NetShare: Virtualizing Data Center Networks across Services

Terry Lam, Sivasankar Radhakrishnan, Amin Vahdat, George Varghese

UCSD

Abstract

Data centers lower costs by sharing the physical infrastructure among multiple services. However, the data center network should also ideally provide bandwidth guarantees to each service in a tunable manner while maintaining high utilization. We describe *NetShare*, a new statistical multiplexing mechanism for Data Center networks that does this without requiring changes to existing routers. NetShare allows the bisection bandwidth of the network to be allocated across services based on simple weights specified by a manager. Bandwidth unused by a service is shared proportionately by other services. More precisely, NetShare provides weighted hierarchical max-min fair sharing, a generalization of hierarchical fair queuing of individual *links*. We present three mechanisms to implement NetShare including one that leverages TCP flows and requires no changes to routers or servers. We show experiments using multiple Hadoop instances and a network of Fulcrum switches and show that the instances can interfere without NetShare and yet complete faster with NetShare when compared to the alternative of static reservation.

1 Introduction

Cloud services are hosted by data centers that can hold thousands of servers connected by a network of switches that concurrently supports a large number of distinct services (e.g., search, video, email, analytics, and utility computing). The services are implemented on a shared data center because the cost of the physical equipment is large (more than 100 million dollars a year for large data centers [16]) and because statistical multiplexing using Virtual Machines (VMs) has been effective. However, the economics also requires two other characteristics of the data center *network*, both of which are only imperfectly provided today. First, to be profitable, the networks must have high *utilization*. At the same time, many services have stringent performance SLAs that must be met to keep customers satisfied: thus the network should also ideally provide *bandwidth guarantees* to each service. Finally, any new mechanism should not require hardware changes to existing switches as service providers are unlikely to retrofit their networks in the short term.

Our paper describes a new statistical multiplexing mechanism for data center networks called *NetShare* that provides both bandwidth guarantees and high utilization and can be implemented without any changes to existing switches. NetShare provides a precise specification of bandwidth guarantees using *hierarchical weighted max-min fair sharing* in which the bisection bandwidth of the network is i) first allocated to services according to simple weights and ii) the bandwidth of each service is then allocated equally among its TCP connections. Hierarchical max-min fair sharing generalizes the well known notion of hierarchical fair sharing of *links* [13] to *networks*.

We present three simple mechanisms to implement NetShare. The first mechanism we call *group allocation* relies on TCP and fair queuing but requires no software or hardware changes to switches or endnodes. It responds to changes in bandwidth in a few round trip delays. Unfortunately, it is not optimal when some services use UDP. Our second mechanism called *rate throttling* can augment link allocation to provide strong guarantees even when some services use UDP. Finally, analogous to the way Ethane [7] and RCP [6] provide centralized route computation, our third mechanism called *centralized allocation* requires a new software fabric manager but can provide more general bandwidth allocations.

NetShare can be viewed as a way to virtualize (i.e., statistically multiplex) a data center network among multiple services or slices. Given that CPUs and disks have been virtualized, it seems imperative to find some accompanying notion of network virtualization so that managers can create virtual data centers with performance guarantees over a shared physical data center. While one can argue whether NetShare's definition of virtualization is the correct one, some such proposal appears to be necessary and NetShare is perhaps the simplest starting point. The contributions of this paper are as follows and are presented in roughly the same order:

- A precise specification of what it means to share data

center bandwidth across services using hierarchical weighted max-min fair sharing (Section 2).

- Three mechanisms to implement NetShare (Section 3) with different tradeoffs (Table 1).
- Experiments (Section 5) using real implementations (Fulcrum switches and multiple Hadoop applications) and ns-2 simulations that show the benefits and scalability of NetShare.

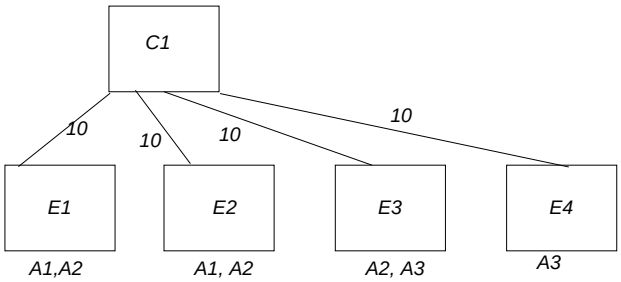


Figure 1: Example of a data center network shared between three services A1, A2, and A3.

2 NetShare Specification

NetShare is inspired by fair queuing of a single link [10] or proportionate sharing of a CPU. However, sharing a network is more complicated because multiple resources (links in a path) must be shared between slices. The generalization of fair sharing to multiple resources is called Pareto Optimality (in economics) or max-min fair sharing (in networks).

While max-min fair sharing at the TCP level is an old goal, we argue this is the wrong model. Instead, a data center manager wishes to share the data center between services/application classes/corporate groups and not at the individual connection level. A large corporation may wish to split bandwidth between a parallel CAD application, SAP, and Microsoft Exchange. max-min fair sharing at the TCP level is the wrong model for two reasons. First, services that open up multiple connections get an unfair share of bandwidth. Second, the manager cannot allocate more bandwidth to certain services based on their importance. Google may wish to allocate 80% of its bandwidth to Search if Search produces 80% of revenue.

Thus we are led to consider a new model: *weighted hierarchical max-min fair sharing*. First, the manager specifies a small number of services with weights assigned to each class that indicate their relative importance. Next, there is some mechanism that allocates network bandwidth in weighted max-min fair fashion among these services. The bandwidth assigned to each service is then recursively divided (again in max-min fair fashion) among the individual flows for that service. Note that the max-min fair definition specifies that bandwidth unused by a service is divided among other active services in proportion to their weights.

Example: Figure 1 shows a simple Data Center topology consisting of four edge switches $E1$, $E2$, $E3$, and $E4$ and 1 core switch $C1$. Each edge switch is connected with a 10 Gbps link to the core. Assume that there are three services $A1$, $A2$, and $A3$. Further, assume $A1$'s traffic needs to be sent from switch $E1$ to $E2$. Service $A2$ needs to send traffic from $E1$ to $E2$, and from $E1$ to $E3$. Service $A3$ needs to send traffic from $E3$ to $E4$. Service $A1$ is most important with a weight of 4, while $A2$, and $A3$ have a weight of 1 respectively.

The link from $E1$ to $C1$ is shared by applications $A1$ and $A2$. Assuming weights of 4:1, $A1$ should be assigned 8 Gbps while $A2$ should be assigned 2 Gbps. However, $A2$ has traffic from $E1$ to $E2$ and from $E1$ to $E3$. Assuming equal sharing of each edge-to-edge traffic flow *within* a given service, service $A2$ is allocated a bandwidth of 1 Gbps for traffic from $E1$ to $E2$, and 1 Gbps for traffic from $E1$ to $E3$. Thus, we can view NetShare as a generalization of hierarchical fair queuing on a link by link level to a network level.

But this allows service $A3$ to be assigned 9 Gbps for its traffic from switch $E4$ to $E3$ even though it has only the same weight as service $A2$ with which it shares a link. This happens because $A2$ is bottlenecked because of another link (the link from $E1$ to $C1$). This kind of calculation where a bottleneck limits the bandwidth of a flow, which then affects the bandwidth available to another flow, and so on iteratively, is formalized in the so-called Weighted max-min fair share calculation.

Now assume that service $A1$ reduces its bandwidth need to 6 Gbps. After some amount of time (measured by the responsiveness of the algorithm) NetShare can allocate 2 Gbps to $A2$'s traffic on the link from $C1$ to $E3$. This in turn reduces $A3$'s share to 8 Gbps. We can formalize this allocation as follows.

Definitions: A feasible bandwidth allocation of a set of flows is *max-min fair* if and only if a rate increase of one flow must come at the cost of a rate decrease of another flow with a smaller rate. A feasible bandwidth allocation of a set of flows is *weighted max-min fair* if and only if a weighted rate increase of one flow must be at the cost of a weighted rate decrease of another flow with a smaller rate.

A feasible bandwidth allocation to a set of applications is *hierarchical max-min fair* if and only if a weighted rate increase of a flow within one application must be at the cost of

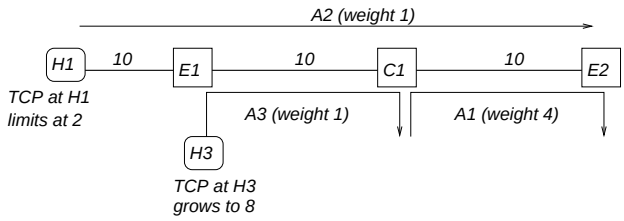


Figure 2: Simple fair queuing at switches together with TCP implements max-min fair sharing of TCP flows [17].

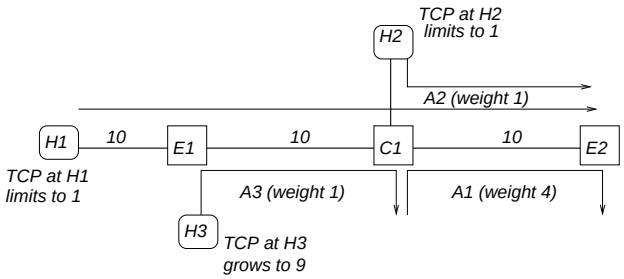


Figure 3: Simple fair queuing at switches at the *service* level together with TCP achieves *hierarchical* max-min fair sharing of services.

a weighted rate decrease of some other flow either (i) within the same application with a smaller flow rate or (ii) within some other application with a smaller weighted application bisection bandwidth.

3 NetShare Algorithms

In this section, we describe how NetShare can be implemented. Section 3.1 describes *group allocation* which relies on TCP. Section 3.2 describes *rate throttling* for UDP hosts. Finally, Section 3.3 describes a centralized bandwidth allocator that can implement more general allocation policies.

3.1 Group Allocation Leveraging TCP

Our starting point is a classic result by Hahne [17] which is paraphrased as follows in [27].

Proposition 1: [17]: A large sliding window at sources plus fair queuing achieves max-min allocation.

The intuition is illustrated in Figure 2. Assume 3 competing TCP flows: a first from service $A1$ that traverses bottlenecked link from $C1$ to $E2$; a second from service $A2$ starts at host $H1$ and goes from $E1$ to $C1$ and from $C1$ to $E2$; finally, a third flow from service $E3$ that traverses the link from $E1$ to $C1$. Assume that $A1$'s flow is configured to have a fair queuing weight of 4 at core switch $C1$ while other flows are assigned weight 1.

Thus fair queuing at $C1$ will assign $1/5$-th of the bandwidth of the $C1, E2$ link to the $A2$ flow because the $A1$ flow has 4 times the weight. In a few round trip delays, TCP at $H1$ will adjust its rate to 2 Gbps. But this allows TCP at H3 to grow to 8 Gbps because only 2 Gbps is used on the link from $E1$ to $C1$. Hahne's result formalizes this intuition but has a number of caveats. For example, the proof [17] applies to only some arrival distributions such as Bernoulli arrivals.

However, in NetShare we wish to allocate in hierarchical max-min fashion first at the service level and only then at the TCP connection level. So consider Figure 3 which adds one more host $H2$ that also belongs to service $A2$ with weight 1 and shares the link from $C1$ to $E2$ with $A2$'s other TCP flow from $H1$ and $A1$'s flow. Fair queuing at the *TCP connection level does not* achieve hierarchical max-min fair sharing. 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, then 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 conjecture the following proposition:

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

We have no formal proof of this result which we leave as an open problem. It has the same caveats as the Hahne result [17] but has some further wrinkles. For example, it assumes the individual TCP flows share the service bandwidth equally. However, it is well known that TCP flows with shorter round trip delays can get larger shares. Fortunately, in a data center network such cases should be rare as most connections are likely to use a similar number of hops. A second subtlety is the use of ECMP path splitting which is common in data centers. The Hahne result assumes a single path. Max-min allocation with path splitting in general topologies requires complex optimization algorithms [22, 9].

Despite the lack of formal proof, we have found in our experiments with real switches and ns-2 experiments on data center topologies that Proposition 2 holds even with multipath topologies. Proposition 2 suggests an extremely simple

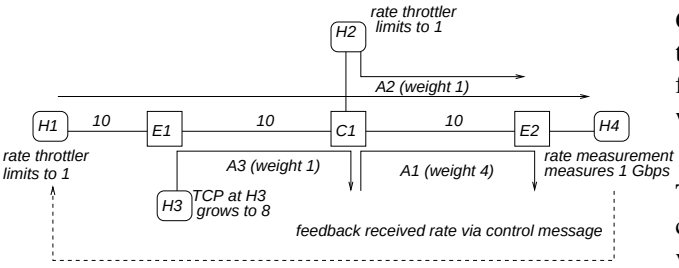


Figure 4: Simple fair queuing at switches at the *service* level together with rate measurement and rate throttling implements hierarchical max-min fair even with UDP.

mechanism that requires no software or hardware changes to endnodes or switches.

Group Allocation Mechanism: For every switch and every outbound link configure separate fair queuing queues for each service/application class with weights specified by the manager.

For example, in Fulcrum switches [1] we have used DRR [29] to configure fair queuing and TOS bits to distinguish services. The queue weights are then set (on all outbound links) to the NetShare weights specified by the manager. 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.

3.2 Rate Throttling for UDP

Group allocation relies on TCP senders. However, many important applications including Tibco's multicast protocols, Veritas, and Oracle [4] do not use TCP. Buggy or erroneous settings can cause such traffic to flood the network. Thus, we seek a a simple mechanism that can preserve the goals of NetShare and yet work with UDP sources.

First, what goes wrong in Figure 3 if $H1$ and $H2$ use UDP? In that case, $H1$ could continue to send at 10 Gbps on the link to $E1$ and the fair queuing mechanism at $E1$ will assign it 5 Gbps on the link to $C1$ (dropping the remaining traffic) providing only 5 Gbps to $A3$'s traffic from $H3$. This is unfortunate because the fair queuing mechanism at $C1$ will only allocate 1 Gbps to the traffic from $H1$. Thus if $H1$ sends at 10 Gbps, 5 Gbps of traffic is dropped at $E1$ and 4 Gbps is dropped at $C1$. There is no congestion collapse but the allocation is far from optimal.

We propose a simple idea to effectively replace TCP. Assume that each host has *rate throttling shim layer* just below UDP. For example, in Figure 3 suppose that $H1$ sends at 10 Gbps to some other host $H4$ as shown in Figure 4. The rate throttling layer at $H4$ measures a received traffic of 1 Gbps from $H1$. This is sent back to the corresponding layer at $H1$ which rate limits the traffic to close to 1 Gbps.

Unfortunately, $H1$ cannot rate limit exactly to 1 Gbps. This is because if say the flow from $H2$ disappears, $H1$ could grow to 2 Gbps. But the rate limit at $H1$ will prevent $H1$ from ever finding that it can grow. Thus we need to set the throttled rate to somewhat more (say $x\%$) than the measured rate to allow ramp up. Higher values of x allow faster ramp-up but increase the amount by which a flow can overshoot its allocation. We chose a value of $x = 20\%$ as a compromise.

The throttling code we use is slightly more complicated and is described in Algorithm 1. Note that Rate Throttling requires weighted fair queuing at each router as well. We assume the receiver measures received throughput in some period T (we use 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.

First, the code adds some hysteresis to prevent changes when the difference between the last measured rate (stored in L) and the current rate is too small (less than 5%). Next, if the current measured rate is greater than the last measured rate, the new sending rate is set to 20% higher than the measured rate C. If the current measured rate is smaller than the last measured rate, the new sending rate is set to 10% higher than the measured rate C.

There are two final subtleties. First, even if the difference is too small, if the sender had increased on the last iteration (this is kept track of by flag f), the sending rate is set to 10% higher than the measured rate C. This limits the final overshoot to 10%. For example, suppose the target max-min rate is 100 and the last measured rate is 94 and the current measured rate is 100. The sender goes 20% higher in the next iteration to roughly 114. However, if the next measured rate is also 100, the next iteration will set the sending rate to 110. Thus after a brief overshoot of at most 20% the final overshoot will be 10%. Finally, we do not let the rate to fall below a threshold, because if a flow's rates becomes too small it will take too long to ramp up.

Implementation Choices: Rate throttling can be implemented entirely in the hosts through a kernel patch. This can gradually be deployed at all data center hosts; in the interim if fair queuing is used, minimum bandwidths will be provided though statistical multiplexing may be reduced. The other choice is to perform the rate throttling in the network using say OpenFlow [24] switches and a platform like FlowVisor [26]. Egress switches perform rate measurement and the ingress switches do rate limiting. While we could

Algorithm 1 Compute NetShare Rates at Rate Throttler

```
Let L denote last measured rate
Let C denote current measured rate at receiver
Let f denote a flag indicating that the flow increased on
last iteration.
Let R denote current rate limit
if (|(L−C)/L| ≥ 0.05) then {is rate change substantial}
  if C − L > 0 then {increasing, increase by 20%}
    R ← C ∗ 1.2
    f ← true
  end if
  if C − L < 0 then {decreasing, increase by 10%}
    R ← C ∗ 1.1
  end if
else
  if f = true then {limit overshoot}
    R ← C ∗ 1.1
    f ← false
  end if
end if
if R < T then {do not lower below threshold}
  R ← T
end if
L ← C
```

use existing TCP-friendly UDP congestion control protocols like DCCP [23] at hosts, the advantage of rate throttling is that it is simple to implement even at switches. Note that most TCP-friendly congestion protocols measure drops and use the TCP equation [25] unlike our simple scheme.

3.3 Centralized Bandwidth Allocator

While NetShare based on fair queuing is efficient, it can only calculate a hierarchical max-min allocation. A more general policy would allow some connections between important servers to be allocated higher bandwidth. As a more complex example of a useful allocation policy, consider the single link example shown in Figure 5 with 3 services $A1, A2, A3$ sharing a congested link from an edge to a core switch.

If the link bandwidth is 10 Gbps and each service has the same weight, then each service is guaranteed $1/3$ of the bandwidth i.e., 3.33 Gbps. However, if $A3$ idles, then $A1$ and $A2$ get 5 Gbps each. Suppose, however, we wish to limit the "excess" bandwidth that $A2$ gets. Then we can define a second set of weights for sharing the excess bandwidth that becomes available when some services are idle. For example, if we define the excess bandwidth weight of $A1$ to be twice as much as that of $A2$, then if $A3$ idles, 3.33 Gbs of bandwidth becomes available. This excess is now allocated

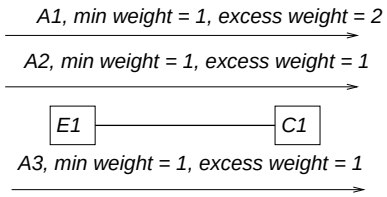


Figure 5: Allocation mechanisms that divide excess bandwidth using different weights.

among $A1$ and $A2$ in the ration 2:1, so that $A1$ gets 2.22 Gbps of the excess and $A2$ gets 1.1 Gbps. Thus, in sum $A1$ gets 5.55 Gbps and $A2$ gets 4.4 Gbps. By setting the the excess weight to zero we can prevent a service from getting any excess bandwidth.

However, such an allocation is impossible using fair queuing at switches. Instead, inspired by centralized routing schemes like RCP [6] or [7] we propose the use of a centralized bandwidth allocator based on four simple mechanisms.

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 PC 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. For example, it is easy to implement the standard water filling algorithm ([5]) to calculate max-min allocations. It is also easy to generalize to minimum and excess weights.

4. Rate Enforcement: Token bucket rate limiters are used at the hosts or ingress switch ports to limit the rates to the calculated rates. 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 5 measurements of offered traffic. The algorithm in Step 3 is a variant of the standard water-filling algorithm [5] which starts by finding the *weighted bottleneck*. This is the link with the smallest value of $B_j/\Sigma_s w_s$, where B_j is the bandwidth of link j and we take the sum of the weights of all services s that traverse link j.

For example, in Figure 3, the initial bottleneck is the link

from $C1$ to $E2$ which has the value $10/5 = 2$. The algorithm allocates 8 to service $A1$ and 2 to service $A2$. Next, the algorithm allocates equal shares to $H1$ and $H2$ flows within service $A2$ to give them 1 Gbit each. This second step (connection allocation) differs from the standard water-filling algorithm but is needed for hierarchical max-min allocation. These values are then subtracted off each link that these flows use and the algorithm recurses to examine the link from $E1$ to $C1$. This allows the flow from $H3$ to be allocated 9 Gbps.

The generalizations we proposed can easily be added to the main iterative step without increasing complexity. For example, a connection can be allocated more shares at each step. Further, we can allocate the excess bandwidth on the current bottleneck link using two weights at each step.

We used some algorithmic techniques to reduce complexity to $O(FPS)$ where F is the number of flows, P is the average path length of all flows, and S is the number of services. First, we maintained two auxiliary data structures: one that maps from links to flows that traverse the link, and one that maps from flow to the set of links used by the flow. Second, the bottleneck link is determined by a bucket-sorting heap which removes the logarithmic term. In a real data center, the diameter is typically small (4 to 6) and we assume the number of key services is small < 20). Thus the resulting complexity is linear in the number of flows.

We implemented the centralized algorithm on several large simulated 2-tier data center topologies. On a simulated topology with 16 cores, 128 edge switches and 128 million flows, the algorithm took less than 100 msec a standard Intel Core2Duo 3GHz desktop. Smaller and more common topologies took less than 10 msec. This is important because this is the only extra term that affects responsiveness to bandwidth changes compared to rate throttling. It also suggests that the measurement in Step 1 should not be done over intervals finer than 10's of msec.

Table 1 shows the tradeoffs between the three Netshare algorithms: group allocation, rate throttling, and centralized allocation. Note that increasing generality must be paid for by smaller responsiveness and more software deployment. None of the algorithms require hardware changes to switches.

4 Implementation

In this section, we show the effectiveness of NetShare in sharing real data center applications, providing both bandwidth isolation and statistical multiplexing. Data centers (e.g., Google, Walmart data centers) host multiple concurrent applications (e.g., MapReduce instances, data mining

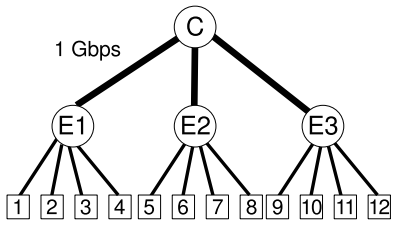


Figure 6: A topology with one core and three pods

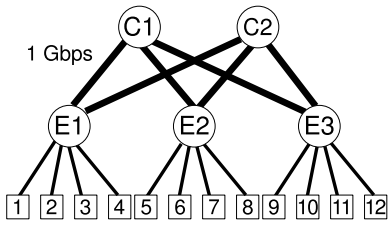


Figure 7: A topology with two cores

instances). For simplicity, we model each application as a MapReduce instance using Hadoop[2].

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 advanced customization. Twelve switch ports are directly connected to servers. Each server has 2 quad core Intel Xeon E5520 2.26GHz processors, 24 GB of RAM, and 8 TB of local disk storage. The remaining twelve ports are all connected to a Glimmerglass optical MEMS switch which is used like a patch panel to setup loopbacks between these twelve ports on the Fulcrum switch. This gives us the flexibility to partition the 24 port physical switch into virtual switches using VLANs and create interesting multi-switch data center topologies through the loopbacks.

In particular, we configured the simple data center topology shown in Figure 6 with one core switch and three edge switches. Figure 7 shows another data center topology that we used with two core switches and three edge switches. Here, we configured multipathing on the edge switches to utilize both core switches through Equal Cost Multipath (ECMP). All ports in the topology were configured to run at 1Gbps to stress the network. End to end RTT between any two nodes is less than 100us. Note that the physical links are short optical fiber cables, so link latency is mainly due to processing overhead at the switches and end hosts. Also, in our topologies, the term pod corresponds to an edge switch and there are four servers connected to each pod.

We implemented Group Allocation by configuring Deficit

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

Round Robin (DRR) at a service level. DRR already existed on the switch but we had to implement UDP rate throttling in the switches; we did not modify servers. To classify application traffic, we marked the application ID in the Type of Service (ToS) field in the packet header.

5 Evaluation

We describe experiments using a single path topology in Section 5.1 and using a multipath topology in Section 5.2. We describe experiments to evaluate the effectiveness of rate throttling in Section 5.3 the scalability of NetShare in Section 5.4. All experiments were conducted on the testbed except the scalability experiments which used ns-2.

5.1 Single Path Experiments

We evaluated the performance of 2 Hadoop applications $A1$ and $A2$ (that model two concurrent Map-Reduce "services") , in a topology with a single core switch as shown in Figure 6. HDFS was configured with a replication factor of 6 to stress the system by transferring more data through the network for replication; this only affects the phase where the sorted results are written back to HDFS. We also set the HDFS block size to 256MB. One of the servers was configured as a master while all the servers were configured as slaves. The two instances were configured to use 8 disks each (4 for HDFS and 4 for the task tracker) on each server.

We first generated 96GB of data for each instance using the Hadoop RandomWriter application (8 maps per slave $\times$ 12 slaves). We subsequently ran 2 Hadoop Sort jobs in the 2 Hadoop instances $A1$ and $A2$. $A1$ used a total of 96 maps (8 per slave) and 96 reducers (8 per slave) while $A2$ used 96 maps (8 per slave) and 48 reducers (4 per slave).

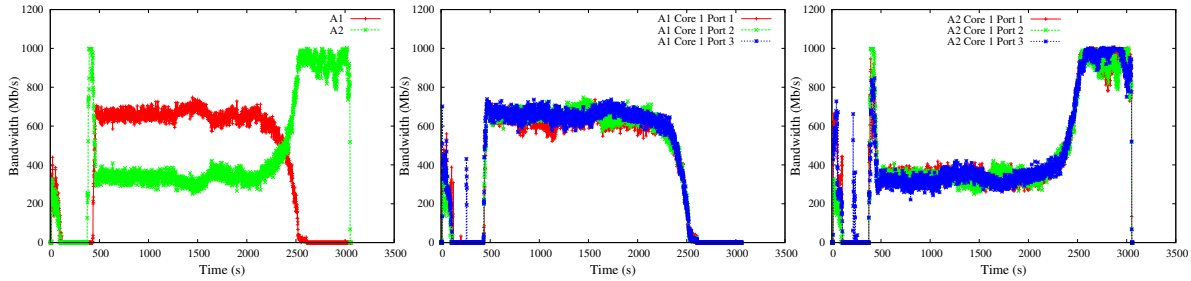
In the map phase, there is minimal network utilization and the jobs are mainly CPU/disk bottlenecked. During the reduce phase, there is considerable data shuffling and the network is highly utilized. Note that $A2$ uses twice as many reducers and so generates twice as many TCP connections to other slaves as $A1$. When we ran the experiment without configuring any fair queuing (no NetShare), we observed that $A1$ used nearly twice as much bandwidth as $A2$ corresponding to the ratio of connections that it opens.

Figure 8a shows the bandwidth observed at a representative port on the core switch for the 2 applications without fair queuing (no NetShare). It shows the ratio in which the bandwidth is shared on the core switch link. Figure 8b shows the bandwidth used by $A1$ on all the 3 core switch ports. Figure 8c shows the bandwidth used by application $A2$ on all the 3 core switch ports. Once again, $A2$ uses twice the bandwidth on all core links because it has twice the connections.

$A1$ completed sorting in 2586s. $A2$ completed sorting the data in 3028s. The difference in completion times is not twice because for the first 500s both are in the map phase and do not need much network bandwidth; for the next $\approx$ 2000s, $A2$ gets twice the bandwidth and $A1$ runs two times slower. Finally, from 2000s onwards, $A1$ runs 3 times faster because it has the core links to itself. Thus the second half of its sort shuffle takes $2000/3 \approx 660$. Thus the total sorting time for $A2$ should be around $500 + 2000 + 660 = 3160$. Note that if $A2$ was running continuously (as can happen in a production network), the completion time will go up to 4500 which is nearly twice as slow.

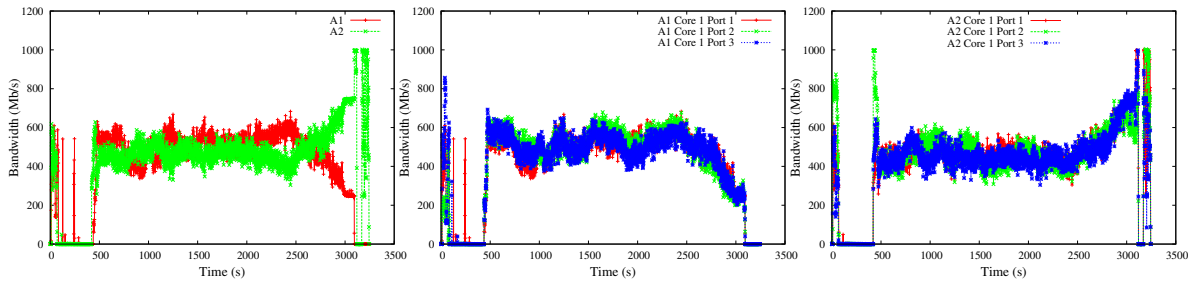
When DRR was configured with equal weights for the 2 Hadoop instances (NetShare Group Allocation), we observed that $A1$ and $A2$ were able to utilize almost equal bandwidth on the core switch links although $A1$ had more connections open.

Figure 9a shows the bandwidth observed at a representative port on the core switch for the 2 applications with NetShare using equal weights for both applications. It shows that the bandwidth is shared by the 2 applications almost equally on the core switch link. Figure 9b shows the bandwidth used by application A1 on all the 3 core switch ports. Figure 9c shows the bandwidth used by application $A2$ on all the 3 core switch ports. The bandwidth is shared almost equally by the 2 applications on all the core switch ports.



(a) Representative Link BW **(b) BW for App 1 on all core links** **(c) BW for $A2$ on all core links**

Figure 8: Single Path Experiment without NetShare



(a) Representative Link BW **(b) App 1 BW on all core links** **(c) App 2 BW on all core links**

Figure 9: Single Path Experiment with NetShare Group Allocation (via DRR)

In this case, $A1$ completed sorting the data in 3070s. $A2$ completed sorting the data in 3212s. The smaller difference in completion times arises because NetShare shares bandwidth equally despite the larger number of connections opened by $A1$. Notice that the maximum completion time for both instances is almost equal with or without NetShare. This is because the total amount of work that is done (sorting 96 x 2 GB) of data is the same in both experiments.

5.2 Multipath Experiments

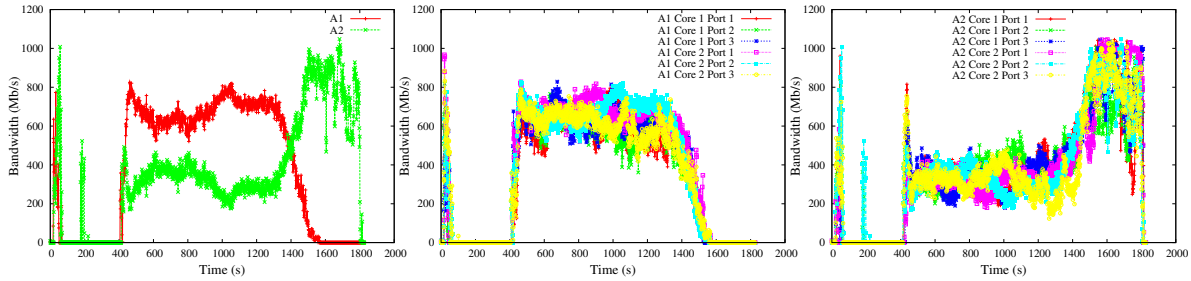
To examine how NetShare performs with multipathing as is common in data centers, we deployed the same Hadoop experiment described earlier in the 1 core switch topology but this time using the 2 core switch topology shown in Figure 7. Each edge/pod switch performs ECMP (Equal Cost Multipath) to hash flows onto the two paths for interpod flows. Again the core switches were the bottlenecks with an oversubscription factor of 2:1 for interpod traffic. We again setup Hadoop Sort applications $A1$ with 96 maps and 96 reducers, and $A2$ with 96 maps and 48 reducers.

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. Figure 10a shows the bandwidth observed at a representative port on one of the core switches for the 2 applications without NetShare. Figure 10b shows the bandwidth used by application A1 on all the 6 core switch ports. Figure 10c shows the bandwidth used by application $A2$ on all 6 core switch ports.

In the multipath case, $A1$ completed sorting the data in 1558s. In the single path case, $A1$ completed sorting in 2586s. Since 500s is devoted to the map phase, the actual sort phase has been sped up by almost a factor of 2, as expected with twice the bisection bandwidth.

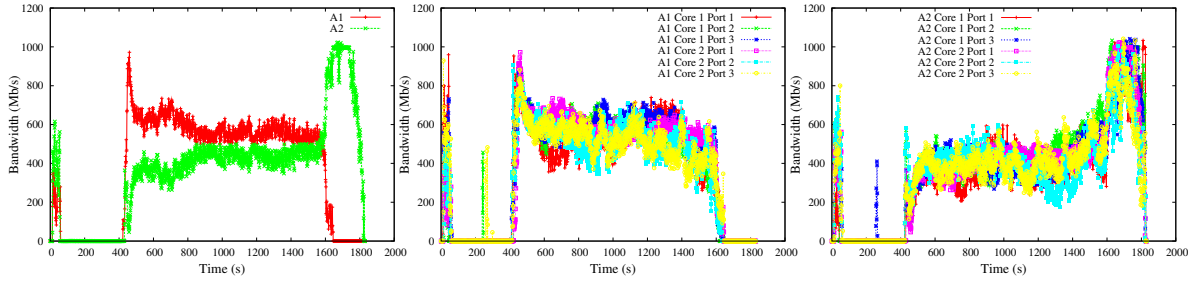
Next, we set up NetShare by configuring DRR with equal weights for the 2 applications. Figures 11a, b, and c show the bandwidth distribution on a representative core link, the bandwidth distribution on all 6 core ports for $A1$, and the bandwidth distribution on all 6 core ports for $A2$ respectively. Note that the bandwidths on the various core links are not shared as uniformly because of hashing effects and because the sort does not saturate all links consistently.

However, $A1$ completed sorting in 1633s while $A2$ completed sorting the data in 1810s compared to sorting in 3070s and 3212s with a single core. 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". Some difference is not



(a) Representative Link BW **(b) BW for $A2$ on all core links** **(c) BW for $A2$ on all core links**

Figure 10: Multipath Experiment without NetShare



(a) Representative link BW **(b) BW for $A1$ on all core links** **(c) BW for $A2$ on all core links**

Figure 11: Multipath Experiments with NetShare Group Allocation via DRR

surprising because $A1$ and $A2$ have different access patterns and thus do not always contend for the core links at the same time.

5.3 How Effective is Rate Throttling?

In order to evaluate the performance of rate throttling, we used the same testbed as described earlier in the 1 core switch configuration (Figure 6). We used 3 applications for this experiment. Application $A1$ generates a TCP flow from host $H1$ to host H5, Application $A2$ generates a UDP flow from host $H2$ to host $H9$. Application $A3$ generates a UDP flow from host $H3$ to host $H10$. For the DRR and rate throttling experiments, the weights of the applications A1, $A2$, A3 were set to 1:3:9 respectively.

The traffic pattern we generated as an input to the experiment is shown in Table 2. Note that during time 5-35s, $A3$ is inactive and thus the TCP flow $A1$ (weight 1) contends with the UDP flow $A2$ (weight 2) for the core link $E1, C$. Next, from time 35-65, the two UDP applications $A2$ and $A3$ (with weights 3 and 9) contend for the core link $C, E3$. In the third phase (time 65-95) the TCP application $A1$ contends with the high weight UDP application but only on the link from edge router $E2$ to core router C. Thus the UDP application can only interfere with TCP *acknowledgements* for $A1$ destined to Host $H1$.

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 are active.

We performed experiments under different scenarios. In each case we measured the bandwidth at the receiver for the 3 applications:

- **Group Allocation and Rate Throttling:** First, we measured the bandwidth achieved by the 3 flows with Group Allocation (via DRR) and Rate Throttling (Figure 12). In this case, each application received its weighted share of the network resources as dictated by the application weights. For instance, during the period 5-35s, $A2$ gets 750 Mbps and $A1$ gets 250 Mbps as they are sharing the bottleneck $E1, C$ in the ratio 3:1 of their weights. However, from t=95-125s $A1$'s TCP flow gets close to 725Mbps, which exceeds the share allocated by its application weight, but since $A2$'s UDP flow has a downstream bottleneck on the link $C, E3$ only 250 Mbps of the UDP flow is "useful" (that is the through-

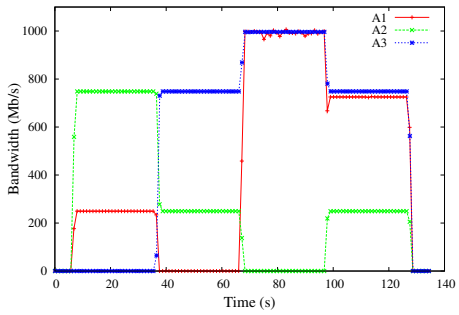


Figure 12: NetShare with Group Allocation (DRR) + Rate Throttling

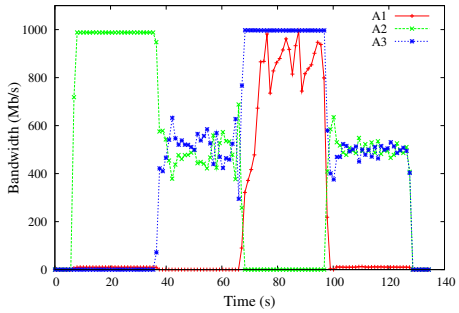


Figure 13: No NetShare Mechanisms

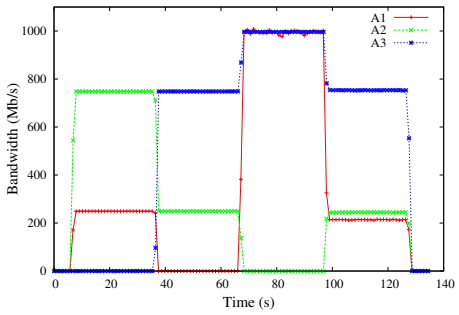


Figure 14: NetShare with Group Allocation Alone

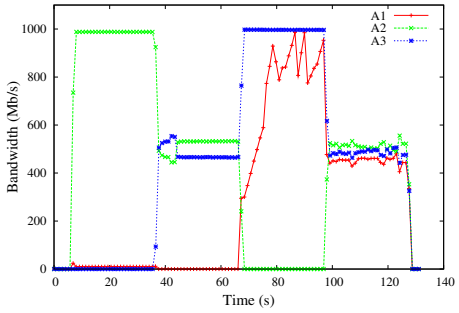


Figure 15: NetShare with Rate Throttling Alone

put of the UDP flow that actually reaches the receiver $H9$). So in this case, $A2$ gets rate limited at the ingress switch to 275 Mbps (250 * 1.1) which results in $A1$ getting close to 725 Mbps. Without rate throttling we will see that $A1$ will send at much higher rates and get dropped at C.

- **No NetShare:** Second, we measured the bandwidth achieved with no NetShare mechanisms (Figure 13). In this case, whenever $A1$ and $A2$ were both active in time 5-35s, $A1$'s TCP flow is overwhelmed by $A2$'s UDP flow and receives zero throughput. Note that from t=65-95s, $A1$'s throughput does not reach 1 Gbps although its path from $H1$ to $H5$ is not affected by $A3$'s UDP flow. However, the ACKs from $H5$ to $H1$ share a link with $A3$'s UDP flow; some of the ACKs get dropped, this results in $A1$'s throughput dropping to sometimes as low as 750 Mbps.

- **Group Allocation Only:** Third, we measured the bandwidth achieved with only Group Allocation (DRR configured with weights for $A1$, $A2$ and A3 as 1:3:9) as shown in Figure 14. The most interesting scenario occurs from t=95-125s. In that period, $A2$ and $A3$ share the bandwidth of their shared bottleneck link in the ratio of their application weights (3:9). Thus $A2$ only receives 250Mbps. Unfortunately, $A1$ also receives only 250Mbps. this is because $A2$ continues to send greedily at 750Mbps on the $E1, C$ link of which 500Mbps gets dropped at C. This motivates the need for rate throttling.

- **Rate Throttling Only:** Fourth, we measured the bandwidth achieved by the 3 flows with only rate limiting but no DRR configuration as shown in Figure 15. Here the behavior is similar to the case without any DRR (Case 1) from t=5-95s. However from t=95-125s, observe that $A1$ is able to achieve nearly 450-500Mbps. This is because $A2$ gets rate limited at $E1$ to a little over 500Mbps, so $A1$ is able to use the remaining bandwidth on the $E1, C$ link. Thus rate throttling and fair queuing are orthogonal mechanisms each of which independently adds value to NetShare.

5.4 Scalability of NetShare

In the previous sections, we performed experiments with NetShare implementation on real switches. Due to the resource constraints of our hardware testbed, we explore the

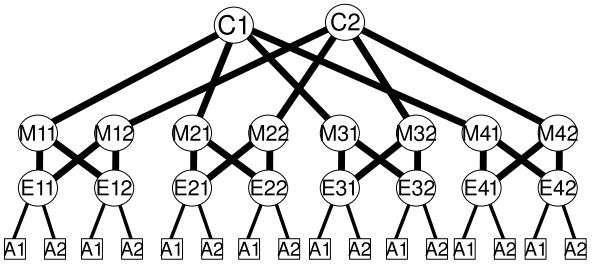


Figure 16: Three-tiered data center topology used for scalability experiments

scalability of NetShare with a larger data center topology by simulation using ns-2. Because even ns-2 took too long to run at 1 Gbps, we scaled down the speeds to 10 Mbps and used a moderate sized 3-tier topology as shown in Figure 16. We explored scalability by running NetShare with a varying number of services and TCP connections per service.

The topology used four pods and three layers of switches (core switches $C1$ and $C2$, aggregation switches $M11$ to $M42$, and edge switches $E11$ to $E42$) as shown in Figure16. All links between the switches are 10 Mbps. Each service has its own dedicated node at each edge switch that alternate. Figure 16 shows the pattern for two services: $A1$ and $A2$ are assigned to nodes alternately. If there are three services, the edge switches will each host 3 machines which are assigned to $A1$, $A2$ and $A3$ respectively.

The communication pattern for each service is all-pairs, i.e. each node opens connections to all other nodes in the same service. We also allow each pair of nodes to open up to C connections in parallel, where C is a parameter we vary to explore scalability. We explore the parameter space of the number of applications (N), application connections (C) and policies with and without NetShare as shown in Table 3.

We vary the parameters of the first service (A) and keep same configurations for the remaining $(N-1)$ services (B). W denotes the DRR weights for A and B respectively. Since B represents a set of services, we report only the maximum and minimum bandwidths for the services in set B.

We observe that NetShare with Group Allocation via DRR comes close to achieving the desired network sharing independent of the number of connections that each individual service makes. For example, as shown in Table 3, each of the four service with two connections between any pair of nodes get 36 Mbps. Without NetShare, one heavyweight service can acquire more bandwidth by increasing its connections per node pair (e.g. upto 84 Mbps with 8 connections and 106 Mbps with 16 connections). On the other hand, NetShare always prevents that service from getting more that 40 Mbps. Note that this is slightly above its fair share (36 Mbps) but independent of the connections made other services.

Table 3: Application bisection bandwidth under several traffic parameters and with and without NetShare (DRR only).

N	C	DRR?	W	Bandwidth (Mbps)		
				A	B (Max)	B (Min)
1	2	-	-	131.7		
1	8	-	-	141.7		
4	2, 2	Y	1, 1	35.6	35.1	37.0
4	2, 2	Y	1, 2	22.6	40.0	41.5
4	8, 2	Y	1, 1	40.2	34.0	36.0
4	8, 2	N	-	83.8	19.4	20.6
4	8, 2	Y	1, 2	24.7	38.5	41.9
4	16, 2	Y	1, 1	40.1	35.1	35.8
4	16, 2	N	-	105.5	12.2	14.0
4	16, 2	Y	1, 2	25.2	39.4	40.7
8	2, 2	Y	1, 1	18.3	17.7	19.0
8	2, 2	Y	1, 2	10.4	18.9	20.0
8	8, 2	Y	1, 1	20.4	17.1	18.5
8	8, 2	Y	1, 2	11.6	18.3	19.8
8	8, 2	N	-	53.1	11.6	13.9
8	16, 2	Y	1, 1	20.2	17.1	18.4
8	16, 2	Y	1, 2	11.7	19.0	20.2
8	16, 2	N	-	77.9	9.1	10.4

Furthermore, bisection bandwidths also reflect the NetShare administrative weights. For example, the bisection bandwidths for the four services, one of which having weight 1 and the remaining having weight 2, are 22 Mbps and 41 Mbps respectively. Also, the service with smaller weight cannot increase its share by increasing the number of connections between its nodes. Finally, we scale the experiments from 2 to 4 to 8 services and observe similar effects.

6 Discussion

We briefly discuss NetShare guarantees and potential generalizations.

NetShare and Bisection Bandwidth: The bisection bandwidth of a network is the bandwidth of the smallest cut that divides the network into two halves. A service i is deployed over some subset of machines S_i. We can definite a bisection bandwidth B_{s_i} of that subset. NetShare will allocate the bisection bandwidth of a service in proportion to

its NetShare weight. More precisely, service i is allocated $B_{S_i} \cdot (w_i/\Sigma_j w_j)$ regardless of the allocations of other services j. However, if we knew that some other service k was not sharing any links with service i, Service i could be assured a higher fraction of its bisection bandwidth (w_k can be removed from the denominator). This is akin to the hose model [12]. Note if a service is deployed initially in a rack with high bisection bandwidth and is then deployed on machines spread across racks, its B_{S_i} may decrease but its assured fraction stays the same.

NetShare and Failures: Group allocation and rate throttling are naturally fault tolerant. When a network failure occurs, after routing recalculates paths, the fair queuing changes at any affected links take place instantaneously. By contrast, reservation based schemes that reserve bandwidth on backup links would be unnecessarily wasteful in the normal case when failures do not occur. However, a centralized allocator would have to learn the new routes, possibly by listening to routing updates.

NetShare for ISPs: The economic model for ISPs is unclear because if a customer pays for a bandwidth slice, how should the customer be charged for bandwidth gleaned from unused slices? Second, if a customer slice was idle it will take NetShare some time to return the customer's bandwidth. Customers may respond by introducing artificial traffic. Such gaming could be ameliorated by a tit-for-tat scheme. Neither problem is an issue with enterprise networks. A simple economic model is to allocate NetShare Service weights according to revenue or to allocate costs to each service according to weights. Gaming should be unlikely with a central administration.

Generalizing NetShare: NetShare appears to generalize to fat tree topologies. It possibly also generalizes to more dynamic routing as in VL-2 [16]. VL2 provides isolation by randomization but cannot be used to tune service allocations via weights. One concern about deploying NetShare over VL-2 is the speed of routing changes in VL-2 compared to NetShare responsiveness.

7 Related Work

Flowvisor [26] virtualizes a testbed network to allow multiple experiments to run concurrently but does so using wasteful hop-by-hop allocation. The HP QoS Framework [21] allows network QoS to be implemented centrally and can be used to implement NetShare. Fair queuing [11] and Core-stateless fair queuing(CFSQ) [30] do not guarantee max-min allocation. citecorelite and [28] extend CFSQ to obtain max-min allocations but require header changes. DiffServ allows statistical multiplexing by marking traffic exceeding the allocated share and dropping marked packets if needed. For lack of space, we omit experiments that show that Diff Serv dropping reduces efficiency compared to NetShare.

Centralized algorithms for max-min allocation were described in [5] and approximations in [3]. Many papers (e.g., [22, 9]) show that max-min allocations in the presence of load balancing or multipath is, in general, NP-complete. However, the hardness result does not apply to regular data center topologies.

Duffield et al introduce a hose model [12] to specify aggregate demand but requires complex algorithms which decrease responsiveness. NetShare assumes that routing is fixed by a routing protocol such as OSPF. Thorup and Rexford show that there is considerable flexibility to change routes by changing OSPF weights [14] without pinning every route using MPLS. Traffic engineering such as OSPF-TE (RFC 3630) and TexCP [19] can be used to efficiently route a traffic matrix but does not allocate bandwidth across services. RFC 3630 (Traffic Engineering Extensions to OSPF) only supports static reservation.

Several congestion avoidance algorithms (e.g., [20, 18, 8, 15] converge to a max-min allocation via signaling from the network. By contrast, we generalize the technique of [17] which uses no signaling from the network in the case of TCP flows. Finally, NetShare differs from DRL which controls bandwidth *in and out* of the cloud as opposed to *within* the cloud.

8 Conclusions

The notion of a virtual data center requires *both* computing and bandwidth guarantees. NetShare allows managers to use weights to tune the relative bandwidth allocation of different services. Without NetShare, a service can be held hostage by other services that either open multiple connections or use non-compliant congestion control protocols. Our paper introduces three simple techniques for implementing the NetShare abstraction ranging from group allocation per link to centralized allocation that trade decreasing responsiveness for more general allocation policies.

References

[1] Fulcrum Monaco Product Brief. http://www.fulcrummicro.com/documents/products/Monaco_PB_1.2.pdf.

[2] Hadoop: http://hadoop.apache.org/. Technical report.

[3] Arash Asadpour and Amin Saberi. An approximation algorithm for max-min fair allocation of indivisible goods. In *STOC '07: Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*, pages 114–121, New York, NY, USA, 2007. ACM.

[4] D. Bergamasco and et al. Backward congestion notification (bcn): A tuto rial. Technical report.

[5] D. Bertsekas and R. Gallager. *Data Networks*. Prentice Hall, 1992.

[6] M. Caesar and et al. Design and implementation of a routing control platform. In *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, pages 15–28, 2005.

[7] Martin Casado and et al. Ethane: taking control of the enterprise. *SIGCOMM Comput. Commun. Rev.*, 37(4), 2007.

[8] Anna Charny, K. K. Ramakrishnan, and Anthony Lauck. Time scale analysis scalability issues for explicit rate allocation in atm networks. *IEEE/ACM Trans. Netw.*, 4(4):569–581, 1996.

[9] J. Chou and B. Lin. Optimal multi-path routing and bandwidth allocation under utility max-min fairness,. In *EEE International Workshop on Quality of Service (IWQoS)*, 2009.

[10] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queuing algorithm. In *Proc. SIGCOMM '89*, September 1989.

[11] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queuing algorithm. In *Proc. SIGCOMM '89*, September 1989.

[12] N. G. Duffield, Pawan Goyal, Albert Greenberg, Partho Mishra, K. K. Ramakrishnan, and Jacobus E. van der Merwe. Resource management with hoses: point-to-cloud services for virtual private networks. *IEEE/ACM Trans. Netw.*, 10(5):679–692, 2002.

[13] Sally Floyd and Van Jacobson. Link-sharing and resource management models for packet networks. *IEEE/ACM Transactions on Networking*, 3:365–386, 1995.

[14] Bernard Fortz, Jennifer Rexford, and Mikkel Thorup. Traffic engineering with traditional ip routing protocols. *IEEE Communications Magazine*, 40:118–124, 2002.

[15] Emily E. Graves, R. Srikant, and Don Towsley. Decentralized computation of weighted max-min fair bandwidth allocation in networks with multicast flows. *Lecture Notes in Computer Science*, 2170, 2001.

[16] A. Greenberg and et al. Vl2: A scalable and flexible data center network. In *Proceedings of ACM SIGCOMM*, 2009.

[17] Ellen L. Hahne. Round-robin scheduling for max-min fairness in data networks. *IEEE JOURNAL ON SELECTED AREAS IN COMMUNICATIONS*, 9, 1991.

[18] Raj Jain. A timeout-based congestion control scheme for window flow-controlled networks. *IEEE Journal on Selected Areas in Communications*, October 1986.

[19] Srikanth Kandula, Dina Katabi, Bruce Davie, and Anna Charny. Walking the tightrope: responsive yet stable traffic engineering. In *SIGCOMM '05: Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 253–264, New York, NY, USA, 2005. ACM.

[20] Dina Katabi, Mark Handley, Charlie Rohrs, and Mit lcs Icsi Tellabs. Internet congestion control for future high bandwidth-delay product environments. In *in Proceedings of ACM SIGCOMM*, 2002.

[21] W. Kim and et al. Automated and scalable qos control for network convergence. In *USENIX INM/WREN 2010*, 2010.

[22] Jon Kleinberg, Yuval Rabani, and Eva Tardos. Fairness in routing and load balancing. In *J. Comput. Syst. Sci*, pages 568–578, 1999.

[23] Eddie Kohler, Mark Handley, and Sally Floyd. Designing dccp: congestion control without reliability. *SIGCOMM Comput. Commun. Rev.*, 36(4):27–38, 2006.

[24] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, 2008.

[25] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling tcp throughpu: A simple model and its empirical validation. Technical report, Amherst, MA, USA, 1998.

[26] R. Sherwood and et al. Flowvisor: A network virtualization layer. Technical report.

[27] R. Sherwood and et al. Rate adaptation, congestion control and fairness: A tutorial. Technical report.

[28] Z. Shi. Token-based congestion control: Achieving fair resource allocations in p2p networks. In *Innovations in NGN: Future Network and Services, 2008. K-INGN 2008. First ITU-T Kaleidoscope Academic Conference*, 2008.

[29] M. Shreedhar and G. Varghese. Efficient fair queueing using deficit round robin. In *In the Proceedings of SIGCOMM'95*, pages 231–243, 1995.

[30] Ion Stoica, Scott Shenker, and Hui Zhang. Core-stateless fair queueing: a scalable architecture to approximate fair bandwidth allocations in high-speed networks. *IEEE/ACM Trans. Netw.*, 11(1):33–46, 2003.