-stealing and stateless offload compatibility: A practical concern is the need to be compatible with NIC offloads. In particular, adding an extra packet header to support Seawall prevents the use of widely-used NIC offloads, such as large send offload (LSO) and receive side coalescing (RSC) which only work for known packet formats such as UDP or TCP. This leads to increased CPU overhead and decreased throughput. On a quad core 2.66 Intel Core2 Duo with an Intel 82567LM NIC, sending at the line rate of 1Gbps requires 20% more CPU without LSO (net: 30% without vs 10% with LSO) [37].

NIC vendors have plans to improve offload support for generic headers. To be immediately deployable without performance degradation, Seawall *steals* bits from existing packet headers, that is, it encodes information in parts of the packet that are unused or predictable and hence can be restored by the shim at the receiver. For both UDP and TCP, Seawall uses up to 16 bits from the IP ID field, reserving the lower order bits for the segmentation hardware if needed. For TCP packets, Seawall repurposes the timestamp option: it compresses the option Kind and Length fields from 16 bits down to 1 bit, leaving the rest for Seawall data. In virtualized environments, guest OSes are para-virtualized to always include timestamp options. The feedback is sent out-of-band in separate packets. We also found bit-stealing easier to engineer than adding extra headers, which could easily lead to performance degradation unless buffers were managed carefully.

Offloading rate limiters and direct I/O: A few emerging standards to improve network I/O performance, such as Direct I/O and SR-IOV, let guest VMs bypass the virtual switch and exchange packets directly with the NIC. But, this also bypasses the Seawall shim. Below, we propose a few ways to restore compatibility. However, we note that the loss of the security and manageability features provided by the software virtual switch has limited the deployment of direct I/O NICs in public clouds. To encourage deployment, vendors of such NICs plan to support new features specific to datacenters.

By offloading token bucket- and window-based limiters from the virtual switch to NIC or switch hardware, tenant traffic can be controlled even if guest VMs directly send packets to the hardware. To support Seawall, such offloaded rate limiters need to provide the same granularity of flow classification (entity to entity tunnels) as the shim and report usage and congestion statistics. High end NICs that support stateful TCP, iSCSI, and RDMA offloads already support tens of thousands to millions of window-control engines in hardware. Since most such NICs are programmable, they can likely support the changes needed to return statistics to Seawall. Switch *policers* have similar scale and expressiveness properties. In addition, low cost programmable switches can be used to monitor the network for violations [38]. Given the diversity of implementation options, we believe that the design point occupied by Seawall, i.e., using rate- or window-controllers at the network edge, is feasible now and as data rates scale up.

6. EVALUATION

We ran a series of experiments using our prototype to show that Seawall achieves line rate with minimal CPU overhead, scales to typical data centers, converges to network allocations that are agnostic to communications pattern (i.e., number of flows and destinations) and protocol mix (i.e., UDP and TCP), and provides performance isolation. Through experiments with web workloads, we also demonstrate how Seawall can protect cloud-hosted services against DoS attacks, even those using UDP floods.

All experiments used the token bucket filter-based shim (i.e., rate limiter), which is our best-performing prototype and matches commonly-available hardware rate limiters. The following hold unless otherwise stated: (1) Seawall was configured with the default parameters specified in §4, (2) all results were aggregated from 10 two minute runs, with each datapoint a 15 second average and error bars indicating the 95% confidence interval.

Testbed: For our experiments, we used a 60 server cluster spread over three racks with 20 servers per rack. The physical machines were equipped with Xeon L5520 2.27 GHz CPUs (quad core, two hyperthreads per core), Intel 82576 NICs, and 4GB of RAM. The NIC access links were 1Gb/s and the links from the ToR switches up to the aggregation switch were 10Gb/s. There was no oversubscription within each rack. The ToR uplinks were 1:4 over-subscribed. We chose this topology because it is representative of typical data centers.

For virtualization, we use Windows Server 2008R2 Hyper-V with Server 2008R2 VMs. This version of Hyper-V exploits the Nehalem virtualization optimizations, but does not use the direct I/O functionality on the NICs. Each guest VM was provisioned with 1.5 GB of RAM and 4 virtual CPUs.

6.1 Microbenchmarks

6.1.1 Throughput and overhead

To evaluate the performance and overhead of Seawall, we measured the throughput and CPU overhead of tunneling a TCP connection between two machines through the shim. To minimize extraneous sources of noise, no other traffic was present in the testbed during each experiment and the sender and receiver transferred data from and to memory.

	Throughput (Mb/s)	CPU @ Sender (%)	CPU @ Receiver (%)
Seawall	947 ± 9	20.7 ± 0.6	14.2 ± 0.4
NDIS	977 ± 4	18.7 ± 0.4	13.5 ± 1.1
Baseline	979 ± 6	16.9 ± 1.9	10.8 ± 0.8

Table 1: CPU overhead comparison of Seawall, a null NDIS driver, and an unmodified network stack. Seawall achieved line rate with low overhead.

Seawall achieved nearly line rate at steady state, with negligible increase in CPU utilization, adding 3.8% at the sender and 3.4% at the receiver (Table 1). Much of this overhead was due to the overhead from installing a NDIS filter driver: the null NDIS filter by itself added 1.8% and 2.7% overhead, respectively. The NDIS framework is fairly light weight since it runs in the kernel and requires no protection domain transfers.

Subtracting out the contributions from the NDIS filter driver reveals the overheads due to Seawall: it incurred slightly more overhead on the sender than the receiver. This is expected since the sender does more work: on receiving packets, a Seawall receiver need only buffer congestion information and bounce it back to the sender, while the sender incurs the overhead of rate limiting and may have to merge congestion information from many destinations.

Seawall easily scales to today's data centers. The shim at each node maintains a rate limiter, with a few KBs of state each, for every pair of communicating entities terminating at that node. The per-packet cost on the data path is fixed regardless of data center size. A naive implementation of the rate controller incurs O(DL) complexity per sending entity (VM or task) where D is the number of destinations the VM communicates with and L is the number of links on paths to those destinations. In typical data center topologies, the diameter is small, and serves as an upper bound for L. All network stacks on a given node have collective state and processing overheads that grow at least linearly with D; these dominate the corresponding contributions from the rate controller and shim.

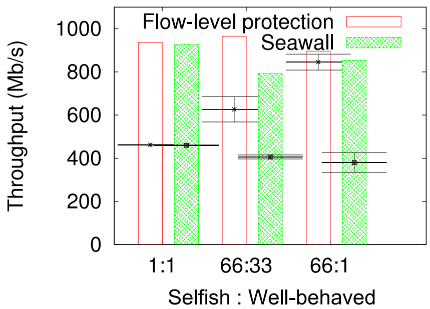
6.1.2 Traffic-agnostic network allocation

Seawall seeks to control the network share obtained by a sender, regardless of traffic. In particular, a sender should not be able to attain bandwidth beyond that allowed by the configured weight, no matter how it varies protocol type, number of flows, and number of destinations.

To evaluate the effectiveness of Seawall in achieving this goal, we set up the following experiment. Two physical nodes, hosting one VM each, served as the sources, with one VM dedicated to selfish traffic and the other to well-behaved traffic. One physical node served as the sink for

	TCP victim throughput (Mb/s)
Seawall	429.76
No protection	1.49

(a) Full Burst UDP



(b) Many TCP Flows

Figure 9: Seawall ensures that despite using full burst UDP flows or many TCP flows, the share of a selfish user is held proportional to its weight. (In (b), the bars show total throughput, with the fraction below the divider corresponding to selfish traffic and the fraction above corresponding to well-behaved traffic.)

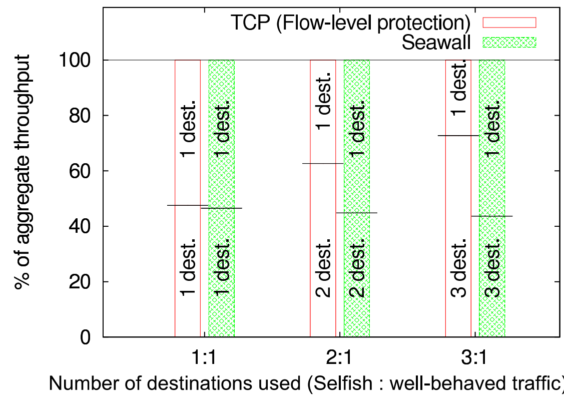


Figure 10: By combining feedback from multiple destinations, Seawall ensures that the share of a sender remains independent of the number of destinations it communicates with. (The fraction of the bar below the divider corresponds to the fraction of bottleneck throughput achieved by selfish traffic.)

all traffic; it was configured with two VMs, with one VM serving as the sink for well-behaved traffic and the other serving as the sink for selfish traffic.

Both well-behaved and selfish traffic used the same number of source VMs, with all Seawall senders assigned the same network weight. The well-behaved traffic consisted of a single long-lived TCP flow from each source, while the selfish traffic used one of three strategies to achieve a higher bandwidth share: using full burst UDP flow, using large numbers of TCP flows, and using many destinations

Selfish traffic = Full-burst UDP: Figure 9(a) shows the aggregate bandwidth achieved by the well-behaved traffic (long-lived TCP) when the selfish traffic consisted of full rate UDP flows. The sinks for well-behaved and selfish traffic were colocated on a node with a single 1Gbps NIC. Because each sender had equal weight, Seawall assigned half of this capacity to each sender. Without Seawall, selfish traffic overwhelms well-behaved traffic, leading to negligible throughput for well-behaved traffic. By bundling the UDP traffic inside a tunnel that imposed congestion control, Seawall ensured that well-behaved traffic retained reasonable performance.

Selfish traffic = Many TCP flows: Figure 9(b) shows the bandwidth shares achieved by selfish and well-behaved traffic when selfish senders used many TCP flows. As before, well-behaved traffic ideally should have achieved $\frac{1}{2}$ of the bandwidth. When selfish senders used the same number of flows as well-behaved traffic, bandwidth was divided evenly (left pair of bars). In runs without Seawall, selfish senders that used twice as many flows obtained $\frac{2}{3}$'rds the bandwidth because TCP congestion control divided bandwidth evenly across flows (middle pair of bars). Runs with Seawall resulted in approximately even bandwidth allocation. Note that Seawall achieved slightly lower throughput in aggregate. This was due to slower recovery after loss– the normal traffic had one sawtooth per TCP flow whereas Seawall had one per source VM; we believe this can be improved using techniques from §4. When the selfish traffic used 66 times more flows, it achieved a dominant share of bandwidth; the well-behaved traffic was allocated almost no bandwidth (rightmost pair of bars). We see that despite the wide disparity in number of flows, Seawall divided bandwidth approximately evenly. Again, Seawall improved the throughput of well-behaved traffic (the portion above the divider) by several orders of magnitude.

Selfish traffic = Arbitrarily many destinations: This experiment evaluated Seawall's effectiveness against selfish tenants that opened connections to many destinations. The experiment used a topology similar to that in Figure 7. A well-behaved sender VM and a selfish sender VM were located on the same server. Each sink was a VM and ran on a separate, dedicated machine. The well-behaved traffic was assigned one sink machine and the selfish traffic was assigned a variable number of sink machines. Both well-behaved and selfish traffic consisted of one TCP flow per sink. As before, the sending VMs were configured with the same weight, so that well-behaved traffic would achieve an even share of the bottleneck.

Figure 10 plots the fraction of bottleneck bandwidth achieved by well-behaved traffic with and without Seawall. We see that without Seawall, the share of the selfish traffic was proportional to the number of destinations. With Seawall, the share of the well-behaved traffic remained constant at approximately half, independent of the number

	Throughput (Mb/s)	Latency (s)
Seawall	181	0.61
No protection	157	0.91

Figure 11: Despite bandwidth pressure, Seawall ensures that the average HTTP request latency remains small without losing throughput.

of destinations.

6.2 Performance isolation for web servers

To show that Seawall protects against performance interference similar to that shown in §2, we evaluated the achieved level of protection against a DoS attack on a web server. Since cloud datacenters are often used to host web-accessible services, this is a common use case.

In this experiment, an attacker targeted the HTTP responses sent from the web server to its clients. To launch such attacks, an adversary places a source VM and a sink VM such that traffic between these VMs crosses the same bottleneck links as the web server. The source VM is close to the server, say on the same rack or machine, while the sink VM is typically on another rack. Depending on where the sink is placed, the attack can target the ToR uplink or another link several hops away.

All machines were colocated on the same rack. The web server VM, running Microsoft IIS 7, and attacker source VM, generating UDP floods, resided in separate, dedicated physical machines. A single web client VM requested data from the server and shared a physical machine with an attacker sink VM. The web clients used WcAsync to generate well-formed web sessions. Session arrivals followed a Poisson process and were exponentially sized with a mean of 10 requests. Requests followed a WebStone distribution, varying in size from 500B responses to 5MB responses with smaller files being much more popular.

As expected, a full-rate UDP attack flood caused congestion on the access link of the web client, reducing throughput to close to zero and substantially increasing latency. With Seawall, the web server behaved as if there were no attack. To explore data points where the access link was not overwhelmed, we dialed down the UDP attack rate to 700Mbps, enough to congest the link but not to stomp out the web server's traffic. While achieving roughly the same throughput as in the case of no protection, Seawall improved the latency observed by web traffic by almost 50% (Figure 11). This is because sending the attack traffic through a congestion controlled tunnel ensured that the average queue size at the bottleneck stays small, thereby reducing queuing delays.

7. DISCUSSION

Here, we discuss how Seawall can be used to implement rich cloud service models that provide bandwidth guarantees to tenants, the implications of our architectural decisions given trends in data centers and hardware, and the benefits of jointly modifying senders and receivers to achieve new functionality in data center networks.

7.1 Sharing policies

Virtual Data Centers (VDCs) have been proposed [20, 17, 40] as a way to specify tenant networking requirements in cloud data centers. VDCs seek to approximate, in terms of security isolation and performance, a dedicated data center for each tenant and allows tenants to specify SLA constraints on network bandwidth at per-port and per-source/dest-pair granularities. When allocating tenant VMs to physical hardware, the data center fabric simultaneously satisfies the specified constraints while optimizing node and network utilization.

Though Seawall policies could be seen as a simpler-to-specify alternative to VDCs that closely matches the provisioning knobs (e.g., disk, CPU, and memory size) of current infrastructure clouds, Seawall's weight-based policies can enhance VDCs in several ways. Some customers, through analysis or operational experience, understand the traffic requirements of their VMs; VDCs are attractive since they can exploit such detailed knowledge to achieve predictable performance. To improve VDCs with Seawall, the fabric uses weights to implement the hard bandwidth guarantees specified in the SLA: with appropriate weights, statically chosen during node- and path-placement, Seawall will converge to the desired allocation. Unlike implementations based on static reservations [17], the Seawall implementation is work-conserving, max-min fair, and achieves higher utilization through statistical multiplexing.

Seawall also improves a tenant's control of its own VDC. Since Seawall readily accepts dynamic weight changes, each tenant can adjust its allocation policy at a fine granularity in response to changing application needs. The fabric permits tenants to reallocate weights between different tunnels so long as the resulting weight does not exceed the SLA; this prevents tenants from stealing service and avoids having to rerun the VM placement optimizer.

7.2 System architecture

Topology assumptions: The type of topology and available bandwidth affects the complexity requirements of network sharing systems. In full bisection bandwidth topologies, congestion can only occur at the core. System design is simplified [44, 40, 30], since fair shares can be computed solely from information about edge congestion, without any topology information or congestion feedback from the core.

Seawall supports general topologies, allowing it to provide benefits even in legacy or cost-constrained data centers networks. Such topologies are typically bandwidth-constrained in the core; all nodes using a given core link need to be accounted for to achieve fair sharing, bandwidth reservations, and congestion control. Seawall explicitly uses topology information in its control layer to prevent link over-utilization.

Rate limiters and control loops: Using more rate limiters enables a network allocation system to support richer, more granular policies. Not having enough rate limiters can result in aliasing. For instance, VM misbehavior can cause Gatekeeper [40] to penalize unrelated VMs sending to the same destination. Using more complex rate limiters can improve system performance. For instance, rate limiters based on multi-queue schedulers such as DWRR or Linux's hierarchical queuing classes can utilize the network more efficiently when rate limiter parameters and demand do not match, and the self-clocking nature of window-based limiters can reduce switch buffering requirements as compared to rate-based limiters. However, having a large number of complex limiters can constrain how a network sharing architecture can be realized, since NICs and switches do not currently support such rate limiters at scale.

To maximize performance and policy expressiveness, a network allocation system should support a large number of limiters of varying capability. The current Seawall architecture can support rate- and window-based limiters based in hardware and software. As future work, we are investigating ways to map topology information onto hierarchical limiters; to compile policies given a limited number of available hardware limiters; and to tradeoff rate limiter complexity with controller complexity, using longer adaptation intervals when more capable rate limiters are available.

7.3 Partitioning sender/receiver functionality

Control loops can benefit from receiver-side information and coordination, since the receiver is aware of the current traffic demand from all sources and can send feedback to each with lower overhead. Seawall currently uses a receiver-driven approach customized for map-reduce to achieve better network scheduling; as future work we are building a general solution at the shim layer.

In principal, a purely receiver-directed approach to implementing a new network allocation policy, such as that used in [44, 40], might reduce system complexity since the sender TCP stack does not need to be modified. However, virtualization stack complexity does not decrease substantially, since the rate controller simply moves from the sender to the receiver. Moreover, limiting changes to one endpoint in data centers provides little of the adoption cost advantages found in the heterogeneous Internet environment. Modifying the VMs to defer congestion control to other layers can help researchers and practitioners to identify and deploy new network sharing policies and transport protocols for the data center.

A receiver-only approach can also *add* complexity. While some allocation policies are easy to attain by treating the sender as a black box, others are not. For instance, eliminating fatesharing from Gatekeeper and adding weighted, fair work-conserving scheduling appears non-trivial. Moreover, protecting a receiver-only approach from attack requires adding a detector for non-conformant senders. While such detectors have been studied for WAN traffic [13], it is unclear whether they are feasible in the data center. Such detectors might also permit harmful traffic that running new, trusted sender-side code can trivially exclude.

8. RELATED WORK

Proportional allocation of shared resources has been a recurring theme in the architecture and virtualization communities [42, 15]. To the best of our knowledge, Seawall is the first to extend this to the data center network and support generic sending entities (VMs, applications, tasks, processes, etc.).

Multicast congestion control [14], while similar at first blush, targets a very different problem since they have to allow for any participant to send traffic to the group while ensuring TCP-friendliness. It is unclear how to adapt these schemes to proportionally divide the network.

Recent work in hypervisor, network stack, and software routers have shown that software-based network processing, like that used in Seawall for monitoring and rate limiting, can be more flexible than hardware-based approaches yet achieve high performance. [35] presents an optimized virtualization stack that achieves comparable performance to direct I/O. The Sun Crossbow network stack provides an arbitrary number of bandwidth-limited virtual NICs [41]. Crossbow provides identical semantics regardless of underlying physical NIC and transparently leverages offloads to improve performance. Seawall's usage of rate limiters can benefit from these ideas.

QCN is an emerging Ethernet standard for congestion control in datacenter networks [29]. In QCN, upon detecting a congested link, the switch sends feedback to the heavy senders. The feedback packet uniquely identifies the flow and congestion location, enabling senders that receive feedback to rate limit specific flows. QCN uses explicit feedback to drive a more aggressive control loop than TCP. While QCN can throttle the heavy senders, it is not designed to provide fairness guarantees, tunable or otherwise. Further, QCN requires changes to switch hardware and can only cover purely Layer 2 topologies.

Much work has gone into fair queuing mechanisms in switches [12]. Link local sharing mechanisms, such as Weighted Fair Queuing and Deficit Round Robin, separate traffic into multiple queues at each switch port and arbitrate service between the queues in some priority or proportion. NetShare [24] builds on top of WFQ support in switches. This approach is useful to share the network between a small number of large sending entities (e.g., a whole service type, such as "Search" or "Distributed storage" in a platform data center). The number of queues available in today's switches, however, is several orders of magnitude smaller than the numbers of VMs and tasks in today's datacenters. More fundamentally, since link local mechanisms lack end-to-end information they can let significant traffic through only to be dropped at some later bottleneck on the path. Seawall can achieve better scalability by mapping many VMs onto a small, fixed number of queues and achieves better efficiency by using end-to-end congestion control.

9. FINAL REMARKS

Economies of scale are pushing distributed applications to co-exist with each other on shared infrastructure. The lack of mechanisms to apportion network bandwidth across these entities leads to a host of problems, from reduced security to unpredictable performance and to poor ability to improve high level objectives such as job completion time. Seawall is a first step towards providing data center administrators with tools to divide their network across the sharing entities without requiring any cooperation from the entities. It is novel in its ability to scale to massive numbers of sharing entities and uniquely adapts ideas from congestion control to the problem of enforcing network share agnostic to traffic type. The design space that Seawall occupies – push functionality to software at the network edge – appears well-suited to emerging hardware trends in data center and virtualization hardware.

Acknowledgements

We thank Deepak Bansal, Dave Maltz, our shepherd Bill Weihl and the NSDI reviewers for discussions that improved this work.

Notes

[1] Perhaps because it is hard to predict such events and find appropriate tasks at short notice. Also, running more tasks requires spare memory and has initialization overhead.

References

[1] *Understanding the Available Bit Rate (ABR) Service Category for ATM VCs*. Cisco Systems, 2006.
[2] A. Greenberg, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network. In *SIGCOMM*, 2009.
[3] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *SIGCOMM*, 2008.
[4] M. Alizadeh, A. Greenberg, D. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data Center TCP (DCTCP). In *SIGCOMM*, 2010.
[5] Amazon.com. Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2/.
[6] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the Outliers in MapReduce Clusters Using Mantri. In *OSDI*, 2010.
[7] H. Balakrishnan, H. Rahul, and S. Seshan. An integrated congestion management architecture for internet hosts. In *SIGCOMM*, 1999.
[8] Bill Claybrook. Comparing cloud risks and virtualization risks for data center apps. http://searchdatacenter.techtarget.com/tip/0,289483,sid80_gci1380652,00.html.
[9] R. Chaiken, B. Jenkins, P. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: Easy and Efficient Parallel Processing of Massive Datasets. In *VLDB*, 2008.
[10] L. Cherkasova, D. Gupta, and A. Vahdat. Comparison of the three CPU schedulers in Xen. In *SIGMETRICS*, 2007.
[11] J. Crowcroft and P. Oechslin. Differentiated end-to-end Internet services using a weighted proportional fair sharing TCP. *ACM CCR*, 28(3), 1998.
[12] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. *ACM CCR*, 19(4), 1989.
[13] S. Floyd and K. Fall. Promoting the use of end-to-end congestion control in the Internet. *IEEE TON*, 7(4), 1999.
[14] J. Golestani and K. Sabnani. Fundamental observations on multicast congestion control in the Internet. In *INFOCOM*, 1999.
[15] A. Gulati and C. A. Waldspurger. PARDA: Proportional Allocation of Resources for Distributed Storage Access. In *FAST*, 2009.
[16] C. Guo et al. Bcube: High performance, server-centric network architecture for data centers. In *SIGCOMM*, 2009.
[17] C. Guo, G. Lu, H. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang. SecondNet: A Data Center Network Virtualization Architecture with Bandwidth Guarantees. In *ACM CoNEXT*, 2010.
[18] M. Harchol-Balter, B. Schroeder, N. Bansal, and M. Agrawal. Size-based scheduling to improve web performance. *ACM TOCS*, 21(2), 2003.
[19] M. Isard. Autopilot: Automatic Data Center Management. *OSR*, 41(2), 2007.
[20] M. Kallahalla, M. Yusal, R. Swaminathan, D. Lowell, M. Wray, T. Christian, N. Edwards, C. I. Dalton, and F. Gittler. SoftUDC: A Software-Based Data Center for Utility Computing. *Computer*, 2004.
[21] S. Kandula, J. Padhye, and P. Bahl. Flyways to de-congest data center networks. In *HotNets*, 2009.
[22] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. The Nature of Datacenter Traffic: Measurements & Analysis. In *IMC*, 2009.
[23] D. Katabi, M. Handley, and C. Rohrs. Internet Congestion Control for High Bandwidth-Delay Product Networks. In *SIGCOMM*, 2002.
[24] T. Lam et al. NetShare : Virtualizing Data Center Networks across Services. Technical Report CS2010-0957, UCSD, 2010.
[25] A. Li, X. Yang, S. Kandula, and M. Zhang. Cloudcmp: Comparing public cloud providers. In *IMC*, 2010.
[26] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. In *SIGCOMM*, 2008.
[27] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, A. Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *SOSP*, 2009.
[28] Microsoft. An Overview of Windows Azure. http://download.microsoft.com/download/A/A/6/AA6A260A-B920-4BBC-AE33-8815996CD8FB/02-Article Introduction to Windows Azure.docx.
[29] R. Pan, B. Prabhakar, and A. Laxmikantha. QCN: Quantized Congestion Notification. http://www.ieee802.org/1/files/public/docs2007/au-prabhakar-qcn-description.pdf, 2007.
[30] L. Popa, S. Y. Ko, and S. Ratnasamy. CloudPolice: Taking Access Control out of the Network. In *HotNets*, 2010.
[31] B. Raghavan, K. Vishwanath, S. Ramabhadran, K. Yocum, and A. C. Snoeren. Cloud control with distributed rate limiting. In *SIGCOMM*, 2007.
[32] I. Rhee and L. Xu. CUBIC: A New TCP-Friendly High-Speed TCP Variant. *OSR*, 42(5), 2008.
[33] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *ACM CCS*, 2009.
[34] G. Rosen. Anatomy of an Amazon EC2 Resource ID. http://www.jackofallclouds.com/2009/09/anatomy-of-an-amazon-ec2-resource-id/.
[35] J. R. Santos, Y. Turner, G. J. Janakiraman, and I. Pratt. Bridging the Gap between Software and Hardware Techniques for I / O Virtualization. Technical Report HPL-2008-39, HP Labs, 2008.
[36] L. Shalev, J. Satran, E. Borovik, and M. Ben-yehuda. IsoStack Highly Efficient Network Processing on Dedicated Cores. In *USENIX ATC*.
[37] A. Shieh, S. Kandula, A. Greenberg, and C. Kim. Seawall: Performance isolation for cloud datacenter networks. In *HotCloud*, 2010.
[38] A. Shieh, S. Kandula, and E. Sirer. SideCar: Building Programmable Datacenter Networks without Programmable Switches. In *HotNets*, 2010.
[39] M. Singh, P. Pradhan, and P. Francis. MPAT: Aggregate TCP Congestion Management as a Building Block for Internet QoS. In *ICNP*, 2004.
[40] P. V. Soares, J. R. Santos, N. Tolia, D. Guedes, and Y. Turner. Gatekeeper: Distributed Rate Control for Virtualized Datacenters. Technical Report HPL-2010-151, HP Labs, 2010.
[41] S. Tripathi, N. Droux, T. Srinivasan, and K. Belgaied. Crossbow: from hardware virtualized NICs to virtualized networks. In *ACM VISA*, 2009.
[42] C. A. Waldspurger. *Lottery and Stride Scheduling : Proportional Share Resource Management*. PhD thesis, MIT, 1995.
[43] G. Wang and T. S. E. Ng. The Impact of Virtualization on Network Performance of Amazon EC2 Data Center. In *INFOCOM*, 2010.
[44] H. Wu, Z. Feng, C. Guo, and Y. Zhang. ICTCP: Incast Congestion Control for TCP in Data Center Networks. In *ACM CoNEXT*, 2010.
[45] M. Zaharia, A. Konwinski, A. Joseph, R. Katz, and I. Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *OSDI*, 2008.