Public Review for

NetShare and Stochastic NetShare: Predictable Bandwidth Allocation for Data Centers

Vinh The Lam, Sivasankar Radhakrishnan, Amin Vahdat, George Varghese, and Rong Pan

Tenants in datacenters desire performance isolation from each other. Such isolation for the network has been difficult to achieve without sacrificing utilization. This paper presents a set of techniques that together could achieve such isolation without requiring hardware changes in switches. The system is evaluated on a testbed of Fulcrum switches.

The techniques employed are as follows. On each switch, on each outbound link, a separate DRR queue is configured for each class of service. Tenants are clustered into these classes, and the weight of each class is the sum of the weights of the tenants. These weights are assigned by an operator when a tenant is provisioned. The traffic for each tenant is labeled so that it lands in the right queue. To handle UDP, each host needs a rate throttling shim. A centralized bandwidth allocator measures the rates of flows and then decides on new rates that are enforced using token bucket rate limiters at hosts or ingress switch ports.

There is a lot to absorb in this paper and the reviewers craved more details. One reviewer was concerned about how the system scales down to a small number of tenants because of a potential for bandwidth stealing, or how it scales to fast churn in tenants. Another was more concerned about the speed with which switch configurations could be updated. All the reviewers liked the paper. It is timely and the topic is important. The implementation on Fulcrum switches impressed them.

A general question worth pondering is what type of isolation the datacenter operator wants to offer, and what type tenants desire, and are those two in conflict? I suspect that one wants to offer proportional sharing of bandwidth, while the other wants minimum guaranteed bandwidths.

Public review written by
Sharad Agarwal
Microsoft Research

acm sigcomm

NetShare and Stochastic NetShare: Predictable Bandwidth Allocation for Data Centers

Vinh The Lam, Sivasankar Radhakrishnan, Amin Vahdat, and George Varghese
University of California, San Diego
{vtlam,sivasankar,vahdat,varghese}@cs.ucsd.edu

Rong Pan
Cisco Systems
ropan@cisco.com

ABSTRACT

Application performance in cloud data centers often depends crucially on network bandwidth, not just the aggregate data transmitted as in typical SLAs. We describe a mechanism for data center networks called *NetShare* that requires no hardware changes to routers but allows bandwidth to be allocated predictably across services based on weights. The weights are either specified by a manager, or automatically assigned at each switch port based on a virtual machine heuristic for isolation. Bandwidth unused by a service is shared proportionately by other services, providing weighted hierarchical max-min fair sharing. On a testbed of Fulcrum switches, we demonstrate that NetShare provides bandwidth isolation in various settings, including multipath networks.

Categories and Subject Descriptors

C.2 [**Computer-Communication Networks**]: Network Architecture and Design

Keywords

data center networks, bandwidth virtualization

1. INTRODUCTION

Cloud services and enterprise networks are hosted by data centers that concurrently support many distinct services — e.g., search and email for cloud services, or accounting and engineering for an enterprise data center. The services use a shared data center because physical equipment is expensive, costing over 100 M a year to maintain [4] and because statistical multiplexing using Virtual Machines (VMs) is effective. However, the economics also require two other characteristics of the *network*, both of which are imperfectly provided today. First, to be profitable, the networks must have high *utilization*. Second, many services have stringent performance service-level agreement (SLA) that must be met to keep customers satisfied; thus the network should also ideally provide *bandwidth guarantee* to each service. Any new mechanism to provide these should not require hardware changes to existing switches so that providers do not have to retrofit their networks.

Service-level agreements today specify network requirement in terms of dollar cost per Gigabyte transferred and not in terms of network bandwidth. But the performance of application frameworks such as MapReduce depends greatly on network performance. With current SLAs, a user may pay for 10 hours for a number of VMs only to find that the VMs are mostly idle waiting for slow network transfers. The user job may complete in one hour with a faster network and the user may be willing to pay more for the higher bandwidth. In addition, as cloud computing and shared data centers gain momentum, there is a growing demand to provide performance isolation between different services and tenants. While isolation can be achieved by strict rate limits, this leads to inefficient use of the expensive data center network because traffic is often bursty.

We propose a new mechanism for data center networks called *NetShare* that provides predictable bandwidth allocation, bandwidth isolation, and high utilization — and can be implemented without any changes to existing switches. NetShare does so using *hierarchical weighted max-min fair sharing* in which the bisection bandwidth of the network is first allocated to services according to weights and the bandwidth of each service is then allocated equally among its TCP connections. Hierarchical max-min fair sharing generalizes hierarchical fair sharing of *links* [7] to *networks*. We also generalize stochastic fair queuing [12] to stochastic weighted max-min fair queuing.

If Internet QoS did not succeed, why hope for data center QoS? First, Internet QoS issues are often solved by overprovisioning, but overprovisioning core links in data centers from say 10 Gbps to 40 Gbps is very expensive. Current core links are indeed oversubscribed [4]. Second, users have begun to notice latency degradation when VM traffic from different services [1] interfere [8]. Third, a reason for the failure of QoS was that there was no simple policy for setting QoS parameters. NetShare uses a simple set of per-service weights which can be set automatically based on VM placement, or set manually by a manager based on the revenue or cost of each service analogous to VMware's ESX server shares [8].

NetShare can also be viewed as a way to virtualize (i.e., statistically multiplex) a data center network among multiple services. Together with virtualized CPUs and disks, it allows managers to create "virtual data centers" with performance isolation. While one can argue whether our model of network virtualization is right, NetShare is perhaps the simplest starting point.

2. NETSHARE ALGORITHMS

The generalization of fair sharing to multiple resources such as a network is called Pareto Optimality (in economics) or max-min fair sharing (in networking). However, max-min fair sharing at the TCP level is not the appropriate model for sharing data center services. First, services that open up multiple connections get an unfair share of bandwidth. Second, the network administrator cannot allocate more bandwidth to certain services based on their importance.

In this paper, we propose generalizing the above connection-level fairness concept to services. In particular, the NetShare framework presents an abstraction for *service-level weighted hierarchical max-min fairness* as follows. First, the weights for different services are specified. The network demand for a particular VM

[1] We use the terms *application* and *service* interchangeably.

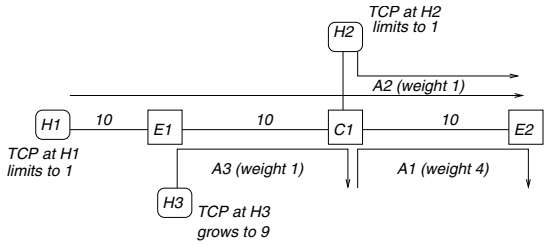


Figure 1: Simple fair queuing at switches at the *service* level together with TCP achieves *hierarchical* max-min fair sharing of services.

is a function of the application and can be set in the SLA policy, just as application developers specify CPU cores and memory requirement in each VM for the service. Next, there is a mechanism that allocates network bandwidth in weighted max-min fair fashion among these services. The bandwidth assigned to each service is then divided recursively (again in max-min fair fashion) among the individual flows for that service.

By adopting weighted max-min fair sharing, NetShare allocates bandwidth based on relative weights, and does not strictly guarantee minimum bandwidth to a service. Minimum bandwidth policies can be added to the NetShare controller (Section 2.4), but the controller would report no feasible solution after the rate computation if all policies could not be met with the existing VM assignment and traffic matrix. Hence we note that NetShare can be augmented with a VM placement and admission control strategy to guarantee a minimum bandwidth. The admission control agent should reject a request for VM instances when the requirement could not be met.

We present three mechanisms to implement NetShare. The first mechanism (Section 2.1) relies on TCP and fair queuing, only requires configuration changes, and responds to changes in a few round trip delays. We show how this can scale to a larger number of applications using the idea of *Stochastic NetShare* (Section 2.2). The second mechanism (Section 2.3) augments the first mechanism to handle UDP. Finally, the third mechanism (Section 2.4) uses centralized allocation to provide more general bandwidth allocations.

2.1 Group Allocation Leveraging TCP

Our starting point is a classic result by Hahne [9].

Proposition 1 [9]: Flow control with large sliding window at sources plus fair queuing achieves max-min fair allocation.

Proposition 1 is applicable to every sliding window based flow control (hence TCP in particular). A corollary of Proposition 1 is that TCP congestion control together with fair queueing achieves max-min allocation.

In NetShare, we wish to allocate bandwidth in hierarchical max-min fashion first at the service level and only then at the TCP connection level. As shown in Figure 1, let's consider three services $A1$, $A2$, and $A3$ with weights $A1 : A2 : A3 = 4 : 1 : 1$ and suppose $A2$ has two TCP flows (from $H1$ and $H2$) while $A1$ and $A3$ have one TCP flow each. Fair queuing at TCP connection level does not achieve hierarchical max-min fair sharing: on link $C1$ to $E2$, the TCP connection from $A1$ is allocated $4/6$th of the bandwidth and thus gets only 6.6 Gbps instead of 8 Gbps.

However, if we do fair queuing at the *service* level, both connections belonging to service $A2$ are treated identically at core router $C1$ (i.e., mapped to the same queue). Assuming the fair mechanism gives both the TCP connections from $H1$ and $H2$ equal bandwidth, both limit themselves to 1 Gbps, which then allows TCP at $H3$ to grow to 9 Gbps. Thus we state the following proposition.

Proposition 2: Window flow control plus fair queuing at the service level achieves hierarchical max-min allocation.

The argument sketch is as follows. We adopt the standard water-filling algorithm [3] with some modifications to accommodate the hierarchical allocation. It starts by finding the *weighted bottleneck*. NetShare will emulate this by DRR [16] at the bottleneck link to give each application its weighted share. Next, we assume that the TCP flows of each application share the bottleneck link equally. While this is not strictly true if the TCP flows have very different RTTs, we assume this is true in data centers. Hence, each of these TCP flows cannot increase any further. Just as in the standard water-filling algorithm, we remove these TCP flows and their bandwidths, and recurse to find the new bottlenecks.

Concretely, for every switch and outbound link, we configure a separate fair queuing queue for each service class with the respective weight. In our hardware testbed, we used DRR [16] to configure fair queuing and ToS bits to distinguish services. Note that this is not the same as reservation. If a service is inactive or is routed on a different path it will not consume bandwidth on this link.

2.2 Stochastic NetShare

Since the switches may support limited DRR queues, NetShare scales to a large number of application classes by *stochastic weighted max-min fair sharing*, which is a generalization of McKenney's Stochastic Fair Queuing [12]. Concretely, applications are randomly hashed to specific DRR queues at each switch port. Each DRR group is assigned weight equal to the sum of weights of the individual applications that are hashed on to the DRR queue. The DRR grouping of applications in each switch can be different and can also be different at each switch port. Also, the grouping of applications onto DRR queues is changed periodically to avoid intermittent hashing imbalance. Note that most current switches do not support direct hashing to queues based on packet headers, but instead only allow mapping from header fields to queues via ACLs. We simulate hashing by having a central allocator provide labels to services; these labels can be randomized and changed periodically.

Since queue sharing allows a small weight service to steal more bandwidth, we propose a mixture of random and weight-based allocation as follows. First, we group services based on weight classes (say all services of weight 1, all of weight 2, all of weight 4, etc.). Then we map each weight class into a set of queues and randomly assign services to a specific queue within each set. In practice, given the small number of existing queues, we suggest grouping large weight services and low weight services into two classes, and randomly assigning within each set of queues. Clearly, this introduces errors due to weight aggregation and these errors can cascade. Nevertheless, it provides a solution to the difficult problem of combining scalability together with tunability.

2.3 Rate Throttling for UDP

In this section, we augment the TCP-based group allocation in Section 2.1 to handle aggressive UDP flows and misbehaving applications. Each host is instrumented with a *rate throttling shim layer* just below UDP. As illustrated in Figure 2, suppose $H1$ sends traffic at 10 Gbps to another host $H4$. The shim layer at $H4$ measures received traffic of 1 Gbps from $H1$. This is sent back to the corresponding rate throttling layer at $H1$ which rate-limits the traffic at close to 1 Gbps. Furthermore, to allow legitimate rate growth (e.g., $H1$ could grow to 2 Gbps if $H2$ disappears), we set the throttled rate to somewhat more than the measured rate to allow ramp-up.

The throttling mechanism is described in Algorithm 1. The receiver measures received throughput in some period T (e.g., 50 msec in our experiments) and sends a control (e.g., ICMP) message to the sender with the current measured rate C every T msec. The sender then executes Algorithm 1 to set the throttled rate R. The

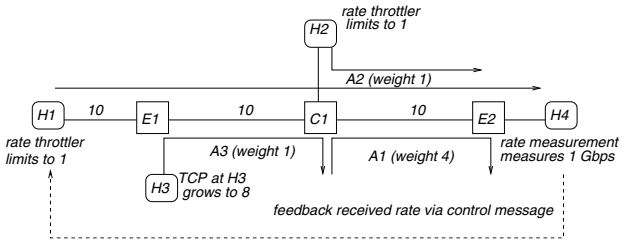


Figure 2: Simple fair queuing at switches at the *service* level together with rate measurement and rate throttling implements hierarchical max-min fair even with UDP.

Algorithm 1 Compute NetShare Rates at Rate Throttler

```
Performance tuning knobs:
  d: threshold of rate difference
  r_I: factor for increasing flows
  r_D: factor for decreasing flows
  r_O: factor for overshooting flows
Measurement parameters:
  L: last measured rate
  C: current measured rate at receiver
  f: flag indicating that the flow increased on last iteration.
  R: current rate limit
if (|(L − C)/L| ≥ d) then {is rate change substantial?}
  if C − L > 0 then {increasing, allow overshoot by r_I}
    R ← C · (1 + r_I)
    f ← true
  end if
  if C − L < 0 then {decreasing, allow overshoot by r_D}
    R ← C · (1 + r_D)
  end if
else
  if f = true then {limit overshoot by r_O}
    R ← C · (1 + r_O)
    f ← false
  end if
end if
if R < Th then {do not lower below threshold}
  R ← Th
end if
L ← C
```

values r_I and r_D are performance tuning knobs, indicating how much network bandwidth can be wasted. If r_I or r_D are large, then any newly freed up bandwidth can be acquired fast by the UDP traffic class. Note that very large difference between r_I and r_D would also cause instability and would make it harder for the rate to stabilize. In our experiments, we achieved good performance with $r_I = 20\%$ and $r_D = 10\%$. The flag f detects if the sender did increase in the last iteration so that the rate is set to r_O (typically $r_O < r_I$, say $r_O = 10\%$) higher than the measured rate C. This limits the amount of overshoot committed previously by the higher r_I. Also, we do not let the rate to fall below a threshold to avoid long ramp-up of small flows.

2.4 Centralized Bandwidth Allocator

In this section, we describe a centralized bandwidth allocator to allow advanced bandwidth allocation policies beyond max-min fairness. For instance, a more general policy can allow some connections between important servers to be allocated higher bandwidth. Another example is a policy on reallocation of excess bandwidth [6]. There are four steps.

1. Rate Measurement: The rate of each flow (TCP or UDP) for each service is measured at either the switches (using ACLs) or at the hosts (using a shim layer) in intervals of T seconds and used to predict a demand for the next interval.

2. Rate Reporting: The predicted rates are sent to a centralized bandwidth allocator (implemented on a server in the network) that is also supplied with the service weights and the topology via routing updates.

3. Centralized Calculation: The centralized allocator calculates rates for each flow and each service and sends back rate updates to the switches or hosts.

4. Rate Enforcement: Token bucket rate-limiters are used at the hosts or ingress switch ports to limit the rates to the calculated rates. Note that we rate-limit at the service level by the rate computation in Step 3. As in rate-throttling, each flow (especially TCP flows) must be allocated say 10% higher than its optimal centralized allocation to allow it to grow.

We have designed and implemented such a centralized allocator. The predictor in Step 1 is a standard least squares predictor using the last five measurements of traffic demand. The algorithm in Step 3 is a variant of the standard water-filling algorithm [3] which starts by finding the weighted bottleneck. We implemented the centralized algorithm on several large simulated 2-tier data center topologies. To approximate the solution to a large number of flows, we aggregated flows of the same weight and edge-to-edge path during the computation. On a simulated topology with 16 cores and 128 edge switches, the algorithm took less than 100 msec on a standard Intel Core2Duo 3GHz desktop.

Since the centralized bandwidth allocator includes a feedback loop, we also developed a formal feedback control framework for a rigorous proof of stability and convergence [6].

2.5 Discussion

NetShare can be integrated to a centralized management framework (such as Openflow / SDN controller) to update switch configurations (e.g., assigning weights during VM allocation and migration). Furthermore, the switch configuration process can be amortized into the launch and migration of VMs. Newly launched services are detected when VMs are allocated to them. The VM admission controller would also update the network controller, which then reconfigures switch weights accordingly. This does not limit NetShare scalability since the time to configure some switches in the network (in parallel) is less than the time to allocate and bring up a new VM. To identify an application or service class, we can use a combination of Type-of-Service (ToS) bits, IP options, etc., which are tagged on outgoing packets by the hypervisor and ACL rules in the network.

Modern switches typically support rate-limiting at finer granularity than port rate-limiting (QoS class, VLAN, ACL matching etc). Otherwise, rate-limiting can be offloaded to the VMM layer on the host and hence scales well to a large number of flows. Note that the number services hosted by servers locating at an edge switch can be much less than the total number of services in the whole data center. Therefore, with Stochastic NetShare we expect to scale to enterprise-level data centers with tens to hundreds of services. For a much larger scale with thousands of services, an approach such as Approximate Fair Dropping (AFD) [13] can reduce the approximation error of stochastic queue sharing in Stochastic NetShare. Future routers are expected to be equipped with a few thousand AFD queues for 16 DRR queues [13]. Note that AFD scales better because it uses a counter for each class as opposed to a queue.

Table 1 shows the tradeoffs between the three NetShare algorithms: group allocation, rate throttling, and centralized allocation. The group allocation mechanism has high responsiveness but relies

Table 1: Comparison of different NetShare mechanisms

	Deployment	Responsiveness	Generality
Group Allocation	Configuration at routers	$<$ 1 msec	Only TCP flows Only Hierarchical max-min
Rate Throttling	Configuration at routers Added endnode *or* router software	10-50 msec	Only Hierarchical max-min
Centralized Allocation	Centralized allocation software Added router software	10 - 100 msec	More general allocation policies

on TCP, and so it can be augmented with the UDP rate throttling to handle non-TCP traffic. The centralized allocator avoids the need for the UDP rate throttling in general, at the cost of higher complexity and lower responsiveness due to the control loop. Note that increasing generality must be paid for by smaller responsiveness and more software deployment.

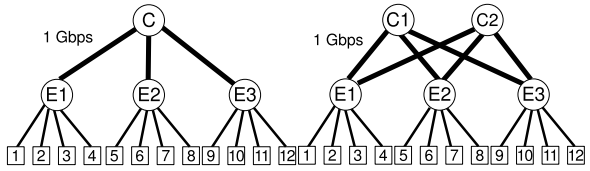
3. AUTOMATIC WEIGHT ASSIGNMENT

We propose a technique to perform service isolation by assigning weights to different services at each port in the network automatically. This helps avoid the need for network administrators to manually determine what weights different services must use. In an enterprise or cloud data center, when resources are provisioned for an application or customer, the customer usually requests some number of servers or VMs (instances) each with some number of CPUs, RAM and disk. Besides this, each instance must also be allocated some units of network bandwidth. For example, if each server has a 10Gbps NIC, we could place up to 10 VMs on the server with each allocated 1Gbps of bandwidth.

We leverage two fundamental ideas. First, we use per switch-port weights i.e., weights per application can vary from switch to switch, and even from one switch port to another. Second, we assign weights based on VM placement. We compute both the *downstream* and *upstream* sums of the bandwidths assigned to all VMs allocated to application A with respect to switch port P. Then the weight assigned to A at P is the smaller of the two.

As an example, suppose there is an accounting application with 2 servers connected to an edge switch and each server has 4 instances of an accounting application. The uplink of the edge switch is connected to a core switch and from there to other servers with 8 instances of the accounting application. Assume each VM instance is allocated 1 Gbps. Then we set the accounting application's weight at each of the 2 downlink ports of the edge switch to be 4 (smaller of 4 and 12), while we set its weight at the uplink port to be 8 (smaller of 8 and 8). Note that taking the minimum makes sense because even if there are 8 VMs upstream that can transmit at 8 Gbps, there are only 4 VMs downstream that can receive only at an aggregate capacity of 4 Gbps.

We make the following assumptions. First, VM bandwidths at the servers are enforced using mechanisms like Linux HTB qdisc. Second, we have knowledge of the complete topology and placement of each VM instance. Third, in a multirooted tree network, forwarding is based on ECMP. We assume that each egress port in a switch either forwards traffic upwards from a server towards the core layer (up-facing ports) and the rest of the fabric or forwards traffic down towards a server (down-facing ports). This is a technique that simplifies the routing while also avoiding routing loops. Finally, in this setup, each egress port on the switch has a definite role in terms of which server's traffic flows through it. For example in a two level multirooted tree network, a down facing port on a core switch can forward traffic to servers in a particular edge switch from all other edge switches while an up-facing port on an edge switch can forward traffic from the servers on that edge switch



(a) Single path (1 core) (b) Multipath (2 cores)

Figure 3: Testbed topologies

to all servers in other edge switches. Servers to which the particular port forwards traffic are called downstream servers and the servers from which this traffic could be coming are called upstream servers of that port.

While we have used global weights for the bulk of this paper, we note that extending the definitions to per-port weights is straightforward. For example, the standard water-filling algorithm [3] can be modified to use the weight of each application/service at the current bottleneck link as opposed to a global weight.

4. EVALUATION

We implemented NetShare on a small scale data center testbed consisting of a 24-port Fulcrum Monaco 10GigE switch [1] – a commercial switch with an extensive programming API for customizations. Out of the 24 switch ports, 12 were directly connected to the servers. Each server had two quad core Intel Xeon E5520 2.26GHz processors, 24 GB of RAM, and 16 local hard disks with 8 TB of total capacity. The remaining 12 ports were used to setup loopbacks (through a Glimmerglass optical MEMS switch) for partitioning the original 24-port physical switch into several virtual switches using VLANs and creating multi-switch data center topologies. Figure 3 shows our topologies. Multipathing was based on Equal-Cost Multipath (ECMP).

We evaluate the performance by investigating application completion times as the overall performance metric for the applications. Since the network is not the only factor, we also plot bandwidth utilization to demonstrate the ground truth. In this section, we show several key results to demonstrate the effectiveness of NetShare in sharing real data center applications, providing both bandwidth isolation and statistical multiplexing. Comprehensive experimental results are presented in [6].

4.1 Multipath Experiments

In this section, we demonstrate NetShare effectiveness by showing that it truly divides the bisection bandwidth (both core links) on demand with the topology in Figure 3b. Note that the core switches were the bottlenecks with an oversubscription factor of 2:1 for traffic between different edge switches. We used two Hadoop Sort applications $A1$ with 96 maps and 96 reducers, and $A2$ with 96 maps and 48 reducers.

Concretely, we first generated 96GB of data for each instance using the Hadoop RandomWriter application (8 maps per slave $\times$ 12 slaves). We subsequently ran two Hadoop Sort jobs in the two Hadoop instances $A1$ and $A2$. $A1$ used a total of 96 maps (8 per

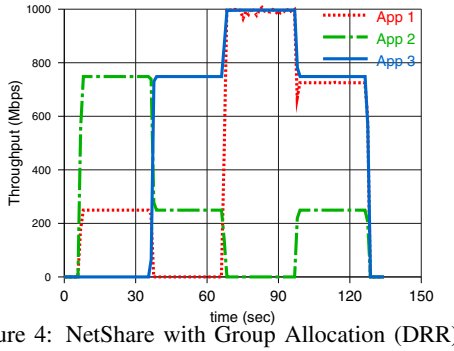


Figure 4: NetShare with Group Allocation (DRR) + Rate Throttling

Time(s)	$A1$	$A2$	$A3$	Bottlenecks
5-35	✓	✓	X	$E1, C$
35-65	X	✓	✓	$C, E3$
65-95	✓	X	✓	$E2, C$
95-125	✓	✓	✓	All of the above

Table 2: Traffic pattern that indicates times during which different flows were active.

slave) and 96 reducers (8 per slave) while $A2$ used 96 maps (8 per slave) and 48 reducers (4 per slave). The Hadoop Distributed File System (HDFS) was configured with a default replication factor of 3 and the HDFS block size was set to 256 MB.

First we ran the sort jobs without NetShare in the network. In this case, $A1$ used twice the bandwidth (summed over all core links, the "bisection bandwidth") when compared to $A2$ because it opens up nearly twice the connections. Next, we set up NetShare by configuring DRR with equal weights for the 2 applications. Note that the bandwidths on the various core links are not shared as uniformly because of hashing effects and because the sort job does not saturate all links consistently.

Using NetShare, $A1$ completed sorting in 1633s while $A2$ completed sorting the data in 1810s. As a comparison, we also ran $A1$ and $A2$ in the same fashion, but using the single core topology shown in Figure 3a. In this case, we found that $A1$ and $A2$ finished sorting in 3070s and 3212s respectively. After factoring out the 500s for the map phase (that is unaffected by the extra bandwidth), the bisection bandwidth appears to be nearly equally shared between the two "services" and both were sped up by nearly a factor of 2 in the multipath topology. Some difference is not surprising because $A1$ has more connections, and thus its use of ECMP load balancing is likely more effective than $A2$.

4.2 How Effective is Rate Throttling?

We deployed three applications in the testbed in Figure 3a: $A1$ generated a TCP flow from host $H1$ to host H5; $A2$ generated a UDP flow from host $H2$ to host $H9$; and $A3$ generated a UDP flow from host $H6$ to host $H10$. The weights of the applications $A1$, $A2$, $A3$ were set to 1:3:9 respectively.

Table 2 shows the traffic pattern. During the time 5-35s, $A3$ was inactive and thus the TCP flow $A1$ (weight 1) contended with the UDP flow $A2$ (weight 3) for the core link $E1, C$. From time 35-65s, the two UDP applications $A2$ and $A3$ (with weights 3 and 9) contended for the core link $C, E3$. From time 65-95s, the TCP application $A1$ contended with the high weight UDP application but only on the link from edge router $E2$ to core router C. Thus the UDP application could only interfere with TCP *acknowledgements* for $A1$ destined to Host $H1$.

We evaluate the following scenarios.

1. Group Allocation and Rate Throttling: We show that if we just set static weights locally, then with UDP, bandwidth allocation can be suboptimal. In Figure 4, each application received its weighted share of the network resources. For instance, during the period 5-35s, $A2$ got 750 Mbps and $A1$ got 250 Mbps as they were sharing the bottleneck $E1, C$ in the ratio 3:1 of their weights. However, from t=95-125s $A1$'s TCP flow got close to 725Mbps, which exceeded the share allocated by its application weight, but since $A2$'s UDP flow had a downstream bottleneck on the link $C, E3$ only 250 Mbps of the UDP flow was "useful" (that is the throughput of the UDP flow that actually reached the receiver $H9$). So in this case, $A2$ was rate-limited at the ingress switch to 275 Mbps (250 * 1.1) which results in $A1$ getting close to 725 Mbps. Without rate throttling $A1$ would have sent at much higher rates and got dropped at C.

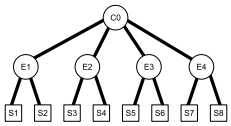


Figure 8: Topologies for Stochastic DRR experiments

2. No NetShare: We demonstrate that without a bandwidth isolation mechanism, a bandwidth-aggressive application can acquire much more than fair share. In Figure 5, when $A1$ and $A2$ were both active from t=5-35s, $A1$'s TCP flow was overwhelmed by $A2$'s UDP flow and received zero throughput. Note that from t=65-95s, $A1$'s throughput did not reach 1 Gbps although its path from $H1$ to $H5$ was not affected by $A3$'s UDP flow. However, the ACKs from $H5$ to $H1$ shared a link with $A3$'s UDP flow; some of the ACKs were dropped, this resulted in $A1$'s throughput dropping to sometimes as low as 750 Mbps.

3. Group Allocation Only: Figure 6 shows the impact of omitting Rate Throttling. In the period t=95-125s, $A2$ and $A3$ shared the bandwidth of their shared bottleneck link in the ratio of their application weights (3:9). Thus $A2$ only received 250Mbps. Unfortunately, $A1$ also received only 250Mbps because $A2$ continued to send greedily at 750Mbps on the $E1, C$ link of which 500Mbps got dropped at C.

4. Rate Throttling Only: In Figure 7, the behavior was similar to Case 1 from t=5-95s. However from t=95-125s, $A1$ only achieved 450-500Mbps. This is because $A2$ was rate-limited at $E1$ to a little over 500Mbps, so $A1$ was able to use the remaining bandwidth on the $E1, C$ link. Thus rate throttling and fair queuing are orthogonal and complementary mechanisms.

4.3 Scalability of Stochastic NetShare

A concern with Group Allocation is that it requires a number of queues equal to the number of applications. To scale beyond 16 queues commonly available today and the 1000's available shortly with AFD-based routers [13], we proposed Stochastic NetShare in Section 2.2. Due to restrictions on the physical queues in the switch, we had a simulation setup as in Figure 8. The topology had one core switch $C0$, four edge switches $E1$ to $E4$, and eight servers $S1$ to $S8$ (two servers per edge switch). Note that all links had equal capacity with an oversubscription factor of 2 at the core. We had 32 applications and one instance of each application on each server creating an all-to-all traffic pattern. One application was "bad", i.e. with low priority weight and competing aggressively for bandwidth by opening ten times the number of connections. The link capacity was $B = 100$ Mbps. We evaluated the scalability of Stochastic DRR by varying the number of DRR queues per switch port $Q = 4, 8, 16$.

Table 3 shows the application bandwidth at one typical server. All DRR queues were assigned the same weights independent of the number of applications being hashed into them. Due to the

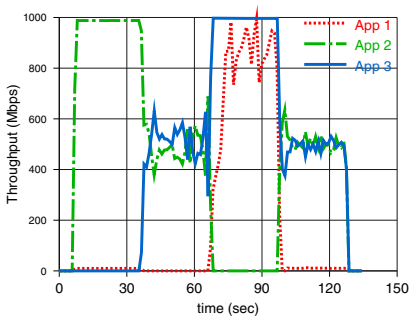


Figure 5: No NetShare mechanisms

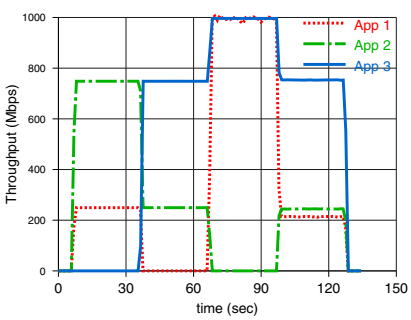


Figure 6: NetShare with Group Allocation Alone

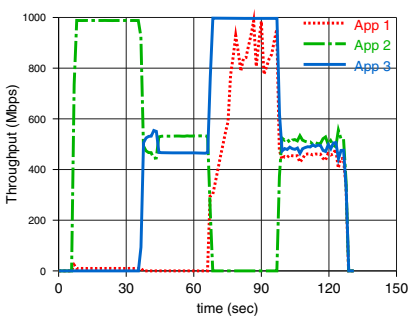


Figure 7: NetShare with Rate Throttling Alone

		T=5	T=10	T=20
Q=4	Bad app	(13.6, 3.3)	(14.3, 2.6)	(12.4, 2.8)
$\bar{B} = 2.3$	Good app	(2.2, 1.5)	(2.2, 1.3)	(2.4, 1.4)
Q=8	Bad app	(8.9, 2.1)	(8.9, 1.9)	(9.0, 2.0)
$\bar{B} = 2.7$	Good app	(2.8, 2.2)	(2.3, 1.7)	(2.5, 1.8)
Q=16	Bad app	(6.7, 1.5)	(6.6, 1.6)	(6.7, 1.7)
$\bar{B} = 2.9$	Good app	(2.8, 1.9)	(3.1, 2.2)	(3.3, 2.5)

Table 3: Scalability of Stochastic DRR: application bandwidth at one typical server in (mean, stddev) over time. All queues had equal weights. $\bar{B} = \frac{B}{N} \cdot \frac{Q-1}{Q}$ is the expected bandwidth per application. Ideal bandwidth is $\frac{B}{N} = 3.1$ Mbps. T is rehashing period (in seconds).

periodic rehashing process of Stochastic NetShare, the application rates oscillated, so our evaluation relied on mean and variance of the rate measurements. As shown in Table 3, the isolation performance of NetShare degraded gracefully with decreasing number of queues. Also, the impact of the bad application declined with additional queues in the system. The mean was close to our prediction (the ideal bandwidth is B/N which was around 3.1 Mbps, together with a degradation of $\frac{Q-1}{Q}$, where Q is the number of queues).

Note that periodic rehashing of applications onto DRR queues reduced variance. Clearly the rehashing period T should neither be too small (for good stability and minimizing out-of-order packets) nor too large (for good bias correction).

5. RELATED WORK

The need for QoS in data centers has become apparent in several recent papers [11]. Seawall [15] performs isolation by enforcing VM-to-VM rates for VMs belonging to one application/customer in the hypervisor using congestion feedback. SecondNet [5] proposes a heuristic to map *virtual data center* specifications into the physical data center infrastructure with constraints on resource demands. SecondNet uses reservations and hence is complementary to NetShare. Flowvisor [14] virtualizes a testbed network to allow multiple experiments to run concurrently but does so using suboptimal hop-by-hop allocation. The HP QoS Framework [10] allows network QoS to be implemented centrally but is only a framework that can, in fact, be used to implement NetShare. Oktopus [2] discusses a VM placement policy based on the network requirements for each customer. Bandwidth is reserved for each customer's VMs and the fair share for each flow is computed by a centralized controller for that customer. Multiplexing across customers requires coordination among controllers of different customers or a single central controller similar to NetShare.

6. CONCLUSIONS

NetShare allows managers to use weights to tune the relative bandwidth allocation for different services, providing isolation and statistical multiplexing without changing routers. Managers can use NetShare with Virtual Disks and Virtual Machines to create Virtual Data Centers. While NetShare is based on a simple packaging of existing ideas (max-min fair share, stochastic fair queuing, UDP rate throttling), no such mechanism is used today.

Group allocation works well with only configuration changes at routers; it can be extended to scale to more applications than the number of DRR queues available today either using AFD [13] or stochastic methods. Rate throttling handles UDP applications and may be simpler than deploying TCP-friendly UDP by modifying applications. Finally, centralized allocation can implement arbitrary bandwidth allocation policies, and can provide stability. We suggest a simple automatic weight assignment algorithm based on finding the number of VMs upstream and downstream from a port.

7. REFERENCES

[1] Fulcrum Monaco http://www.fulcrummicro.com/.
[2] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. Towards predictable data center networks. In *Proc. SIGCOMM'11*.
[3] D. Bertsekas and R. Gallager. *Data Networks*. P. H., 1992.
[4] A. Greenberg et al. VL2: A Scalable and Flexible Data Center Network. In *SIGCOMM'09*.
[5] C. Guo et al. SecondNet: A Data Center Network Virtualization Architecture with Bandwidth Guarantees. In *Proc. ACM CoNEXT '10*.
[6] Lam et al. NetShare and Stochastic NetShare: Predictable Bandwidth Allocation for Data Centers. In *UCSD Tech Report 2011*. `http://cse.ucsd.edu/users/vtlam/netshare-TR-2011.pdf`.
[7] S. Floyd et. al. Link-sharing and resource management models for packet networks. *IEEE/ACM Trans. Netw.'95*.
[8] Ajay Gulati, Arif Merchant, and Peter Varman. mClock: Handling Throughput Variability for Hypervisor IO Scheduling. In *Proc. OSDI '10*.
[9] Ellen L. Hahne. Round-robin scheduling for max-min fairness in data networks. *IEEE J. Comms*, 1991.
[10] W. Kim and et al. Automated and scalable qos control for network convergence. In *USENIX INM/WREN*, 2010.
[11] T. Lam, S. Radhakrishnan, A. Vahdat, and G. Varghese. NetShare: Virtualizing Data Center Networks across Services. In *UCSD Tech Report 05/2010*. `http://csetechrep.ucsd.edu/Dienst/UI/2.0/Describe/ncstrl.ucsd_cse/CS2010-0957`.
[12] P. McKenney. Stochastic fairness queueing. In *Internetworking,1991*.
[13] Rong Pan, Balaji Prabhakar, Flavio Bonomi, and Bob Olsen. Approximate Fair Bandwidth Allocation: A Method for Simple and Flexible Traffic Management. In *46th Allerton Conf., 2008*.
[14] Rob Sherwood, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martin Casado, Nick McKeown, and Guru Parulkar. Can the Production Network Be the Testbed? In *Proc. OSDI'10*.
[15] Alan Shieh, Srikanth Kandula, Albert Greenberg, and Changhoon Kim. Seawall: Performance Isolation in Cloud Datacenter Networks. In *Proc. HotCloud'10*.
[16] M. Shreedhar and G. Varghese. Efficient fair queueing using deficit round robin. In *In the Proceedings of SIGCOMM'95*.