NetShare: Virtualizing Bandwidth *within* the Cloud

Terry Lam and George Varghese

Computer Science and Engineering
University of California, San Diego, MS 0114
La Jolla, CA 92040

Tel: 858 822 1619
Fax: 858 534 7029
{varghese}@cs.ucsd.edu

February 4, 2009

Abstract

We present NetShare, a way of dividing a network into slices, such that that each network slice has the illusion of a network all to itself with guaranteed bandwidth. Bandwidth unused by a network slice can also be shared proportionately among active network slices. NetShare extends fair queuing of *links* to fair sharing of *networks*. NetShare's virtualization of bandwidth complements existing technologies for virtualizing computation (as in VMWare) and storage (as in Amazon's S3). Together they allow organizations to rent or internally maintain multiple "virtual clouds" on the same physical infrastructure. While some degree of bandwidth virtualization can be provided by mechanisms such as virtual circuits and RSVP, NetShare provides a higher-level abstraction that can provide both minimum guarantees *and* statistical multiplexing. NetShare can also handle adversarial sources and requires no state in the core. In NetShare, ingress routers measure their demand to egress routers, perform a hierarchical Max-Min fair share computation, and then drop packets at the ingress if a slice exceeds its share. We demonstrate that even on large networks like ATT the optimized calculation takes less than 100 msec, and that the mechanism can be deployed with only software changes to existing routers. We show application experiments in which NetShare is used to provide throughput guarantee to multiple Map-Reduce users who can otherwise interfere with each other.

1 Introduction

> *As companies virtualize physical servers and provision processor and memory resources to specific virtual machines (VMs), delivering I/O to these VMs presents the next set of challenges.* – Jerome Weindt, Virtualization Review, 2008 [2]

Network balance allocation has historically swung between the twin poles of *predictable* and *efficient* allocation. When the source demand is predictable as with voice traffic, predictable allocation as in simple TDM on T1 links works well. On the other hand, when the source is bursty traffic, networks resort to statistical multiplexing by allowing all the traffic into the network on demand. At times, this causes the equivalent of traffic deadlock in highways, and results in congestion collapse and other unpredictable behavior. However, the standard approach has been to overprovision the network and to assume that such cases are rare, which is ironic because the orginal goal was efficient allocation.

A more elusive goal, which we seek in this paper, is to combine efficiency with predictability. More precisely, we seek bandwidth guarantees between endpoints of a network together with potential increased bandwidth when some users do not use their shares. We believe a particularly interesting application for such technology is what we call *internal cloud computing*.

Amazon's EC2 [1] and Google's GoogleApps have stimulated excitement about what is called "cloud-computing". The idea is that users access computational services from a provider cloud (e.g., Amazon), avoiding the need to understand these services or deploy them locally. Users avoid capital expenditures for equipment and operational expenditures for maintenance, but instead pay for what they use. The cloud provider profits by economies of scale because aggregation allows resources to be utilized more fully. We

distinguish *commodity* cloud-computing where the users are individuals (for example, Joe accessing his Google Calendar) from *business* cloud computing where the users are companies (say Southwest running flight schedule computation on EC2).

Cloud computing is typically implemented on data centers consisting of servers running virtual machines interconnected by a network of routers and switches. On Amazon's EC2 web service, for instance, customers can rent an arbitrary number of Virtual Machines where they can load software of their choice. Similarly, Amazon's S3 allows customers to rent an arbitrary amount of disk storage. However, for business cloud computing that is computationally intensive and naturally parallel, the software running on each Virtual Machine must also communicate. Thus, there is a need to not just get computational and storage guarantees but also *bandwidth* guarantees. This can be used to create the abstraction of what some have called a virtual data center or a virtual cloud.

While cloud computing provides a trendy motivator for slicing network bandwidth, the problem arises in a bigger arena: enterprise data centers. Data centers run by traders and major enterprises such as Walmart will typically have multiple applications running on separate Virtual Machines. Many of these applications are highly parallel and need to communicate. Further, in a traditional data center, a server may use only 10 percent of the processor capacity.

By loading more applications using VMs, utilization may grow by a factor of 5 [2], potentially also increasing network I/O needs. In particular, there can be peak demands such as VM backups. If the VMs for one application are being backed up, these can cause other critical applications to reduce their effective bandwidths. We refer to this as *internal* cloud computing (customers are internal applications) versus *external* cloud computing as in Amazon's EC-2 (where the customers are external).

The need for bandwidth guarantees is accentuated by the desire to combine storage (e.g., FiberChannel), CPU (e.g., Infiniband), and LAN (e.g., Ethernet) networks into a single unified interconnect (e.g., FiberChannel over Ethernet). In such a setting it becomes even more critical to guarantee disk and CPU traffic, the real-time responses they need with minimal latency. Companies such as Fulcum and Woven have already announced technologies for virtualizing single switch fabrics. Our paper is about extending such virtualization to a *network* of switches.

Bandwidth slicing can also be applied to ISPs with say ATT creating network slices connecting customer locations for major clients such as FedEx. However, our proposal appears more useful for internal cloud computing because it is less likely that internal customers will "game" the system. Our ideas also apply to PlanetLab and the NSF GENI vision of a virtualized testbed. While PlanetLab provides computation slices and some ad hoc bandwidth guarantees, a future testbed could benefit from also providing bandwidth slices. Our algorithms can be added with small changes to GENI hardware proposals such as [21].

The rest of the paper is organized as follows. In Section 2 we describe the model for customers and providers. Section 3 describes the basic technique used internally to realize the model, while Section 4 provides a detailed specification of the NetShare algorithm. Section 6 describes experiments that demonstrate its viability. Section 7 describes related work and Section 8 describes our conclusions.

2 Model for Bandwidth Slicing

Abstractly, each customer specifies a set of endpoints and bandwidth requirements between each endpoint. This is similar to the pipe model for VPNs [**?**]. A customer slice consists of the set of bandwidth demands between each endpoint together with a weight. The weight will determine how any excess bandwidth will be distributed. We propose the following goals for a bandwidth slicing mechanism. Various subsets of these goals are shared by earlier solutions; however, the combination of these goals appears to be unique.

- *Simple Model:* The slicing model should be easy for customers and providers to reason about at a network level. The slicing mechanism should "compile" customer requirements to QoS settings on routers.
- *Bandwidth Firewalls:* A network slice must always be guaranteed the bandwidth specified between all endpoints, even if other customers are adversarial. For instance, some applications may use UDP, others may use multiple TCP connections, others may be buggy or being developed.
- *Statistical Multiplexing with Guarantees:* The mechanism should allow bandwidth unused by a slice to be shared by other slices in proportion to their weights (as in fair queuing). For internal cloud computing, it makes sense for bandwidth unused by one application to be shared by other applications. In an ISP, there is less of an incentive to provide a customer "more" than what has been paid for.
- *Responsiveness to Change:* The time $T_{recover}$ taken for the mechanism to respond to a change in slice bandwidth requirement should be as small as possible (minutes if not seconds).

- *Software Deployment:* The mechanism should be deployable with software changes to existing routers and switches, and with no IP header changes.
- *Miminal state in core switches:* As in core-stateless fair queuing, there should be minimal (or no) state required for the mechanism in core switches.

Our network slicing model is inspired by fair queuing of a single link where a single expensive link can be shared between two customers in proportion to their weights. Fair queuing offers a simple model that is easy to reason about, provides bandwidth firewalls, proportional statistical multiplexing, and fast response to change. However, we wish to extend this notion to networks. This is more complicated because network slices use paths that overlap arbitrarily.

At first glance, network slicing appears as old as the hills. Bandwidth reservation mechanisms have been available from the days of IBM's SNA and early virtual circuit networks such as TYMNET and TELENET. Modern incarnations of bandwidth reservation include ATM flow control for virtual circuits, VPN reservations via CSPF, MPLS and RSVP, and service classes for virtual lanes in Infiniband. However, to the best of our knowledge, none of these existing technologies provides bandwidth reservation together with statistical multiplexing and a core-stateless implementation. We discuss these approaches in Section 7.

In some sense, in the history of systems there has always been a tension between predictability (bandwidth firewalls) and efficiency (statistical multiplexing for bursty traffic), and efficiency has largely ruled the roost. Both predictability and efficiency are combined in solutions for *single* resources like processor sharing (CPU scheduling) or fair queuing (single link scheduling). However, in a network each customer required different subsets of resources (i.e., different sets of links corresponding to the paths needed).

The natural generalization of single link fair sharing to a network of multiple resources is a well known theoretical idea called Pareto Optimality (in economics) or Max-Min fair sharing (in networks). However, the two main contributions of this paper are: first, defining a simple model that leverages the Max-Min fair concept and is easy for users and providers; and second, making this notion work efficiently in a systems setting with minimal changes to existing IP routers. Further, there are aspects of our solution that leverage the special features of internal cloud computing and a data center setting where "gaming" the system is less likely than in an ISP, and where some applications can tolerate a small relaxation of the bandwidth firewall property in return for statistical multiplexing.

Note that this paper is about the ability of managers ability to control the allocation of bandwidth to slices and not about a way to "fairly" allocate bandwidth to slices, where there is some "moral" connotation to fairness. The Max-Min fair computation is simply a tool that drives our allocation mechanism, and tuning is possible by relative weights.

Note also that the primary goal must be balancing the needs of various slices; overall network throughput and efficiency is only a secondary goal. To see this, consider a network of two links $E1$ and $E2$ in series. Suppose a flow $F1$ (representing a pair of VMs) wish to communicate and must traverse both $E1$ and $E2$. Suppose another flow $F2$'s route only traverses $E1$ and a third flow's route only traverses $E2$. Then maximum network throughput is obtained by always scheduling flows $F2$ and $F3$ (which can operate concurrently) and never scheduling $F1$. But presumably flow $F1$ is essential to the business, and thus such scheduling is absurd. The relative importance of $F1$ can be quantified in our model by specifying the bandwidth of $F1$ appropriately. If $F1$ is indeed unimportant, it can be given a prescribed (i.e., reserved) bandwidth of 0 and can thus count only on getting unused bandwidth.

2.1 Usage Model for Internal Cloud Computing

We envisage the following use of network slicing for internal cloud computing. Consider the manager of a data center consisting of a network of switches and links. The manager associates slices with internal "customers" such as cost centers (e.g., Accounting, Research, Order Fulfillment) or applications. Each slice is associated with a set of VMs that reside on physical machines that connect to switches. Assume that the manager allocates bandwidths between VMs. The bandwidth demand between VMs is aggregated to find the bandwidth demand between machines which in turn is used to find the bandwidth demand between ingress-egress switches.

Based on the long term demand between switches, the manager periodically runs an algorithm such as OSPF-TE to set link weights to efficiently satisfy the worst-case traffic matrix (when all demands are operational). This can be used to control routes so that the interference between the network paths taken by slices can be controlled. The manager periodically specifies a traffic matrix of prescribed bandwidths for each slice based on each customer/application need. There are four models for the bandwidth usage of each slice.

- *Low latency, static allocation:* The bandwidth of that slice is reserved and will never be shared by others. However, the slice does not get access to any unused

bandwidth of other slices. This is useful for applications with predictable bandwidth (e.g., VoIP) that cannot afford the latency of waiting for T seconds to regain their share after an idle period.

- *Low latency, elastic:* The bandwidth of that slice is reserved and will never be shared by others. However, the slice can still use unused bandwidth. This is useful for applications whose performance can improve with more bandwidth but have tight latency demands (e.g., trading traffic). Such slices can steal from others but do not let their slice be stolen.
- *Latency tolerant, elastic:* Such slices may take up to $T_{recover}$ seconds to recover their bandwidth share after an idle period. This is useful for applications that can improve with more bandwidth but can afford some ramp-up latency (e.g., bulk transfer, data mining). The model can be extended to allow two specified bandwidths B and B_{min}, where the slice never falls below an allocation B_{min} at any time even if it sends no traffic. We ignore this possibility in what follows.
- *Latency tolerant, static:* Such slices may take up to $T_{recover}$ seconds to obtain their bandwidth share but do not share excess bandwidth.

These four models can easily be implemented by network slicing. If a slice is marked as requiring 0 ramp-up delay, its bandwidth measurement is always reported as its prescribed allocation. If a slice is marked as no sharing, the NetShare computation does not give the slice more than its prescribed share.

3 NetShare Overview

Our technical contributions are simple: 1) First, we do the Max-Min Fair Share allocation hierarchically, reducing an $O(F^2)$ algorithm to an order $O(P^2)$ algorithm. 2) We use a measurement algorithm to accurately predict future traffic from past traffic 3) We use a bandwidth stealing algorithm to ensure that network slices can ramp up usage quickly.

NetShare slices (i.e., virtualizes) network bandwidth in two steps. Step 1 ignores customer slices, and only considers the network as being shared between every ingress and egress router. Step 1 (Figure 1) reduces the network to a set of virtual links between every pair of ingress and egress points. Step 2 divides the bandwidth of each virtual link between customers slices. Steps 1 and 2 combine to slice network bandwidth between customers. Customers have a guaranteed minimum bandwidth (as if they made bandwidth reservations) as well as a potential upside from bandwidth unused by others (as in statistical multiplexing). Both steps are performed every T seconds, where T is the NetShare computation interval, say 1 second in a data center. We describe each step in detail.

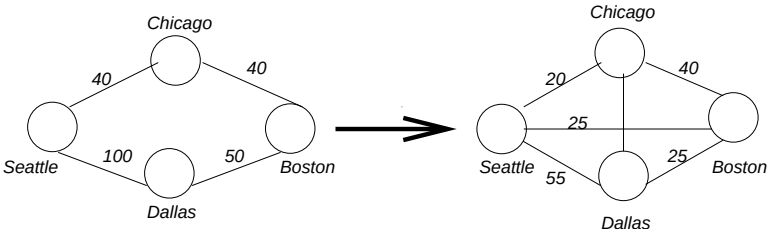


Figure 1: Step 1 of the NetShare algorithm: a network with ingress-egress routers and an arbitrary topology is reduced to a network with virtual links between every pair of inputs and outputs. Links are labelled with bandwidths in Mbps. Bandwidths of virtual links are calculated by a static allocation followed by a Max-Min fair share of the excess; these limits are enforced at the edges by token buckets.

3.1 Step 1, Creating Virtual Links between every Input and Output Port

In some sense, this first step is reminiscent of the older way of provisioning bandwidth between endpoints in say Frame Relay networks by reducing the network to a set of virtual links between every pair of endpoints. Unlike those networks, we will allow statistical multiplexing across these virtual "pipes".

In an ISP, input and egress routers could be POP routers. In a data center, these would be edge routers. We effectively divide the "bandwidth" of the network between every ingress-pair by first satisfying the prescribed bandwidth allocated to each customer network slice. After subtracting out this bandwidth, the remainder is allocated among ingress-pairs using a weighted Max-Min fair share allocation, where the weights are the ratios of the bandwidths prescribed for each ingress-egress pair. At the end of Step 1, each ingress-egress pair has an allocated bandwidth that is at least the bandwidth specified by the sum of the bandwidths allocated to every network slice between this ingress-egress pair.

In Figure 1, assume traffic from Chicago to Dallas is routed via Seattle, and traffic from Seattle to Boston is routed via Dallas. Assume for simplicity that all traffic demands and routes are symmetrical. Thus, the link between

Chicago and Seattle is shared between these two flows. Assume the network manager prescribes a bandwidth of 20 Mpbs for each of these flows. Similarly, the flow from Seattle to Boston shares a 50 Mbps link with Dallas to Boston, and so assume the manager prescribes 25 Mbps to each flow. This leaves 55 Mbps that can be allocated to Seattle to Dallas. While the manager could assign these bandwidths, they can be automatically calculated using a Max-Min fair allocation [5] algorithm with equal weights for every OD pair. With this allocation, traffic from Seattle to Boston is guaranteed 25 Mbps even if there is a Denial-of-Service attack from Dallas to Boston.

While we have used an ISP example, the ideas carry over to Data Center networks. In the example, ingress-egress flows share bottleneck bandwidths equally. However, we generalize to make distinctions. For example, if the Seattle to Boston flow has 4 times more customer traffic than from Dallas to Boston, then the Seattle to Boston flow will be allocated 40 Mbps, and the Dallas to Boston will be allocated 10 Mbps. This has to be done carefully to make sure the resulting traffic matrix is feasible (A simple tool can report infeasible allocations.) For example, the manager may have to change the allocation of Seattle to Dallas to 40 Mbbps.

Note that simple fair sharing on each link in isolation does not use bandwidth optimally. For examplle, in Figure 1 this would reduce the allocation of Seattle to Dallas to 33 Mbps, when both Chicago to Dallas and Seattle to Boston cannot use their (isolated) allocation of 33 Mbps. Note that while TCP flows will lower their demand to the bottleneck, we wish to cope with sources (such as attackers) who are non-cooperative. In such environments, isolated fair queuing will waste bandwidth.

So far we have allocated bandwidth statically but have not allowed for statistical multuiplexing. Statistical multiplexing is achieved via bandwidth stealing.

For bandwidth stealing, suppose that the Chicago-Seattle flow in Figure 1 is sending only 10 Mbps during some period. Then while the prescribed allocation for Chicago to Dallas is 20, this can be increased to 30 Mbps during such a period. To implement bandwidth stealing, the input traffic between ingress-egress pairs is measured in intervals of T seconds (we have done experiments with 10 sec and 5 minute intervals) and the demand for the next period is estimated (we use a least-squares estimator that works well on Internet-2 traffic traces). Then link state packets are sent out with the traffic demands for each ingress router. When all LSPs arrive, all routers have the complete topology and the traffic matrix, and calculate the weighted max-min share.

Any traffic arriving that exceeds the calculated fair-share is dropped at the ingress router using a token bucket limiter. Note that dropping at the ingress allows predictable statistical multiplexing. If packets were merely marked, all flows could try and ramp up at the same time, and packets would be dropped, and cause congestion collapse. Thus NetShare also unilaterally prevents congestion collapse in the network even if network slices do not use TCP and do not backoff in the face of congestion in the network.

However, because the allocation for Period N is based on demand for Period $N-1$, TCP flows that react to congestion can be penalized without care. Suppose the traffic from Chicago to Seattle is allocated 10 Mpbs in Period 2 based on similar demand in Period 1. If the traffic tries to grow in Period 2, the limiter at the ingress will drop packets, causing TCP traffic not to grow further. Instead, as in [17], NetShare gives all flows under their fair share ϵ more ($\epsilon = 1.18\%$ works well) than their measured demand. In the above example, Chicago-Seattle will be allocated around 12 Mbps of traffic even if only 10 Mbps is predicted.

The Max-Min fair share algorithm is expensive: it takes $O(f^2)$ where f is the number of flows [5, 3]. Unlike earlier approaches such as ATM flow control [14] that obtain the Max-Min fair share iteratively using a distributed algorithm that converges over time, we compute the max-min share directly on the graph using a modified form of Link State routing. By doing the Max-Min calculation only for Ingress-Egress Pairs (in the order of 100's) as opposed to TCP flows (millions), the resulting computation takes under 100 msec on a vanilla PC for the largest ISPs or data center networks. The efficiency of computing hierarchically is a principal reason we do the calculation in two steps. Further, we can compute tight bounds on the time to redistribute bandwidth $T_{recover}$ a crucial feature for SLAs to customers, even internal customers.

We added some algorithmic machinery to speedup the Max-Min computation Two data structures are critical: one that maps from ingress-egress pairs to links, and one that maps from links to ingress-egress pairs that use the links. These two data structures are built as part of the Dijkstra calculation for route computation. While simple, these optimizations appear to be new.

The algorithm can be modified to handle load balancing by assigning a number of buckets B to each source and mapping each equal cost route (from a source to a destination) to a separate bucket. This effectively increases the number of sources in the algorithm by the number of buckets used for path splitting. However, it allows different slices to be routed on different paths between the same pair of ingress and egress routers, potentially increasing network efficiency.

The Step 1 algorithm requires the following changes to ingress routers. First, the forwarding path must measure

traffic demand (one counter per egress) and enforce traffic limits (a token-bucket per egress). Both hardware features are available in many existing routers. Second, the the router software must be changed to modify LSP sending and computation. We have found a simple incrementally deployable way to do so.

3.2 Step 2: Dividing Virtual Links between Customers and Applications

At the end of Step 1, the network is abstracted into a set of virtual links between ingress-egress pairs. In Step 2, each virtual link is allocated hierarchically among customers/applications as in Class-based Fair Queuing [9]. For example, in Figure 1, assume the network has been virtualized as shown. Now suppose a customer slice has 3 sites in Seattle, Chicago, and Boston. The customer can be allocated 10 Mbps on each virtual link between these 3 locations, by being assigned 1/2 of Seattle to Chicago, 1/4th of Chicago-Boston and 2/5-th of Seattle to Boston.

Mechanistically, this can be implemented (again at the ingress router) by having a queue for each slice or customer and then doing simple fair queuing on each queue (using say Deficit Round Robin). Note that the complexity is linear with the number of customers at each ingress-router for this second stage, while doing the Max-Min fair share directly on customer slices would greatly increase complexity because of the $O(f^2)$ term in the complexity.

4 Detailed Specification of Algorithm

We describe how Step 1 NetShare can be implemented on a routing network such as an ISP or data center network running link state routing (OSPF) by software modifications to ingress-egress routers.

Let P denote the number of ingress-egress routers, E the number of links and D the diameter of the graph. We present an $O(P^2D)$ algorithm for Netshare core algorithm, together with the cost of P Dijkstra computations at each router, which are $O(E)$ apiece using bucket sort techniques (assume the max link cost being a small constant). Thus the total complexity is $O(P^2DlogE + PE)$. If $P < 100$, $D < 10$, and $E < 1000$, this is around $2,000,000$ operations even for a large ISP which seems feasible in software in under a minute assuming a constant factor of less than 600 and a 10 Mips Processor.

The main algorithm run at every ingress router is shown in Algorithm 1.

Algorithm 1 NetShare bandwidth allocation

Input: Network topology from Link State Packets including the bandwidth of each link, a vector $B[O_iO_j]$ for every pair of POPs O_i and O_j (i.e. bandwidths between every pair of Origin-Destination specified by the administrator) and $P[O_i, O_j]$ for the amount of traffic predicted between O_i and O_j by the prediction algorithm.

Output: $NetShareRate[O_iO_j]$ for every pair of POPs O_iO_j

- Step A. Compute P Dijkstra trees for each origin POP P_i to every other POP P_j.
- Step B. Populate two mapping data structures from Edges to OD pairs and vice versa: $EdgetoODPair$ maps an edge E_i to a linked list of OD pairs that use E_i in their shortest paths and $ODPairtoEdge$ maps Origin POP O_i to Destination POP D_j to a list of the (unordered) edges in the shortest path from O_i to D_j. See Algorithm 2.
- Step C. Do the main NetShare flow calculation as follows. For each edge E, define $Active(E)$ to be the set of OD pairs O_iO_j in $EdgetoODPair[E]$ such that $P[O_i, O_j] > B[O_iO_j]$ (in other words these are OD pairs that want more than their allocated bandwidth. Let $Weight(E)$ be initialized to the sum of $B[O_iO_j$ for every pair O_i, O_j in $Active(E)$. Weights are used to divide unused bandwidth in proportion to allocated bandwidths among active OD pairs. Let $B(E)$ denote the current unallocated bandwidth for edge E. $B(E)$ is initialized to the bandwidth of the link minus the currently allocated bandwidth. The currently allocated bandwidth is $\Sigma_{O_iO_j in EdgetoODPair[E]} Min(c * P[O_iO_j], B[O_iO_j])$. Intuitively, this is because if an OD pair wants less than its prescribed bandwidth, we only allocate it what it wants increased by a small multiplier $c > 1$ that allows TCP traffic to grow. However, if it wants more, we start by allocating it its prescribed bandwidth. However, as the algorithm proceeds, any spare bandwidth on edge E will be divided up among the active flows that traverse E.

  The initializations of $Weight(E)$ and $B(E)$ can be calculated as a side effect of Step B. We also keep a boolean vector $Assigned[]$ with one boolean flag for every OD pair and an Integer vector called $IterationCount[]$ with one integer for every edge. There is also an integer $GlobalIterationCount$ that starts at 0

  i. First calculate the current bottlenecked link E_b as the edge with the smallest current value of $B(E)/Weight(E)$ across all edges E using a heap. Increment $GlobalIterationCount$

  ii. The main iteration is as in Algorithm 3. We iterate steps i) and ii) at most E times till $Assigned(O_iO_j)$ is true for all OD pairs.

Algorithm 2 Algorithm to map between edges and ODpairs

i. $EdgetoODPair$: mapping an edge E_i to a linked list of OD pairs that use E_i in their shortest paths. In other words, this is a mapping from edges to the flow equivalence classes (captured by OD pairs) that use the edge in a shortest path.

In particular, calculate this by walking each Dijkstra tree bottom up and for each edge forming its list by concatenating its existing list with the lists of each of its children. For example, suppose the shortest path tree rooted at POP O_1 starts with two links E_1 and E_2. Suppose E_1 has two child links E_3 and E_4 that lead to O_3 and O_4, and O_5 and O_6 respectively. We start bottom up by adding O_1O_3 to E_3's list; adding O_1O_4 to E_4's list; adding O_1O_5 to E_5's list; and O_1O_6 to E_6's list. Then we go up the tree to E_1, we simply add what we have added to E_3 and E_4's list (namely O_1O_3 and O_1O_4).

ii. $ODPairtoEdge$: This is a vector with one element for each OD pair O_iO_j where the element for O_iO_j is a linked list that lists the (unordered) edges in the shortest path from Origin POP O_i to Destination POP D_j, where both i and j lie between 1 and P. In some sense this is the inverse data structure to $EdgetoODPair$ that maps from Edges to OD pairs because this maps from OD pairs to edges. Again this can be done easily by walking each Dijsktra tree top down from the root O_i to all destinations O_j and simply adding each edge encountered to the list.

Algorithm 3 Compute NetShare Rates

for each OD pair O_iO_j in $EdgetoODPair[E_b]$ **do**
  if $Assigned[O_iO_j] = False$ **then** {OD pair not assigned so far, OK to update}
    $NetShareRate[O_iO_j] \leftarrow B(E_b)/Weight(E_b) * B[O_iO_j]$
    for each edge E in $ODPairtoEdge[O_iO_j]$ **do**
      if $IterationCount[E] < GlobalIterationCount$ **then** {Edge E not updated in this iteration}
        $IterationCount[E] \leftarrow GlobalIterationCount$ {mark edge as updated}
        $Weight[E] \leftarrow Weight[E] - B[O_iO_j]$ {Bandwidth weight Update Step}
        $B(E) \leftarrow B(E) - NetShareRate[O_iO_j]$ {Bandwidth Update Step}
        Update heap to reflect new value of $B(E)/Weight(E)$ {Heap adjust}
      end if
    end for
    $Assigned[O_iO_j] \leftarrow True$ {/* we have assigned bandwidth to the OD pair from O_i to O_j */}
  end if
end for

4.1 Complexity analysis

- Step A. The Dijkstra tree calculation is $O(PE)$ using bucket sort (Dial's algorithm)[**?**], $O(PElogE)$ with standard heap or $O(P(E + PlogP))$ using Fibonacci heaps. Recall P is the number of ingress/egress routers.

- Step B.

  i. $EdgetoODPair$: The complexity seems difficult to calculate at each operation individually but observe that each operation adds one element to some edge list. Thus the number of operations is bounded by the number of elements in $EdgetoODPair$. But O_iO_j can appear in $EdgeODPair[E]$ if and only if E is on the shortest path from O_i to O_j. Thus there is a 1-to-1 correspondence between elements of $EdgeODPair$ and edges in the union of shortest paths for all pairs of POPs. But there are at most D elements (where D is the diameter) in each shortest path, and at most P^2 pairs. Thus this step is $O(P^2D)$

  ii. $ODPairtoEdge$: Each shortest path has at most D edges and there are at most P^2 pairs, so this step is also $O(P^2D)$.

- Step C.

  i. Heap operation is $O(ElogE)$ using a standard heap. This is subsumed by $O(PElogE)$

  ii. Every OD pair is updated just once but an edge E can be updated many times for as many times as it is in $ODPairtoEdge$. But summed over all edges, the number of edge updates is at most the elements in $ODPairtoEdge$ again which is P^2D again (which is a more refined bound than the naive bound of $O(E^2)$. If each update requires a $logE$ heap update, this $O(P^2DlogE)$.

The actual bounds may be better in practice on real topologies because the Flow Update step is only taken if an edge is not already updated. Thus if several flows share an edge, the number of updates will be less than P^2D. Next, the diameter D is a coarse bound. Perhaps a better bound is formed by the average length (not worst case) of a shortest path because we are taking sums across the lengths of all shortest paths. Finally, the number of iterations in practice may bqe small if there are only a small number of bottlenecks. In the evaluation, we will run the algorithm on real Internet topologies to find better estimates for running times. However, the bottom line is that even the theoretical running times are fairly good.

4.2 Statistical Multiplexing via Bandwidth Stealing

Algorithm 1 requires a prediction of the amount of traffic expected in the next interval between every pair of ingress and egress routers. This is essential to providing statistical multiplexing because it allows inactive pairs to be predicted as having traffic lower than their allocated bandwidth, which can then be redistributed to other OD pairs using Algorithm 3.

To calculate $P[O_iO_j]$, each POP router measures the traffic to every other POP router in measurement intervals of T seconds. This is used to predict the amount of traffic that will be used in the next interval. In essence, the next iteration of the Max-Min fair calculation will then use the predicted (and not the allocated) values, if these values are smaller than the allocated. These values are simply subtracted from the link bandwidths of all links in the path of each pair of POPs. The Max-Min fair calculation is then used to divide up the "excess bandwidth" among all flows that are predicted to need more than their allocated bandwidth.

There are many possibilities for prediction schemes. We adopt a standard linear least squares predictor to fit the measurement data points to a linear model. Assume we have n measurement data points (equally separated in time) $v_1, v_2, ..., v_n$ and we want to predict the next point v_{n+1}. Then we find a linear line $v = \alpha_1 + \alpha_2 \cdot t$ where $t = 1, 2, ...n$ that best fits the n points. We finally compute the prediction as the value at $t = n+1$, i.e. $v_{prediction} = \alpha_1 + \alpha_2 \cdot (n+1)$.

Standard results show that the optimal values of α_1 and α_2 satisfies the normal solutions $A^TA\widehat{\alpha} = A^TV$ where $A = \begin{bmatrix} 1 & 1 & ... & 1 \\ 1 & 2 & ... & n \end{bmatrix}^T$ and $V = \begin{bmatrix} v_1 & v_2 & ... & v_n \end{bmatrix}^T$. That gives $\widehat{\alpha} = \begin{bmatrix} \widehat{\alpha_1} \\ \widehat{\alpha_2} \end{bmatrix} = B \cdot \begin{bmatrix} S_n \\ T_n \end{bmatrix}$ where $B = \left(\begin{bmatrix} n & \sum_{k=1}^{n} k \\ \sum_{k=1}^{n} k & \sum_{k=1}^{n} k^2 \end{bmatrix} \right)^{-1}$, $S_n = \sum_{k=1}^{n} v_k$, and $T_n = \sum_{k=1}^{n} kv_k$

Note that the hardware only needs to measure the traffic $M[O_iO_j]$ sent in an interval. Such measurement at an ingress router is already provided in most routers especially for a small number of egress-routers. The measurement is then passed to software that keeps the history and does the matrix multiplication. Note that we get good results using a history of 5 measurements. Thus the matrix sizes are small and led to fast software implementations. Note also that if the number of history values n is fixed at the beginning, then B is also fixed and can be pre-computed. Also S_n and T_n can be efficiently computed in an online manner as shown in Algorithm 4.

Armed with the traffic predictions, $P[O_iO_j]$, POP nodes perform *bandwidth stealing* as follows. If the predicted traffic $P[O_iO_j]$ is less than the prescribed traffic $B[O_iO_j]$ then then the OD pair is only allocated the smaller of $c * P[O_iO_j$ and $B[O_iO_j]$. c is a small multiplier (in our experiments we use $c = 1.18$ that is used to give an OD more slightly more than we predict that it may need.

The multiplier c is essential to allow TCP flows to grow if they wish to. If we give an OD exactly what it had in the previous interval for example, and the OD pair consists of a single TCP flow, the TCP flow will cut its rate to match the allocated bandwidth and not grow even if it had data to send. This is crucial because it affects the time $T_{recover}$ it takes for an OD pair to reclaim its prescribed bandwidth after a period of using less than its prescribed bandwidth. Thus even if the OD pair is a self-regulating traffic stream like TCP, the OD pair can ramp up at the rate of 1.18 every T seconds. For example, for $c = 1.18$, $T_{recover} = 5T$. If T is 1 second, then $T_{recover} = 5$ seconds.

5 Implementation

We have implemented NetShare by extending the existing link state routing protocol in NS2. A brief review of the link state protocol implementation in NS2 is as follows. Each NS2 node in the network is attached to a link state routing object. The link state routing object is responsible for monitoring link status with directly connected neighbors, initiating link state packets (LSPs) and also intercepting and analyzing LSPs from another nodes. The existing implementation already provides mechanisms to handle LSPs and build shortest paths with Dijkstra's algorithm and an engine to handle buffering and retransmission of LSPs.

We modified the NS2 link state framework to implement Netshare as follows. Initially the network administrator defines each OD pair by specifying the *bandwidths* for each pair of ingress and egress routers. In other words, an OD pair is a tuple of ingress router, egress router and prescribed bandwidth. We implement Netshare by embedding Netshare metadata inside LSPs. For instance, in an LSP each link between two nodes contains the IDs of nodes defining the link, together with the cost, bandwidth and status (up/down) of the link. We embed the data for an OD pair O_i, O_j in an LSP by treating an OD pair as a special type of link with cost and bandwidth entries representing the bandwidth $B[O_iO_j]$ and predicted traffic $P[O_iO_j]$ respectively. An OD pair is distinguished from a regular link using a special flag bit in its link status.

Since bandwidth enforcement and measurement is only done at ingress/egress routers, the administrator needs to designate certain routers as ingress routers. We assume every ingress router is also an egress router. Other routers continue to operate normally. An ingress router sends its measured traffic to all other ingress as OD pair data in a Link State packet that is broadcast to all other routers. As with regular link state route computation, an ingress router uses the database of packets broadcast by other ingress routers to compute the NetShare rates (Algorithm 1) along with the Dijkstra algorithm.

The NetShare rates are used to enforce max-min fairness between OD pairs. When a packet arrives at an ingress router, the ingress router reads the destination address of the packets, maps the destination address via the normal prefix lookup table to an index into a destination POP table. The index into a destination POP table points to a token bucket that is initialized to the NetShare rate as its limiting rate. Note that all modern routers today have token bucket limiting hardware and the mapping from a destination address to a given token bucket can easily be done with existing routers. Thus it is only the propagation of the OD pair info and the calculation of the NetShare rates that need router changes, and these are software changes. As described earlier, each ingress router also needs to measure the traffic sent to each egress router to calculate $M[O_iO_j]$ but such counters already exist in hardware.

Upon receiving an LSPs, an ingress router checks to see if there is any change in network topology, or if the LSPs contain updates of traffic measurements from other ingress routers. In the first case, the ingress router recomputes shortest path trees and max-min fair shares for all OD-pairs. Otherwise in the second case, the ingress router only performs Step C of the NetShare algorithm (Algorithm 1).

The implementation uses a token bucket policer to rate limit the traffic of an OD pair. Due to the sawtooth behavior of TCP, it is well-known in the literature that to avoid multiple consecutive discards, the token bucket policer should have a bucket depth of at least $RTT \times configuredrate$. The bucket depth should not be very large to limit short term bursts.

5.1 Hardware Implementation

We did an implementation of NetShare in the NetFPGA platform. While existing routers have the hardware to do measurement and token buckets, we had to add it to the NetFPGA implementation. We simply modified the output interface returned by the route lookup to also return an index into a table of token bucket limiters. Adding the basic hardware to measure traffic and the token bucket limiter took very little hardware and barely added 2 cycles to the overall delay. We did most of our experiments with the NS2 implementation because it was easier to generate traffic and simulate larger topologies.

6 Evaluation

We performed experiments to answer the following questions.

- How fast can the basic NetShare algorithm run? This affects the basic interval T and the bandwidth recovery time $T_{recover}$.
- How well does the traffic prediction algorithm perform at predicting future traffic from past traffic? Poor prediction will affect statistical multiplexing gains.
- How well does NetShare do at its twin goals of achieving statistical multiplexing and providing bandwidth firewalls especially compared to existing solutions.
- Is NetShare needed for real Data Center applications?

6.1 NetShare Calculation Speed

In this section, we show that the basic NetShare calculation takes less than 100 msec even on large networks. This is a crucial parameter as it affects the time it takes for a slice to regain any bandwidth lost because it did use its prescribed bandwidth. It also affects the time for a slice to be granted a proportional share of any bandwidth unused by other slices.

In the absence of standard data center topologies, we modeled a generic "large" network" we used the CSAMP archive [**?**] that provides eight topologies of eight important ISPs including Telstra(AS#1221), Sprint(AS#1239), NTT(AS#2914), Tiscali(AS#3257), Level3(AS#3356), AT&T(AS#7018), Geant and Abilene.

Table 1 shows the microbenchmark experiments running on a standard Intel Core2Duo 3GHz desktop. We observe that Netshare adds around 20% to 50% to the cost of a basic Dijkstra shortest paths computation.

To understand how the cost of Netshare increases as the network scales up, we used the Orbis [**?**] topo generator to scale up the large ISPs (i.e. all except Abilene and Geant) up to around a factor of ten times. To facilitate our purposes, we had to make some modifications to the topology generated by Orbis as follows. First, Orbis does not annotate weights to the links in the scaled topologies. So we adopted a simple weight annotation technique by assigning weights to links in the scaled topology as drawn from the distribution of weights in the original topology. Second, we noted that all topologies except Geant have only symmetric links. In fact Geant has only one link that is unsymmetric (symmetric link means $weight(n_1, n_2) = weight(n_2, n_1)$ for two nodes n_1 and n_2). So we modified Orbis-generated topologies to make all links symmetric.

Table 2 shows that the time is again dominated by the cost of computing Dijkstra's trees. Compared to Table 1, the number of POPs is increased by two orders of magnitude. Thus the timings for Netshare also increase by the same factor. However, we get much better timings for certain topologies (e.g. Telstra, Sprint).

6.2 Traffic Prediction Accuracy

In Netshare, bandwidth stealing is used to allocate unused bandwidth to other slices. Bandwidth stealing is based on traffic prediction. In this section, we investigate how well we can predict traffic from one POP to another POP.

Once again in lieu of any specific data center traffic repository, we analyze Internet2 [**?**] measurement traffic data from two weeks in December 2006. We do recognize that Internet 2 traffic is more highly aggregated and hence more predictable.

Internet2 core routers log all traffic as raw Netflow traces that is made available through the Internet2 Observatory project. Note that raw Netflow traces are sampled at a uniform rate of 1 every 100. Accessing the CSAMP [**?**] data archive, we were able to retrieve the POP-to-POP matrix traffic between 11 POPs in Internet2 topology. Each Internet2 POP is denoted by the city where the POP is situated, i.e. Atlanta, Chicago, Denver, Houston, Indianapolis, Kansas City, Los Angeles, New York City, Seattle, Sunnyvale and Washingto DC. For each pair of POPs, we aggregate the traffic in units of 5 minutes. We chose 5 minutes because we could not get finer grain traffic measurements from the archive.

Thus the key question is that how accurately can the NetShare algorithm predict subsequent traffic from itself to each other POP within the next time interval (e.g. 5 minutes).

We analyzed the Internet2 data and computed the prediction errors when using the prediction algorithm shown in Algorithm 4. We use a simple formula to represent prediction error: $error = |real\_measurement - prediction|/real\_measurement$.

The results are fairly uniform for different POPs and days. Due to space limitations, we present the result for one day (2006-12-08) of data for Seattle POP in Figure 2 and 3. Figure 2 illustrates the pattern of real traffic and prediction. We aggregate measurement data every 5 minutes and keep a history of $n = 5$ measurements. We observe that the prediction algorithm has good performance in matching with the real traffic fluctuation. The number of history measurement n is a performance knob available to the network administrator to fine-tune the prediction algorithm to the traffic pattern of the network. A large value of n tends to "overfit" the data, i.e. stressing on long-term trend to avoid short-term busts. Meanwhile a small value of n tends to "underfit" the data, i.e. quickly adapting to short-term changes.

Figure 3 shows distribution of prediction error. Note that the prediction to destination Sunnyvale is quite poor even though the prediction line is very close to real measurement in Figure 2. This may be caused by the fact that traffic to Sunnyvale is very small and hence the prediction error artificially appears to be too high because we use relative error metric. Other than Sunnyvale, we have satisfactory performance for other POPs.

Table 1: Micro benchmark of NetShare computation on real ISPs

Topology	# nodes	# edges	# POPs	# Dijkstra SPF(ms)	# Netshare(ms)
Telstra(1221)	44	88	1892	40	18
Sprint(1239)	52	168	2652	41	23
NTT(2914)	70	222	4830	58	51
Tiscali(3257)	41	174	1640	32	12
Level3(3356)	63	570	3906	67	28
AT&T(7018)	115	296	13110	131	72
Geant	22	67	462	17	4
Abilene	11	28	110	10	2

Table 2: Micro benchmark of NetShare computation times for $10X$ scaled topologies

Topology	# nodes	# edges	# POPs	# Dijkstra SPF(sec)	# Netshare(sec)
Telstra(1221)	439	908	192,282	1.93	0.44
Sprint(1239)	559	2030	311,922	4.0	0.9
NTT(2914)	720	2448	517,680	10.9	2.4
Tiscali(3257)	438	1870	191,406	4.3	0.33
Level3(3356)	626	6368	391,250	5.7	1.3
AT&T(7018)	1134	2992	1,284,822	43	7.3

6.3 Bandwidth Firewalls and Statistical Multiplexing

Figure 4 shows a simple network topology to analyze baseline performance of Netshare scheme. There are four ingress/egress routers $P1, P2, P3$ and $P4$, and one interbal router $C1$. $S1, S2$ are sources and $D1, D2$ are traffic sinks, both potentially VMs on physical machines.

For simplicity, we used only three OD pair flows (we call them virtual flows) are defined: the first we call $VF1$ from $P1$ to $P3$ is assumed to be routed along the path $P1-C1-P3$. Similarly, $VF2$ is from $P1$ to $P2$ and is routed along the path $P1-C1-P2$. The third OD pair flow is called $VF3$ from $P4$ to $P3$ and is routed along the path $P4-C1-P3$.

Note that $VF1$, $VF2$ share link $P1-C1$ and $VF2$, $VF3$ share link $C1-P3$. There are three slices: $S1-D1$, $S2-D2$ and $S3-D3$ which are mapped to the OD flows $VF1, VF2$ and $VF3$ respectively.

We let the two bottleneck links $P1, C1 = 10$ Mbps, $C1, P3 = 20$ Mbps. The other links are set to be very large bandwidths. Assume the manager allocates the OD pair bandwidths as $VF1 = 5Mbps$, $VF2 = 15Mbps$ and $VF3 = 5Mbps$. We model the only slice using $VF1$ as a UDP virtual flow that can send packets at an arbitrarily high rate. On the other hand, the slices corresponding to $VF2$ and $VF3$ are modeledas TCP flows.

Figure 5a demonstrates that Netshare is able rate limit both UDP and TCP POP-to-POP traffic flows to their prescribed rate. Furthermore, Netshare also effectively reacts to network dynamics as follows. In Figure 5a, at $t = 200s$ we make a change to the topology such that $VF1$, being a high rate UDP stream, is instead bottlenecked at $1.5Mbps$ prior to entering the core network. As shown, $VF2$ and $VF3$ are immediately ramped up to steal the amount of extra bandwidth otherwise wasted. At $t = 400s$, $VF1$ resumes and then it is able to claim back its resource quota while $VF2$ and $VF3$ both fall back to their fair shares. Note that between 200 sec and 400 seconds, the total throughputs at link $P1-C1$ and $C1-P3$ are slightly less than their capacity. The reason is that Netshare reserves a small throughput allowance for the bottlenecked $VF1$ so that it can gradually claim back its fair share any time if necessary.

By contrast, Figure 5b shows the bandwidth allocation without NetShare and and Figure 5c shows the allocated bandwidth without NetShare but instead using individual fair queueing at each router. As $VF1$ sends a UDP stream at an arbitrarily high rate, it is able to dominate the whole network (maximum $10Mbps$ of bottleneck link $C1-P3$ in (b) and can use 40% more than its allocated bandwidth in (c)). Even worse, $VF1$ pushes other legitimate OD pairs to complete collapse in (b). By contrast, NetShare provides bandwidth firewalls even in the face of uncooperative slices. Note that even in the data center, new untested applications may not use TCP, or use multiple TCP connections.

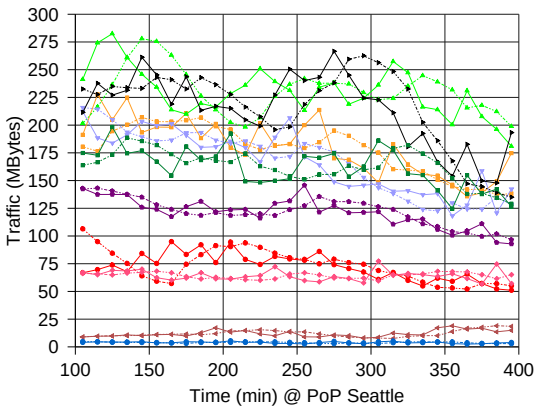


Figure 2: Traffic from Seattle - Solid lines denote measurement; dashed lines denote prediction

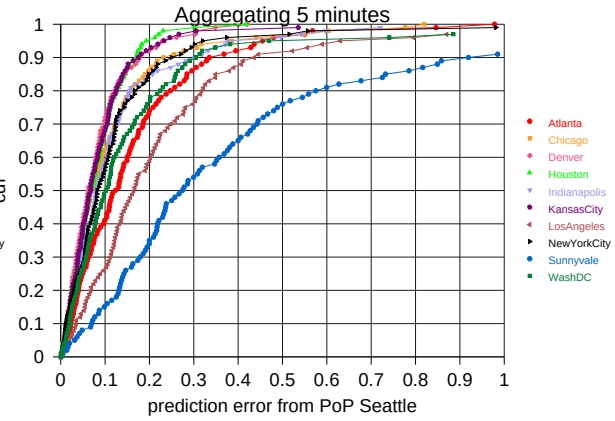
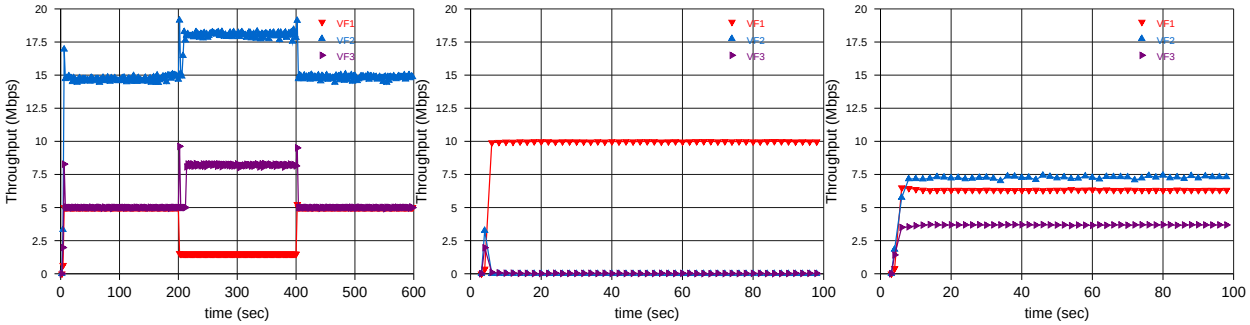


Figure 3: Seattle - Traffic prediction error



(a) Netshare **(b) Without individual link fair queuing** **(c) Fair queuing at all individual links**

Figure 5: Netshare efficacy: (a) illustrates stable flow of traffic at prescribed rates and statistical multiplexing for maximal network resource usage (b) congestion collapse for legitimate flows VF2 and VF3, and (c) shows that individual fair queuing is inefficient and causes unfair resource utilization

Individual fair queuing at all links helps avoid congestion collapse, but as shown in Figure 5c, legitimate flows $VF2$ and $VF3$ are only at 50% and 60% of their their allocated bandwidths. Also the useful bandwidth utilization at $P1-C1$ is only 70%. Of course, this is because link by link fair queuing does not guarantee a *network-wide* optimal distribution of unused bandwidth as does Netshare's Max-Min algorithm.

6.4 NetShare and MapReduce Applications

In this section, we study whether real cloud applications can affect each other, hence requiring the services of a bandwidth allocation mechanism like NetShare to keep each application predictable. Note that real data centers (e.g., Google, Walmart) will very likely host multiple concurrent applications (e.g., MapReduce instances, data mining instances). We use only the simplest experiment to demonstrate potentially interference effects and how they can be removed using NetShare.

Our experiment uses an implementation of MapReduce on top of *Mortar* - a distributed stream processor being developed at UCSD. In a nutshell, Mortar allows users to continuously query data from distributed sources. Examples of such continuous queries include status queries in a sensor network, live data archiving (e.g. wireless network logging, web crawler, etc.) and the calculation of the the average CPU utilization across a testbed (e.g. a cluster or Planetlab) every minute.

Mortar users can implement a simple interface to write

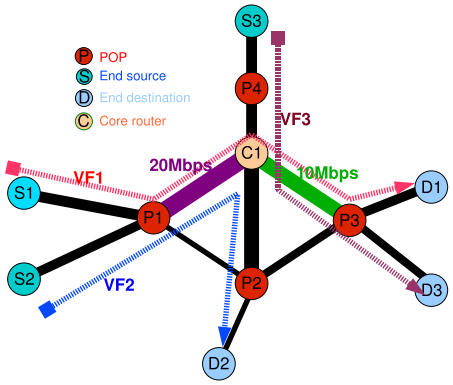


Figure 4: Network topology for NS2 experiments

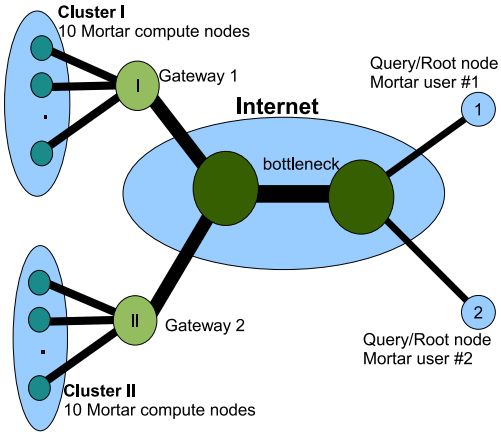


Figure 6: Network topology used for Netshare/Mortar experiments

custom stream operators and the system itself manages the installation of the operators on the data sources and the flow of data between them. To achieve efficiency and scalability, Mortar organizes operators in in-network aggregation trees, to build scalable queries, and tries to provide accurate results when failures happen. The Mortar interface is used to write generic Map and Reduce operators that a user can extend to write custom Map and Reduce operators in a way similar to the Mappers and Reducers in Hadoop. Mortar manages the installation of the Map and Reduce operators, organizing them in aggregation trees since a reduce operation is used most of the times an aggregate operation.

We experimented on a Mortar testbed on a 30-node cluster. One node is a FreeBSD ModelNet core [**?**] that emulates network dynamics. The other nodes are virtual nodes available for running Mortar application instances. We have tested a number of scenarios in Mortar usage: unfair usage between different users on the same Mortar instance; unfair usage between different instances of Mortar applications and interference with external traffic. Unfairness can be caused by one user issuing a higher rate of queries. Though Mortar uses TCP, a user that issues more queries will open more TCP connections and so can get an undue share of network bandwidth.

Also note that some queries are intrinsically more expensive than the others. For example, a "sum" on numerical data can be aggregated on the fly in the network and hence it is much less resource intensive than a string union operator, which attempts to concatenate all data and send back to the user.

In the experiment, Netshare policies are implemented and enforced at the stub routers. Figure 6 shows our experiment setup on Mortar for the first scenario. We have two users competing for common resource nodes by making queries to Mortar.

Our implementation of Netshare framework on top of Mortar cluster comprises of scripts that continually monitors the bandwidth of all ModelNet links and reacts accordingly to enforce fairshare. Figure 7 illustrates the effect of Netshare policy to enforce fair share of cluster resources upon its activation at $t = 200s$. Even though Netshare enforces the traffic allocation at network and packet level, this translates to application throughput. For the application measure of effectiveness, we use the data goodput at the application level.

The experiment demonstrates that application instances can interfere and one cannot rely on TCP fairness mechanisms because they cannot be tuned, and many applications use multiple TCP connections. It also shows that a mechanism like NetShare can control this interference. Note that for the simple topology shown in this experiment, simple

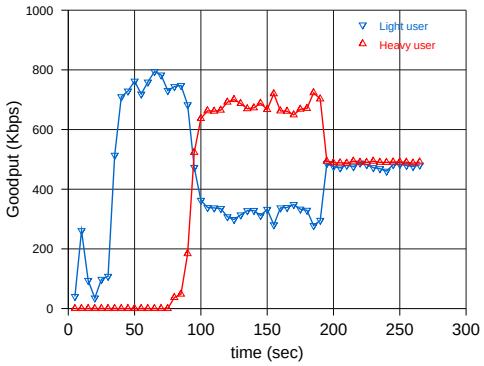


Figure 7: Netshare in action on Mortar. A light user starts first but later (at $t = 80s - 100s$) is dominated by a heavy user as the latter ramps up its rate of queries. At $t = 200s$, Netshare policy is enforced at stub routers and equal bandwidth allocations are enforced between the two users.

	Bandwidth Guarantees	Statistical Multiplexing Guarantees	Simple Abstraction	Software Changes Only?	Tine to react to change
VPNS using RSVP Infiniband	Yes	No	Yes	Exists	1 RTT
ATM Flow Control XCP	No	Muxing but no guarantees	Not for BW Allocation	Header Changes	Several RTTs Non−deterministic
CoreStateless FQ	link by link only	link by link only	Yes	Header Changes	Several RTTs Non−deterministic
Traffic Engineering OSPF−TE, TexCP	No	No, routes traffic efficiently	N/A	Yes	Mostly hours
DRL	BW Cap only In & out of cloud	Yes, but no guarantees	Yes	Yes	Several RTTs
NetShare	Yes, for networks	Yes, for networks	Yes	Yes	1 RTT + O(P^2) Max−Min computation

Figure 8: Previous work contrasted to NetShare in terms of NetShare goals

fair queuing at the bottleneck would suffice. However, it is easy to replace the topology by a slightly more complex one and add a third application. In such scenarios, link by link fair queuing will not provide optimal allocations. Further, it will require complex packet classification within the network (to identify applications) instead of doing so at the edge.

7 Related Work

Figure 8 summarizes previous work with respect to the goals we outlined. We now contrast our approach to earlier work in more detail.

Our paper looks superficially similar to Distributed Rate Limiting or DRL [**?**]. NetShare differs from DRL which controls bandwidth *in and out* of the cloud as opposed to *within* the cloud. NetShare and DRL can be combined fruitfully. Further, DRL's best results are obtained [17] cooperative sources while NetShare deals with adversarial sources.

RFC 3630 (Traffic Engineering Extensions to OSPF) proposes using constrained shortest path routing (CSPF) to pick a path that meets bandwidth constraints. Bandwidth can be reserved by RSVP, and packets can be routed along this path in loop-free fashion using MPLS.

Suppose VPN A and B ask for 10 Gbps each on a common path between Seattle and Atlanta where there is a shared 20 Gbps link. If VPN A is idle, ideally we would like VPN B to use up to 20 Gbps. However, if VPN B and A both send at 20 Gbps, their traffic will be dropped at the shared link, potentially leading to congestion collapse. A form of statistical multiplexing is allowed by marking traffic exceeding the VPN's share (i.e., 10 Gbps) and dropping marked packets in times of congestion. Unfortunately, this requires a policy for deciding which of VPN A's packets to mark, or the dropped marked packets will reduce throughput for VPN A.

Measurement-based admission control [13] shares many of our goals of combining statistical multiplexing with some bandwidth guarantees. However, measurement based really applies to one path. When the input traffic is not being used, signalling is used to reduce the demand on the path. However, there is no corresponding signalling to other affected nodes to tell them that bandwidth is now available together with a network wide view of allocation as we do with Max-Min fairness.

The multiplexing problem with the pipe model led to a proposal for a hose model [8] by Duffield et al that provides more flexibility for multiplexing over the pipe model by allowing the customer to specify its aggregate demand from each endpoint to and from the network (2 N specifications as opposed to N^2. The hose model extends multiplexing to all the pipes within a hose and requires fairly complex algorithms (which can increase responsiveness) to realize their optimization objective.

The network slicing model in this paper by contrast allows multiplexing across all pipes in the network. If the network measures that VPN A is not using its bandwidth and can signal this to B, then VPN B can send at higher than its share without causing congestion. Further, the underlying optimization is a simple Max-Min fair share algorithm that can be implemented efficiently.

In NetShare, instead of solving the joint routing and bandwidth allocation problem, we assume that routing is fixed by OSPF: NetShare then allocates bandwidth among ingress-egress pairs based on adminstrator specified weights using Max-Min fair sharing. Thorup and Rexford show that there is considerable flexibility to change routes by chang-

ing OSPF weights [10] without going to the effort of pinning every route using MPLS. In the rare event that there is a bandwidth need that cannot be fitted by tweaking the OSPF weights and Max-Min fair share weights we provide, then RFC 3630 can be used to override our mechanism.

Traffic engineering deals with more efficiently handling an existing traffic matrix, for instance by minimizing delays or the worst-case link utilization. For instance, OSPF-TE [**?**] tunes link weights to optimally route traffic. TexCP [**?**] goes further and allows load balancing over multiple paths. However, traffic engineering does not have notion of bandwidth firewalls between pipes. If a pipe from I to O misbehaves and sends over capacity, traffic engineering will do its best to route the traffic optimally as opposed to drop the packet at the edge. [**?**] suggests doing both traffic engineering via load balancing and source adaptation. However, it requires a clean-slate approach. Further, it is unclear what notion of fairness it supplies across pipes.

Note that conventional traffic engineering also deals primarily with network efficiency and improving the utilization of links in the networks. But this can fly in the face of user needs. For example, if we have a chain of links L1, L2, L3 and User 1 wants to use all 3 links while users 2 through 4 want to use L1, L2, and L3 respectively, max efficiency from the network point of view is obtained by not scheduling User 1. Thus optimality of the network per se is the wrong point of view without considering the desired traffic matrix.

Several congestion avoidance algorithms (e.g., XCP [15] calculate the Max-Min fair share iteratively based on signalling from the network. The bottleneck conveys its rate, sources adjust, and the process iterates till convergence. Examples proposed for ATM flow control include [12, 6, 11].

Fair queueing [7] by itself does not guarantee a Max-Min fair share. Further, it requires per-flow state. Core-stateless fair queueing (CFSQ) [20] avoids state in core routers but still does not calculate the fair-share. Several papers enhance CSFQ to calculate the fair-share including CoreLite [19], token-based congestion control [18], and [16]

All these papers assume *coooperative* sources, header modifications to add markings and feedback, and require several iterations (each of which is a network delay) to achieve convergence. Ostfeld [3] and [6] show that the time taken for convergence for such distributed algorithms can be much larger than expected. Our algorithm by contrast: does not assume responsive flows, is implemented by edge routers, requires no modification to packet headers to communicate marks, and takes 1 network delay to calculate fair share plus the computation time.

Theoretically, we could use either XCP or the ATM flow control algorithms between ingress-egress pairs. However, it is not clear how to tune to provide specific bandwidths for each pair of flows. Some do not allow weights, and even if they do, it is easy to show that specifying weights for every pair of ingress-egress routers cannot be used to specify arbitrary traffic matrices as NetShare can. Further, the distributed computation does not have precise and fast time bounds for convergence. This is crucial for ensuring a small $T_{recover}$.

Centralized algorithms for Max-Min Fair Share date back to [5]. Several papers (e,g,. [4]) focus on efficient approximations. We achieve reasonable complexity without resorting to approximations because we do the Max-Min Fair Share algorithm between ingress-egress pairs, and do it over large time windows (seconds). Using larger time windows also provides more stable measurements of demand.

8 Conclusions

While cloud computing is emerging as a strong phenomenon, adoption by businesses will require attention to performance and security guarantees. This paper focuses on performance guarantees for business cloud computing, guarantees that can be the basis for Service Level Agreements. Current cloud computing vendors offer computing and storage guarantees (if at all) but the notion of a virtual data center requires *both* computing and bandwidth guarantees. We hope that NetShare will will provide a new paradign for bandwidth allocation within clouds, be they data centers or ISPs. As a side effect, our NetShare allocation scheme provides resilience in the face of uncooperative (malicious, selfish, buggy) sources and statistical multiplexing.

9 Work we need to do for SOSP

- Add data center topologies as in Amin's paper.
- Add realistic workloads
- Add load splitting via ECMP. Some issues here.
- Can we extend the current "pipe" model (point to point) in NetShare to a hose model (point to multipoint).
- Can we find numbers on data center application usage. Bob Feldermann said data center is 1% utilized. Can we get a number and models to show potential multiplexing gain.

- Can we show that NetShare may be applied to an ISP where customers lose bandwidth when not using it, but in turn can share others unused traffic. To discourage gaming and free-loaders, perhaps can implement a Bit Torrent like tit-for-tat scheme,

References

[1] Cloud Computing. Technical report.

[2] The Virtualization of Infrastructure Optimization. Technical report.

[3] Yehuda Afek, Yishay Mansour, and Zvi Ostfeld. Convergence complexity of optimistic rate based flow control algorithms (extended abstract). In *STOC '96: Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 89–98, New York, NY, USA, 1996. ACM.

[4] 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.

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

[6] 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.

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

[8] 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.

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

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

[11] 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.

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

[13] S. Jamin, P. Danzig, S. Shenker, and L. Zhang. A measurement-based admission control algorithms for integrated services. Technical report, 1995.

[14] S. Kalyanaraman. The erica switch algorithm for abr traffic management in atm networks,. *IEEE/ACM Transactions on Networking*, 2000.

[15] 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.

[16] Jung-Shian Li and Jing-Zhi Liang. A novel core-stateless abr-like congestion avoidance scheme in ip networks: Research articles. *Int. J. Commun. Syst.*, 18(5):427–447, 2005.

[17] B. Raghavan and et al. Cloud control with distributed rate limiting. In *In the Proceedings of SIGCOMM 2007*, 2007.

[18] 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.

[19] Raghupathy Sivakumar, Tae eun Kim, Narayanan Venkitaraman, Jia ru Li, and Vaduvur Bharghavan. Achieving per-flow weighted rate fairness in a core stateless network. In *IEEE Conference on Distributed Computing Systems 2000*, pages 188–196, 2000.

[20] 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.

[21] J. Turner. A proposed architecture for the geni backbone platform.

Algorithm 4 Least square-based prediction algorithm

Input: New measurement $M[O_iO_j]$

Output: Prediction of traffic for next interval $P[O_iO_j]$

Precompute matrix B

$T_n \leftarrow T_n - S_n + n \cdot v$

$S_n \leftarrow S_n - v_1 + v$

Substitute $M[O_iO - j]$ for v_1 in the circular buffer of traffic history measurements

$\begin{bmatrix} \widehat{\alpha_1} \\ \widehat{\alpha_2} \end{bmatrix} \leftarrow B \cdot \begin{bmatrix} S_n \\ T_n \end{bmatrix}$ RETURNS $\widehat{\alpha_1} + \widehat{\alpha_2} \cdot (n+1)$