End-to-end Performance Isolation Through Virtual Datacenters

Sebastian Angel, *The University of Texas at Austin;*
Hitesh Ballani, Thomas Karagiannis, Greg O'Shea,
and Eno Thereska, *Microsoft Research*

https://www.usenix.org/conference/osdi14/technical-sessions/presentation/angel

End-to-end Performance Isolation through Virtual Datacenters

Sebastian Angel*, Hitesh Ballani, Thomas Karagiannis, Greg O'Shea, Eno Thereska

Microsoft Research   **The University of Texas at Austin*

Abstract

The lack of performance isolation in multi-tenant datacenters at appliances like middleboxes and storage servers results in volatile application performance. To insulate tenants, we propose giving them the abstraction of a dedicated *virtual datacenter* (VDC). VDCs encapsulate end-to-end throughput guarantees—specified in a new metric based on virtual request cost—that hold across distributed appliances and the intervening network.

We present *Pulsar*, a system that offers tenants their own VDCs. Pulsar comprises a logically centralized controller that uses new mechanisms to estimate tenants' demands and appliance capacities, and allocates datacenter resources based on flexible policies. These allocations are enforced at end-host hypervisors through multi-resource token buckets that ensure tenants with changing workloads cannot affect others. Pulsar's design does not require changes to applications, guest OSes, or appliances. Through a prototype deployed across 113 VMs, three appliances, and a 40 Gbps network, we show that Pulsar enforces tenants' VDCs while imposing overheads of less than 2% at the data and control plane.

1   Introduction

In recent years, cloud providers have moved from simply offering on-demand compute resources to providing a broad selection of services. For example, Amazon EC2 offers over twenty services including networked storage, monitoring, load balancing, and elastic caching [1]. Small and enterprise datacenters are also part of this trend [56, 59]. These services are often implemented using *appliances*, which include in-network middleboxes like load balancers and end-devices like networked storage servers. Although *tenants* (i.e., customers) can build their applications atop these services, application performance is volatile, primarily due to the lack of isolation at appliances and the connecting network. This lack of isolation hurts providers too—overloaded appliances are more prone to failure [59].

We present *Pulsar*, the first system that enables datacenter operators to offer appliance-based services while ensuring that tenants receive guaranteed end-to-end throughput. Pulsar gives each tenant a *virtual datacenter* (VDC)—an abstraction that affords them the elasticity and convenience of the shared cloud, without relinquishing the performance isolation of a private datacen-

ter. A VDC is composed of virtual machines (VMs), and resources like virtual appliances and a virtual network that are associated with throughput guarantees. These guarantees are independent of tenants' workloads, hold across all VDC resources, and are therefore end-to-end.

Providing the VDC abstraction to tenants presents two main challenges. First, tenants can be bottlenecked at different appliances or network links, and changing workloads can cause these bottlenecks to shift over time (§2.1). Second, resources consumed by a request at an appliance can vary based on request characteristics (type, size, etc.), appliance internals, and simultaneous requests being serviced. For example, an SSD-backed filestore appliance takes disproportionately longer to serve WRITE requests than READ requests (§2.4). This behavior has two implications: (i) the *capacity*, or maximum achievable throughput, of an appliance varies depending on the workload. This is problematic because the amount of appliance resources that can be allocated to tenants becomes a moving target. (ii) Standard metrics for quantifying throughput, like requests/second or bits/second, become inadequate. For example, offering throughput guarantees in request/second, irrespective of the request type, requires the operator to provision the datacenter conservatively based on the costliest request.

Pulsar addresses these challenges and provides the VDC abstraction. It responds to shifting bottlenecks through a logically centralized controller that periodically allocates resources to tenants based on their VDC specifications, demands, and appliance capacities. These allocations are enforced by rate enforcers, found at end-host hypervisors, through a novel multi-resource token bucket (§4.4). Since the actual cost of serving requests can vary, Pulsar charges requests using their *virtual cost*, given in *tokens* (§3). This is a unified metric common to all VDC resources, and hence throughput in Pulsar is measured in tokens/sec. For each appliance, the provider specifies a *virtual cost function* that translates a request into its cost in tokens. This gives tenants a pre-advertised cost model, and allows the provider to offer guarantees that are independent of tenants' workloads without conservative provisioning.

Pulsar's implementation of the VDC abstraction allows the provider to express different resource allocation policies. The provider can offer VDCs with fixed or minimum guarantees. The former gives tenants predictable

performance, while the latter allows them to elastically obtain additional resources. The provider can then allocate spare resources to tenants based on policies that, for example, maximize profit instead of fairness. Additionally, tenants enjoy full control of their VDC resources and can specify their own allocation policies. For example, tenants can give some of their VMs preferential access to an appliance, or can divide their resources fairly.

The flexibility of these policies comes from decomposing the allocation of resources into two steps: (1) a per-tenant allocation step in which tenants receive enough resources to meet their VDC specifications, and (2) a global allocation step in which spare resources are given to tenants with minimum guarantees that have unmet demand (§4.1). For each step, tenants and the provider can choose from existing multi-resource allocation mechanisms [24, 30, 38, 48, 57] to meet a variety of goals (e.g., fairness, utilization, profit maximization).

Overall, this paper makes the following contributions:

- We propose the VDC abstraction, and present the design, implementation, and evaluation of Pulsar.

- We introduce a unified throughput metric based on *virtual request cost*. This makes it tractable for the provider to offer workload-independent guarantees.

- We design controller-based mechanisms to estimate the demand of tenants and the capacity of unmodified appliances for a given workload.

- We design a rate-limiter based on a new multi-resource token bucket that ensures tenants with changing workloads cannot affect other tenants' guarantees.

A key feature of Pulsar is its ease of deployment. Pulsar isolates tenants without requiring any modifications to applications, guest OSes, appliances, or the network. As a proof of concept, we deployed a prototype implementation of Pulsar on a small testbed comprising eleven servers, 113 VMs, and three types of appliances: an SSD-backed filestore, an in-memory key-value store, and an encryption appliance. We show that Pulsar is effective at enforcing tenant VDCs with data and control plane overheads that are under 2% (§6.3). We also find that controller scalability is reasonable: the controller can compute allocations for datacenters with 24K VMs and 200 appliances in 1–5 seconds (§6.4).

2 Motivation and background

Performance interference in shared datacenters is well documented both in the context of the network [33, 46, 55, 70, 71], and of shared appliances like storage servers (filestores, block stores, and key-value stores) [25, 31, 46], load balancers [49], IDSes [20], and software routers [23]. These observations have led to propos-

als that provide performance isolation across the network [12, 14, 28, 43, 51, 52, 60, 73], storage servers [26, 62, 66, 72], and middleboxes [23]. However, in all cases the focus is either on a single resource (network or storage), or on multiple resources within a single appliance.

By contrast, today's cloud platforms offer a diverse selection of appliances that tenants can use to compose their applications. Measurements from Azure's datacenters show that up to 44% of their intra-datacenter traffic occurs between VMs and appliances [49]. In this section, we show that tenants can be bottlenecked at any of the appliances or network resources they use, that these bottlenecks can vary over time as tenants' workloads change, and that the end result is variable application performance. Furthermore, we show that existing mechanisms cannot address these challenges.

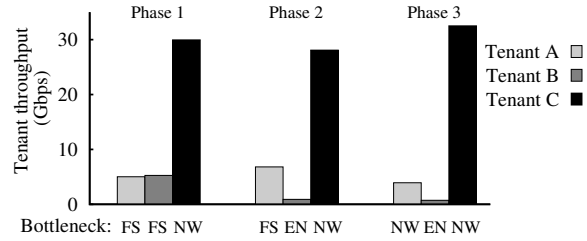### 2.1 How do tenant bottlenecks change?

We begin with a simple experiment on our testbed (detailed in Section 6) comprising 16-core servers connected using RDMA over converged Ethernet (RoCE) on a 40 Gbps network. The setup, depicted in Figure 1(a), involves three physical servers and two appliances: a filestore with an SSD back-end and an encryption appliance. The filestore is a centralized appliance providing persistent storage for all the VMs, while the encryption appliance is a distributed appliance present inside the hypervisor at each server. There are three tenants, *A–C*, running synthetic workloads on six, six, and twelve VMs respectively. Tenant *A* is reading from the filestore and tenant *B* is writing to the filestore, resulting in 64 KB IO requests across the network. Tenant *C* is running an application that generates an all-to-one workload between its VMs. This models the "aggregate" step of *partition/aggregate* workloads, a common pattern for web applications [10].

We focus on tenant performance across three phases of the experiment—phase transitions correspond to one of the tenants changing its workload. The first set of bars in Figure 1(b) shows the aggregate throughput for the three tenants in phase 1. Tenants *A* and *B* are bottlenecked at the filestore. Having similar workloads, they share the SSD throughput and achieve 5.2 Gbps each. Tenant *C*, with its all-to-one traffic, is bottlenecked at the network link of the destination VM and achieves 29.9 Gbps.

In the next phase, tenant *B*'s traffic is sent through the encryption appliance (running AES). This may be requested by the tenant or could be done to accommodate the provider's security policy. Tenant *B*'s throughput is thus limited by the encryption appliance's internal bottleneck resource: the CPU. As depicted in phase 2 of Figure 1(b), this decreases tenant *B*'s performance by $7\times$, and has a cascading effect. Since more of the filestore capacity is available to tenant *A*, its performance improves by 36%, thereby reducing tenant *C*'s throughput by 9.2%
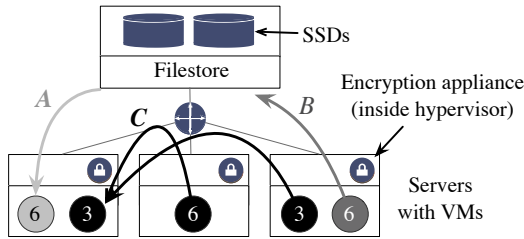
(a) Experiment setup with three tenants.

(b) Tenant performance varies as bottlenecks change.

Figure 1—Tenant performance is highly variable and depends on the appliances used and the workloads of other tenants. Numbers in (a) represent the # of VMs for a tenant; arrows represent the direction of traffic. The x-axis labels in (b) indicate the bottleneck appliance: "FS" is filestore, "EN" is encryption appliance, and "NW" is network.

	50th	90th	95th	99th	Duration (mins)
Key-value IO	0.04	0.14	0.28	0.41	2
Filestore IO	0.14	0.24	0.32	0.87	2 – 23
Network	0.002	0.004	0.005	0.61	1.3 – 49

Figure 2—Throughput at selected percentiles normalized based on the maximum value in each trace. Large differences between the median and higher percentiles indicate workload changes. The last column shows the duration of these changes.
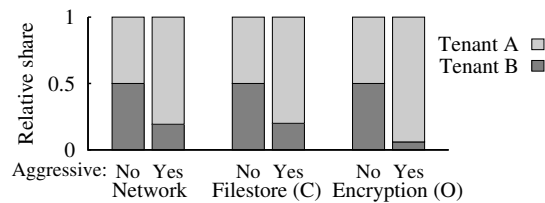

Figure 3—Tenant A can acquire a greater share of any appliance or the network by being aggressive. "(C)" is closed-loop workload, "(O)" is open-loop workload.

(since both tenants are colocated on the same server and share the network link).

In phase 3, tenant C generates more network flows from each of the source VMs to the destination VM. Since most TCP-like transport protocols achieve per-flow fairness, this allows tenant C to grab more of the bandwidth at the destination's network link and its throughput improves. However, this degrades the performance of tenant A's colocated VMs by 2.1×. These VMs are unable to saturate the filestore throughput and are instead bottlenecked at the network.

Overall, these simple yet representative experiments bring out two key takeaways:

- *Variable application performance.* A tenant's performance can vary significantly depending on its workload, the appliances it is using, and the workloads of other tenants sharing these appliances.

- *Multiple bottlenecks.* The performance bottleneck for tenants can vary across space and time. At any instant, tenants can be bottlenecked at different resources across different appliances. Over time, these bottlenecks can vary (as shown by the x-label in Fig. 1(b)).

The observations above are predicated on the prevalence of workload changes. We thus study tenant workloads in the context of two production datacenters next.

2.2 How common are workload changes?

We investigate workload changes by examining two traffic traces: (i) a week-long network trace from an enterprise datacenter with 300 servers running over a hundred applications, (ii) a two-day I/O trace from a Hotmail datacenter [67] running several services, including

a key-value store and a filestore. Figure 2 tabulates the percentiles of the per-second traffic, normalized to the maximum observed value in each trace. The big difference (orders of magnitude for the key-value and network traces) between the median and higher percentiles indicates a skewed distribution and changing workloads. To study the duration of workload changes, we identified the time intervals where the observed traffic is higher than the 95th percentile. The last column of Figure 2 shows that these workload changes can vary from minutes to almost an hour; such changes are common in both traces.

2.3 Why is tenant performance affected?

The root cause for variable tenant performance is that neither the network nor appliances isolate tenants from each other. Tenants can even change their workload to improve their performance at others' expense. We expose this behavior through experiments involving two tenants with six VMs each; the results are depicted in Figure 3.

In the first scenario, both tenants generate the same number of TCP flows through a network link, and hence, share it equally. However, tenant A can grab a greater share of the link bandwidth simply by generating more flows. For instance, the first set of bars in Figure 3 shows that tenant A can acquire 80% of the link bandwidth by generating four times more flows than tenant B.

Similar effects can be observed across appliances, but their relative performance depends on the nature of their workload. With closed-loop workloads, each tenant maintains a fixed number of outstanding requests against the appliance. Any tenant can improve its performance
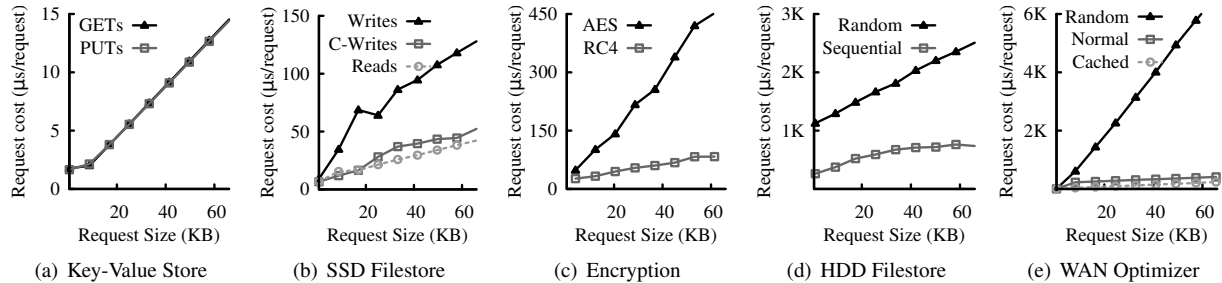
Figure 4—For appliances, the cost of serving a request can vary with request characteristics. "C-Writes" in (b) represents WRITE requests that can be compressed by the filestore's SSDs. "Sequential" and "Random" in (d) refers to the workload's access pattern (for both READs and WRITEs). (e) depicts the costs of a WAN Optimizer that performs compression for compressible ("Normal") and incompressible ("Random") requests; the cost of serving requests that are cached is also depicted.

by being aggressive and increasing the number of requests outstanding. For example, the second set of bars in Figure 3 shows the relative throughput of two tenants accessing a filestore. When both tenants have the same number of outstanding requests, they share the appliance equally. However, tenant *A* can acquire 80% of the filestore throughput simply by having four times as many requests outstanding as tenant *B*. In the case of open-loop workloads, there is no limit on the number of outstanding requests; in the absence of any isolation at the appliance, a tenant's share is dictated by the transport protocol used. The last set of bars in Figure 3 exposes this behavior.

2.4 Why are absolute guarantees hard to provide?

Isolating appliances is challenging because the resources consumed can vary substantially across individual requests. Figure 4 depicts this observation for five appliances: an in-memory key-value store, an SSD-backed and an HDD-backed filestore, an encryption appliance, and a WAN optimizer that performs compression [8]. For each appliance, we measured its throughput for a stream of requests with identical characteristics. We use the average time for serving a request as an approximation of the *actual request cost*. For the encryption appliance (Fig. 4(c)), the request cost depends on the encryption algorithm being used. For the HDD-backed filestore (Fig. 4(d)), the request cost depends not only on the request size, but also on the access pattern (sequential or random). A request's cost also depends on the appliance internals (including optimizations like caching). For example, Figure 4(b) shows that WRITE requests that can be compressed by the filestore SSDs ("C-Writes") are cheaper to serve than an incompressible write workload.

Another source of variability is the interference between tenant workloads. This exacerbates the difficulty of quantifying a request's cost as a function of its characteristics. The combinatorial explosion resulting from considering all possible workload combinations and the diversity of appliances makes this problem intractable.

Variable request cost has two implications for perfor-

mance isolation. First, while tenants should ideally receive guarantees in request/sec (or bits/sec) across an appliance, offering such guarantees regardless of tenants' workloads is too restrictive for the provider. Offering guaranteed requests/sec requires provisioning to support tenants always issuing the most expensive request (e.g., maximum-size WRITEs at a filestore appliance). Similarly, offering guaranteed bits/sec requires provisioning based on the request with the maximum cost-to-size ratio. Moreover, both cases require the provider to quantify the actual request cost which, as we discussed, is hard.

The second implication is that the *capacity*, or maximum aggregate throughput, of an appliance can vary over time and across workloads. This is problematic because sharing an appliance in accordance to tenants' guarantees—while ensuring that it is not underutilized—requires a priori knowledge of its capacity.

2.5 Why are existing solutions insufficient?

Existing systems focus on either network or appliance isolation. In Section 6.1, we show that, independently, these solutions do not guarantee end-to-end throughput. This raises a natural question: *is a naive composition of these systems sufficient to provide end-to-end guarantees?* The answer, as we explain below, is no.[1]

Consider a two-tenant scenario in which both tenants are guaranteed half of the available resources. Tenants *A* and *B* each have a single VM sharing a network link and a key-value store appliance (KVS). Tenant *A* issues PUTs and tenant *B* issues GETs to the KVS. On the network, GETs are very small as they contain only the request header; PUTs contain the actual payload. This means that isolating requests based on network semantics (i.e., message size) would allow many more GETs than PUTs to be sent to the KVS. This is problematic because processing GETs at the KVS consumes as many resources as processing PUTs (Fig 4(a)). Even if the KVS optimally

---

[1] A very similar proposition is discussed as a strawman design in DRFQ [23, §4.2], where each resource within a middlebox is managed by an independent scheduler.

schedules the arriving requests, the task is moot: the scheduler can only choose from the set of requests that actually reaches the KVS. Effectively, tenant *B*'s GETs crowd out tenant *A*'s PUTs, leading to tenant *B* dominating the KVS bandwidth—an undesired outcome!

The key takeaway from this example is that naively composing existing systems is inadequate because their mechanisms are decoupled: they operate independently, lack common request semantics, and have no means to propagate feedback. While it may be possible to achieve end-to-end isolation by bridging network and per-appliance isolation, such a solution would require complex coordination and appliance modifications.

3 Virtual datacenters

We propose virtual datacenters (VDCs) as an abstraction that encapsulates the performance guarantees given to tenants. The abstraction presents to tenants dedicated virtual appliances connected to their VMs through a virtual network switch. Each appliance and VM link is associated with a throughput guarantee that can be either *fixed* or *minimum*. Tenants with fixed guarantees receive the resources specified in their VDCs and no more. Tenants with minimum guarantees forgo total predictability but retain resource elasticity; they may be given resources that exceed their guarantees (when they can use them). These tenants can also specify maximum throughput guarantees to bound their performance variability.

Figure 5 depicts a sample VDC containing two virtual appliances (a filestore and an encryption service), *N* virtual machines, and the connecting virtual network. The guarantees for the filestore and the encryption service are G_s and G_E, respectively. VMs links' guarantees are also depicted. These guarantees are *aggregate*: even if only one or all *N* VMs are accessing the virtual filestore at a given time, the tenant is still guaranteed G_S across it.

For a tenant, the process of specifying a VDC is analogous to that of procuring a small dedicated cluster: it requires specifying the number of VMs, and the type, number, and guarantees of the virtual appliances and virtual network links. Note that VM provisioning (RAM, # cores, etc.) remains unchanged—we rely on existing hypervisors to isolate end-host resources [4, 7, 29, 68]. Providers can offer tenants tools like Cicada [42] and Bazaar [35] to automatically determine the guarantees they need, or tenants can match an existing private datacenter. Alternatively, providers may offer templates for VDCs with different resources and prices, as they do today for VMs (e.g., small, medium, etc.).

Virtual request cost. In Section 2.4 we showed that the actual cost of serving a request at an appliance can vary significantly. We address this by charging requests based on their virtual cost in *tokens*. For each appliance
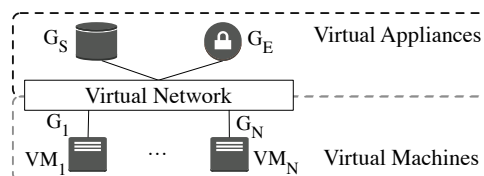

Figure 5—A VDC is composed of virtual appliances and VM links associated with throughput guarantees (in tokens/sec).

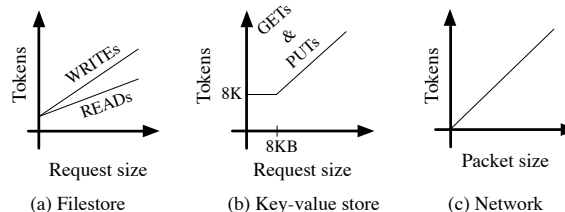(a) Filestore (b) Key-value store (c) Network

Figure 6—Sample virtual cost functions showing the mapping between request characteristics and their cost in tokens.

and the network, the provider advertises a *virtual cost function* that maps a request to its cost in tokens. Tenant guarantees across all appliances and the network are thus specified in *tokens/sec*, a unified throughput metric. This strikes a balance between the provider and tenants' requirements. The provider is able to offer workload-independent guarantees without conservative provisioning, while tenants can independently (and statically) determine the requests/second (and bits/second) throughput that can be expected from an appliance.

Figure 6 shows examples of virtual cost functions. The key-value store cost function states that any request smaller than 8 KB costs a flat 8K tokens, while the cost for larger requests increases linearly with request size. Consider a tenant with a guarantee of 16K tokens/sec. The cost function implies that if the tenant's workload comprises 4 KB PUTs, it is guaranteed 2 PUTs/s, and if it comprises 16 KB PUTs, it is guaranteed 1 PUTs/s. For the network, the relation between packet size and tokens is linear; tokens are equivalent to bytes. Cloud providers already implicitly use such functions: Amazon charges tenants for DyanamoDB key-value store requests in integral multiples of 1 KB [2]. This essentially models a virtual cost function with a step-wise linear shape.

The provider needs to determine the virtual cost function for each appliance. This typically involves approximating the actual cost of serving requests through benchmarking, based on historical statistics, or even domain expertise. However, cost functions need not be exact (they can even be deliberately different); our design accounts for any mismatch between the virtual and actual request cost, and ensures full appliance utilization (§4.3). It is thus sufficient for the provider to roughly approximate a request's cost from its observable characteristics. Section 8 discusses appliances for which observable request characteristics are a poor indicator of request cost.

4 Design

Pulsar enables the VDC abstraction by mapping tenant VDCs onto the underlying physical infrastructure and isolating them from each other. Pulsar's architecture, depicted in Figure 7, consists of a logically centralized controller with full visibility of the datacenter topology and tenants' VDC specifications, and a *rate enforcer* inside the hypervisor at each compute server. The controller estimates tenants' demands, appliance capacities, and computes allocations that are sent to rate enforcers. The rate enforcers collect local VM traffic statistics, and enforce tenant allocations.

Pulsar's design does not require the modification of physical appliances, guest OSes, or network elements, which eases the path to deployment. It also reconciles the isolation requirements of tenants with the provider's goal of high utilization by allocating spare capacity to tenants that can use it. Specifically, Pulsar's allocation mechanism achieves the following goals:

G1 *VDC-compliance.* Tenants receive an allocation of resources that meets the guarantees specified in their VDCs. A tenant can choose from different mechanisms to distribute these resources among its VMs.

G2 *VDC-elasticity.* Tenants receive allocations that do not exceed their demands (i.e., the resources they can actually consume). Moreover, spare resources are allocated to tenants with minimum guarantees and unmet demand in accordance to the provider's policy.

4.1 Allocating resources to tenants

We treat each appliance as an atomic black box and do not account for resources inside of it. For example, a key-value store includes internal resources like its CPU and memory, but we treat all of them as a single resource. Henceforth, a "resource" is either a network link or an appliance.[2] Each resource is associated with both a capacity that can vary over time and must be dynamically estimated, and a cost function that maps a request's characteristics into the cost (in tokens) of servicing that request. All of Pulsar's mechanisms act directly on tenant *flows*. A flow encapsulates all connections between a pair of VMs that share the same path (defined in terms of the physical resources used). Note that a flow can have the same source and destination VM, as is the case with flows that access end-devices like storage servers.

Allocations in Pulsar are performed in control intervals (e.g., 1 sec), and involve the controller assigning *allocation vectors* to flows. Each entry in an allocation vector describes the amount of a particular resource that a flow can use over the control interval. A flow's allocation is the sum of two components. First, a *local* com-
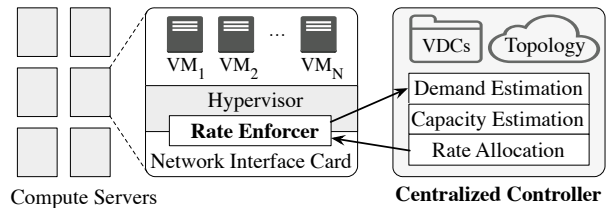

Figure 7—Pulsar's architecture is made up of a centralized controller that apportions resources to tenants, and distributed rate enforcers that uphold these allocations.

ponent is computed by applying a local policy (chosen by the tenant from a pre-advertised set) to the tenant's VDC. Next, a *global* component is computed by applying a global policy (chosen by the provider) to the physical infrastructure. The local policy describes how a tenant distributes its guaranteed resources to its flows, while the global policy describes how the provider distributes spare resources in the datacenter to flows with unmet demand. We describe multi-resource allocation next, followed by a description of the local and global allocations.

Multi-resource allocation (MRA). The goal of an MRA scheme is to distribute multiple *types* of resources among clients with heterogeneous demands. MRA schemes have been around for decades, primarily in the context of multi-capacity (or multi-dimension) bin packing problems [41, 45, 47, 54]. However, recent work [16, 19, 24, 30, 38, 40, 48, 57] has extended MRA schemes to ensure that the resulting allocations are not only efficient, but also meet different notions of fairness.

Generally, an MRA mechanism for m clients (flows in our context) and n resources provides the interface:

$$A \leftarrow MRA(D, W, C) \qquad (1)$$

where A, D, and W are $m \times n$ matrices, and C is an n-entry vector. $D_{i,j}$ represents the demand of flow i for resource j, or how much of resource j flow i is capable of consuming in a control interval. $A_{i,j}$ contains the resulting demand-aware allocation (i.e., $A_{i,j} \leq D_{i,j}$ for all i and j). W contains weight entries used to bias allocations to achieve a chosen objective (e.g., weighted fairness, or revenue maximization). C contains the capacity of each resource. With Pulsar, we can plug in any mechanism that implements the interface above for either allocation step.

Local allocations. Pulsar gives each tenant a private VDC. To give tenants control over how their guaranteed resources are assigned to their flows, we allow them to choose a local MRA mechanism (MRA_L). For example, tenants who want to divide their VDC resources fairly across their flows could choose a mechanism that achieves dominant-resource fairness (DRF) [24] or bottleneck-based fairness [19]. Alternatively, tenants may prefer a different point in the fairness-efficiency space, as achieved by other mechanisms [38, 57]. Hence,

tenant t's local allocation matrix (A^t) is given by:

$$A^t \leftarrow MRA_L(D^t, W^t, C^t) \qquad (2)$$

D^t and W^t are demand and weight matrices containing only t's flows, and C^t is the capacity vector containing the capacities of each virtual resource in t's VDC. These capacities correspond to the tenant's guarantees, which are static and known a priori (§3). W^t is set to a default (all entries are 1) but can be overridden by the tenant. We describe how flow demands are estimated in Section 4.2.

Global allocations. To achieve VDC-elasticity, Pulsar assigns unused resources to flows with unmet demand based on the provider's global policy.[3] This policy need not be fair or efficient. For example, the provider can choose a global allocation mechanism, MRA_G, that maximizes its revenue by favoring tenants willing to pay more for spare capacity, or prioritizing the allocation of resources that yield a higher profit (even if these allocations are not optimal in terms of fairness or utilization).

The resulting global allocation is the $m \times n$ matrix A^G, where m is the total number of flows (across all tenants with minimum guarantees), and n is the total number of resources in the datacenter. A^G is given by:

$$A^G \leftarrow MRA_G(D^G, W^G, C^G) \qquad (3)$$

D^G contains the unmet demand for each flow across each physical resource *after* running the local allocation step; entries for resources that are not in a flow's path are set to 0. The weights in W^G are chosen by the provider, and can be derived from tenants' VDCs to allow spare resources to be shared in proportion to up-front payment (a weighted fair allocation), or set to 1 to allow a fair (payment-agnostic) allocation. The n-entry capacity vector C^G contains the remaining capacity of every physical resource in the datacenter. Since tenants' demands vary over time, we describe how we estimate them next.

4.2 Estimating a flow's demand

The first input to Pulsar's MRA mechanisms is the demand matrix D. A row in D represents the demand vector for a flow, which in turn, contains the demand (in tokens) for each resource along the flow's path. The controller computes each flow's demand vector from estimates provided by rate enforcers. At a high level, the rate enforcer at a flow's source uses old and current request statistics to estimate a flow's demand for the next interval.

A flow's demand is the amount of resources that the application sourcing the flow *could* consume during a control interval, and it depends on whether the application is open- or closed-loop. Open-loop applications have no limit on the number of outstanding requests; the arrival rate is based on external factors like user input or timers. Consequently, a rate enforcer can observe a flow

f's demand for the current control interval by tracking both processed and queued requests.

The two components used to estimate the demand for flows of open-loop applications are the *utilization vector* and the *backlog vector*. Flow f's utilization vector, $u_f[i]$, contains the total number of tokens consumed for each resource by f's requests over interval i.[4] Note that if f's requests arrive at a rate exceeding its allocation, some requests will be queued (§4.4). f's backlog vector, $b_f[i]$, contains the tokens needed across each resource in order to process all the requests that are still queued at the end of the interval. Put together, the demand vector for flow f for the next interval, $d_f[i+1]$, is simply the sum of the utilization and backlog vector for the current interval:

$$d_f[i+1] = u_f[i] + b_f[i] \qquad (4)$$

Estimating the demand for flows of a closed-loop application is more challenging. These flows maintain a fixed number of outstanding requests which limits the usefulness of the backlog vector (since queuing at any point in time cannot exceed the number of outstanding requests). To address this, we account for queuing that occurs *throughout* a control interval and not just at the end of it. Within each control interval, we obtain periodic samples for the number of requests that are queued above and are outstanding beyond the rate enforcer; a flow's *queuing* (q_f) and *outstanding* (o_f) vectors contain the average number of requests (in tokens) that are queued and outstanding during a control interval. The demand vector for closed-loop flows at interval $i+1$ is thus given by:

$$d_f[i+1] = u_f[i] + q_f[i] \cdot \frac{u_f[i]}{o_f[i]} \qquad (5)$$

where "·" and "/" are element-wise operations. The rationale behind the second component is that an average of $o_f[i]$ outstanding tokens results in a utilization of $u_f[i]$. Consequently, if the rate enforcer were to immediately release all queued requests (which on average account for $q_f[i]$ tokens), the maximum expected additional utilization would be: $q_f[i] \cdot (u_f[i]/o_f[i])$.

In practice, however, it is difficult to differentiate between open- and closed-loop workloads. To reduce the probability that our mechanism under-estimates flow demands (which can result in violation of tenants' VDCs), we use the maximum of both equations:

$$d_f[i+1] = u_f[i] + \max\left(b_f[i], \; q_f[i] \cdot \frac{u_f[i]}{o_f[i]}\right) \qquad (6)$$

During every control interval, rate enforcers compute and send demand vectors for their flows to the controller, allowing it to construct the per-tenant and the global demand matrices. To avoid over-reacting to bursty workloads, the controller smoothens these estimates through an exponentially weighted moving average.

---

[3]Tenants with fixed guarantees are excluded from global allocations.

[4]Rate enforcers derive tokens consumed by a flow's requests on resources along its path by applying the corresponding cost functions.

4.3 Estimating appliance capacity

Recall that appliance capacity (measured in tokens/second) is the last input to Pulsar's MRA mechanisms (§4.1). If an appliance's virtual cost function perfectly describes the actual cost of serving a request, the appliance capacity is independent of its workload; the capacity is actually constant. However, determining actual request cost is hard, and thus virtual cost functions are likely to be approximate. This means that the capacity of an appliance varies depending on the given workload and needs to be estimated dynamically.

For networks, congestion control protocols implicitly estimate link capacity and distribute it among flows. However, they conflate capacity estimation with resource allocation which limits them to providing only flow-level notions of fairness, and their distributed nature increases complexity and hurts convergence time. Instead, Pulsar's controller serves as a natural coordination point, allowing us to design a centralized algorithm that estimates appliance capacity independently of resource allocation. This decoupling enables tenant-level allocations instead of being restricted to flow-level objectives. Furthermore, global visibility at the controller means that the mechanism is simple—it does not require appliance modification or inter-tenant coordination—yet it is accurate.

The basic idea is to dynamically adapt the *capacity estimate* for the appliance based on congestion signals. "Congestion" indicates that the capacity estimate exceeds the appliance's actual capacity and the appliance is overloaded. We considered both implicit congestion signals like packet loss and latency [22, 37, 74], and explicit signals like ECN [53] and QCN [32]. However, obtaining explicit signals requires burdensome appliance modifications, while implicit signals like packet loss are not universally supported (e.g., networked storage servers cannot drop requests [58]). Indeed, systems like PARDA [26] use latency as the sole signal for estimating system capacity. Nevertheless, latency is a noisy signal, especially when different flows have widely different paths.[5] Instead, we use the controller's global visibility to derive two implicit congestion signals: *appliance throughput* and *VDC-violation*.

Appliance throughput is the total throughput (in tokens) of all flows across the appliance over a given interval. When the capacity estimate exceeds the actual capacity, the appliance is overloaded and is unable to keep up with its workload. In this case, appliance throughput is less than the capacity estimate, signaling congestion.

The second congestion signal is needed because we aim to determine the appliance's *VDC-compliant capacity*—the highest capacity that meets tenants' VDCs. Since capacity is workload-dependent, and VDCs im-

pact the nature of the workload that reaches the appliance, the VDC-compliant capacity can be lower than the maximum capacity (across all workloads). To understand this, consider a hypothetical encryption appliance that serves either 4 RC4 requests, or 1 RC4 and 1 AES request per second (i.e., AES requests are $3\times$ more expensive than RC4 requests). Assume that the virtual cost function charges 2 tokens for an AES request and 1 token for an RC4 request, and that two tenants are accessing the appliance—t_{AES} with a minimum guarantee of 2 tokens/s, and t_{RC4} with a minimum guarantee of 1 token/s. The VDC-compliant workload in this scenario corresponds to 1 AES and 1 RC4 request every second, resulting in a VDC-compliant capacity of 3 tokens/s. However, notice that the maximum capacity is actually 4 tokens/s (when the workload is 4 RC4 requests). Assuming 1-second discrete timesteps and FIFO scheduling at the appliance, using a capacity of 4 tokens/s in Pulsar's global allocation step would result in t_{AES}'s guarantee being violated at least once every 3 seconds: Pulsar allows an additional RC4 request to go through, leading to uneven queuing at the appliance, and a workload that is not VDC-compliant. To avoid this, we use VDC-violation as a congestion signal.

Capacity estimation algorithm. We use a window-based approach for estimating appliance capacity. At a high level, the controller maintains a probing window in which the appliance's actual capacity (C_{REAL}) is expected to lie. The probing window is characterized by its extremes, *minW* and *maxW*, and is constantly refined in response to the presence or absence of congestion signals. The current capacity estimate (C_{EST}) is always within the probing window and is used by the controller for rate allocation. The refinement of the probing window comprises four phases:

① *Binary search increase.* In the absence of congestion, the controller increases the capacity estimate to the midpoint of the probing window. This binary search is analogous to BIC-TCP [74]. The controller also increases *minW* to the previous capacity estimate as a lack of congestion implies that the appliance's actual capacity exceeds the previous estimate. This process repeats until stability is reached or congestion is detected.

② *Revert.* When congestion is detected, the controller's response depends on the congestion signal. On observing the throughput congestion signal, the controller reverts the capacity estimate to *minW*. This ensures that the appliance does not receive an overloading workload for more than one control interval. Further, *maxW* is reduced to the previous capacity estimate since the appliance's actual capacity is less than this estimate.

③ *Wait.* On observing the VDC-violation signal, the controller goes through the revert phase onto the wait phase. The capacity estimate, set to *minW* in the revert

---

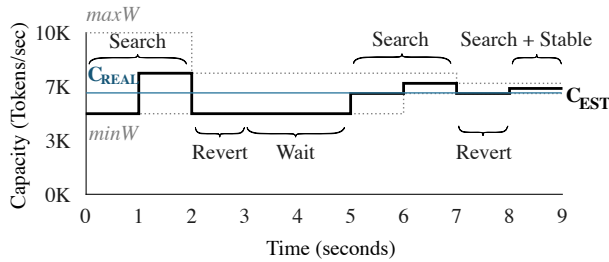[5]PARDA dampens noise through inter-client coordination.

Figure 8—Pulsar estimates an appliance's capacity by probing for a higher capacity and responding to congestion signals. Stability is reached once the probing window is small enough.

phase, is not changed until all guarantees are met again. This allows the appliance, which had been overloaded earlier, to serve all outstanding requests. This is particularly important as, unlike network switches, many appliances cannot drop requests.

④ *Stable.* Once the probing window is small enough (e.g., 1% of the maximum capacity), the controller reaches the stable state in which the capacity estimate is adjusted in response to minor fluctuations in workload. Our mechanism relies on tracking the average number of outstanding requests (measured in tokens) at the appliance during interval i, $O[i]$,[6] and comparing its value to the average number of outstanding requests at the appliance at the beginning of the stable phase, O_s. The difference between these observations affects C_{EST} as follows:

$$C_{EST}[i+1] = C_{EST}[i] - \alpha \cdot (O[i] - O_s) \qquad (7)$$

where α governs the sensitivity to workload changes. The rationale is that O_s serves as a good predictor of the number of outstanding requests that can be handled by the appliance when it is the bottleneck resource. When the outstanding requests at interval i ($O[i]$) exceed this amount, the appliance is likely to be overloaded; the estimate is reduced to ensure that fewer requests are let through by the rate enforcers during the next interval. The opposite is also true. Furthermore, workloads reaching the appliance that differ significantly (more than 10%) from the workload at the beginning of the stable phase restart the estimation process.

Figure 8 depicts a sample run of our algorithm on an appliance with an actual capacity (C_{REAL}) of 6500 tokens/second. Both *minW* and *maxW* are initialized to conservative values known to be much lower/higher than C_{REAL}, while the current estimate (C_{EST}) is initialized to *minW*. The dotted lines represent *minW* and *maxW*, while the solid line represents C_{EST}. At time $t = 1$, there is no congestion signal, so the controller enters phase ①, resulting in C_{EST} being set to 7500 (an overestimate). During the next interval, the controller notices that the

---

[6]The average number of outstanding requests at an appliance, O, is derived by summing over all flows' outstanding vectors (§4.2) and retrieving the entry corresponding to the appliance.

total appliance throughput does not match C_{EST}, which triggers phase ②. The queues that built up at the appliance due to capacity overestimation remain past $t = 3$, causing VDCs to be violated and leading into phase ③. This lasts until time $t = 5$, at which point all remnant queues have been cleared and the controller is able to go back to phase ①. This process repeats until stability is reached at time $t = 9$.

4.4 Rate enforcement

Pulsar rate limits each flow via a rate enforcer found at the flow's source hypervisor. Existing single-resource isolation systems use token buckets [65, §5.4.2] to rate-limit flows. However, traditional token buckets are insufficient to enforce multi-resource allocations, as tenants with changing workloads can consume more resources than they are allocated.

To understand why, assume a rate-limiter based on a single-resource token bucket where the bucket is filled with tokens from the first entry in a flow f's allocation vector (the same applies to any other entry). Further assume that f goes through 2 resources and its estimated demand vector at interval i is $\langle 800, 5000 \rangle$ (i.e., f is expected to use 800 tokens of resource 1 and 5,000 of resource 2). Suppose that the controller allocates to f all of its demand, and hence f's allocation vector is also $\langle 800, 5000 \rangle$. If f changes its workload and its actual demand is $\langle 800, 40000 \rangle$—e.g., a storage flow switching from issuing ten 500 B READs to ten 4 KB READs, where the request messages are the same size but the response message size increases—the rate-limiter would still let ten requests go through. This would allow f to consume $8\times$ its allocation on resource 2; an incorrect outcome!

To address this, we propose a *multi-resource token bucket* that associates multiple buckets with each flow, one for each resource in a flow's path. Each bucket is replenished at a rate given by the flow's allocation for the corresponding resource. For example, in the above scenario, a request is let through only if each bucket contains enough tokens to serve the request. Since f was allocated 5,000 tokens for resource 2, only one 4 KB READ is sent during interval i, and the remaining requests are queued until enough tokens are available. This mechanism ensures that even if a flow's workload changes, its throughput over the next control interval cannot exceed its allocation, and thus cannot negatively impact the performance of other flows or tenants.

4.5 Admission control

Pulsar's allocation assumes that tenants have been admitted into the datacenter, and their VDCs have been mapped onto the physical topology in a way that ensures that enough physical resources are available to meet their guarantees. This involves placing VMs on physical

servers and virtual appliances on their respective counterparts. While VM placement is well-studied [12, 14, 18, 44, 50, 73], prior proposals do not consider appliance placement. Our observation is that Hadrian's [14] placement algorithm can be adapted to support appliances.

Hadrian proposes a technique for modeling network bandwidth guarantees (among a tenant's VMs and across tenants) as a max-flow network problem [13, §4.1.1]. The result is a set of constraints that guide VM placement in the datacenter. Pulsar's VDCs can be similarly modeled. The key idea is to treat appliances as "tenants" with a single VM, and treat all VM-appliance interactions as communication between tenants. Consequently, we are able to model the guarantees in tenants' VDCs as a maximum-flow network, derive constraints for both VM and virtual appliance placement, and use Hadrian's greedy placement heuristic to map tenants' VDCs.

However, Hadrian's placement algorithm requires that the minimum capacity of each resource be known at admission time. While we assume that both the datacenter topology and link capacities are static and well known, determining the minimum capacity for each appliance is admittedly burdensome. Fortunately, Libra [61] proposes a methodology that, while tailored to SSDs, is general enough to cover numerous appliances. Furthermore, the work needed to derive appliances' minimum capacities can be used towards deriving cost functions as well.

5 Implementation

We implemented a Pulsar prototype comprising a standalone controller and a rate enforcer. The rate enforcer is implemented as a filter driver in Windows Hyper-V. There are two benefits from a hypervisor-based implementation. First, Pulsar can be used with unmodified applications and guest OSes. Second, the hypervisor contains the right semantic context for understanding characteristics of requests from VMs to appliances. Thus, the rate enforcer can inspect the header for each request to determine its cost. For example, for a request to a key-value appliance, the enforcer determines its type (GET/PUT) and its size in each direction (i.e., from the VM to the appliance and back). For encryption requests, it determines the request size and kind of encryption.

The driver implementing the rate enforcer is ≈11K lines of C; 3.1K for queuing and request classification, 6.8K for stat collection, support code, and controller communication, and 1.1K for multi-resource token buckets. The rate enforcer communicates with the controller through a user-level proxy that uses TCP-based RPCs; it provides demand estimates to the controller, and receives information about flows' paths, cost functions, and allocations. Each flow is associated with a multi-resource token bucket. The size of each bucket is set to a default of

token rate × 1 second. A 10 ms timer refills tokens and determines the queuing and outstanding vectors (§4.2).

The controller is written in ≈6K lines of C# and runs on a separate server. Inputs to the controller include a topology map of the datacenter, appliances' cost functions, and tenants' VDC specifications. The control interval is configurable and is set to a default of 1 second. Our traces show that workload changes often last much longer (§2.1), so a 1 second control interval ensures good responsiveness and stresses scalability. The controller estimates appliance capacity as described in Section 4.3. To prevent reacting to spurious congestion signals that result from noisy measurements we require multiple consistent readings (3 in our experiments).

At the controller, we have implemented DRF [24], H-DRF [16], and a simple first-fit heuristic as the available MRA mechanisms. In our experiments, we use DRF for local allocations (all weights are set to 1), and H-DRF for global allocations (weights are derived from tenants' guarantees). When computing these allocations the controller sets aside a small amount of headroom (2–5%) across network links. This is used to allocate each VM a (small) default rate for new VM-to-VM flows, which enables these flows to ramp up while the controller is contacted. Note that a new TCP connection between VMs is not necessarily a new flow since all transport connections between a pair of VMs (or between a VM and an appliance) are considered as one flow (§4).

6 Experimental evaluation

To evaluate Pulsar we use a testbed deployment coupled with simulations. Our testbed consists of eleven servers, each with 16 Intel 2.4 GHz cores and 380 GB of RAM. Each server is connected to a Mellanox switch through a 40 Gbps RDMA-capable Mellanox NIC. At the link layer, we use RDMA over converged Ethernet (RoCE). The servers run Windows Server 2012 R2 with Hyper-V as the hypervisor and each can support up to 12 VMs. We use three appliances: (i) a filestore with 6 SSDs (Intel 520) as the back-end, (ii) an in-memory key-value store, and (iii) an encryption appliance inside the hypervisor at each server. Admission control and placement is done manually. Overall, our key findings are:

- Using trace-driven experiments, we show that Pulsar can enforce tenants' VDCs. By contrast, existing solutions do not ensure end-to-end isolation.

- Our capacity estimation algorithm is able to predict the capacity of appliances over varying workloads.

- We find that Pulsar imposes reasonable overheads at the data and control plane. Through simulations, we show that the controller can compute allocations of rich policies for up to 24K VMs within 1-5 seconds.
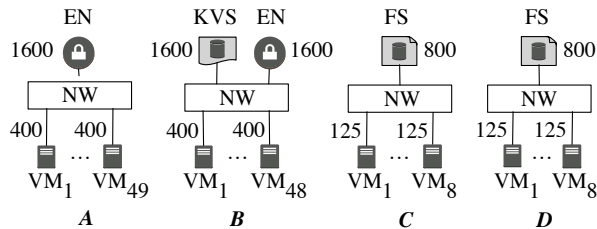
Figure 9—VDCs with minimum guarantees for tenants *A–D*

Tenant	READ or GET %	IO Size	Outstanding IOs #
Database IO (*B*)	61%	8 KB	8
Trans. Log (*C*)	1%	0.5 KB	64
Email (*D*)	56%	64 KB	8

Figure 10—Workload characteristics of each VM for tenants *B–D* are derived from a two-day Hotmail trace (§2.2).

6.1 Virtual datacenter enforcement

We first show that Pulsar enforces tenants' VDCs. The experiment involves four tenants, *A–D*, with their VDCs shown in Figure 9. For example, tenant *A* has forty-nine VMs, each with a minimum network bandwidth of 400 MT/s. For the network, tokens are equivalent to bytes, so these VMs have a guarantee of 400 MB/s (3.2 Gbps). This tenant also has a virtual encryption appliance with a minimum guarantee of 1600 MT/s. For ease of exposition we use the same cost function for all three appliances in our testbed: small requests ($\leq$8 KB) are charged 8 Ktokens, while the cost for other requests is the equivalent of their size in tokens (Figure 6(b) depicts the shape of this cost function).

Workloads. Tenant *A* has an all-to-one workload with its VMs sending traffic to one destination VM (this models a partition/aggregate workflow [10]). We use Iometer [5] parameterized by Hotmail IO traces (§2.2) to drive the workloads for tenants *B–D*; we tabulate their characteristics in Figure 10. Database IO is used for tenant *B*'s key-value store access, while Transactional log IO and Email message IO to Hotmail storage are used for *C* and *D* respectively. Traffic from tenants *A* and *B* is encrypted with RC4 by the encryption appliance before being sent on the wire. Since tenant *C* generates 512 byte requests that cost 8 Ktokens, its bytes/sec throughput is $\frac{1}{16}^{th}$ of the reported tokens/sec. The bytes/sec throughput for other tenants is the same as their tokens/sec.

Tenants *A* and *C* are aggressive: each of *A*'s VMs has 8 connections to the destination, while *C*'s generate a closed-loop workload with 64 outstanding IO requests.

Topology. Figure 11 shows the physical topology of the testbed. We arrange tenants' VMs and appliances across our servers so that at least two tenants compete for each resource. Tenants *A* and *B* compete for the encryption appliance. Tenants *C* and *D* compete for the bandwidth at the physical filestore appliance. Further, the destination
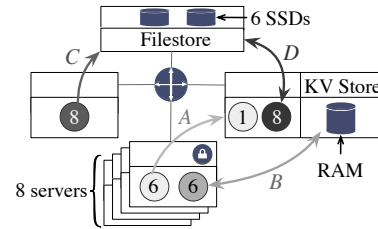

Figure 11—Testbed's physical topology. Numbers indicate # of VMs while arrows show the direction of traffic.
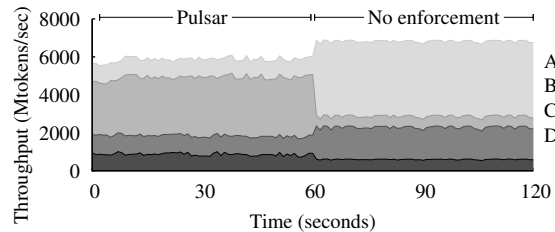

Figure 12—VM-to-VM and VM-to-appliance traffic of four tenants (113 VMs). Pulsar ensures all guarantees are met.

VM for tenant *A*, the key-value store used by tenant *B*, and all the VMs of tenant *D* are co-located on the same server. Thus, they compete at the server's network link.

Tenant guarantees. For the workload in this experiment, the end-to-end throughput for tenants *A-D* should be at least 400, 1600, 800, 800 MT/s respectively.

Tenant performance. The first half of Figure 12 shows that, with Pulsar, the aggregate throughput of each tenant exceeds its minimum guarantee. Further, spare capacity at each resource is shared in proportion to tenants' guarantees. On average, tenant *A* gets 1100 MT/s for its VM-to-VM traffic, tenant *B* gets 3150 MT/s for its key-value traffic, and tenants *C* and *D* get 930 and 900 MT/s across the filestore respectively.

By contrast, the second half of Figure 12 shows baseline tenant throughput without Pulsar. We find that the aggressive tenants (*A* and *C*) are able to dominate the throughput of the underlying resources at the expense of others. For instance, tenants *C* and *D* have the same guarantee to the filestore but *C*'s throughput is 3× that of *D*'s. Tenant *B*'s average throughput is just 580 MT/s, 64% lower than its guarantee. Similarly, tenant *D*'s average throughput is 575 MT/s, 28% lower than its guarantee.

In this experiment, the total throughput (across all tenants) with Pulsar is lower than without it by 8.7%. This is because Pulsar, by enforcing tenant guarantees, is effectively changing the workload being served by the datacenter. Depending on the scenario and the cost functions, such a workload change can cause the total throughput to either increase or decrease (with respect to the baseline).

We also experimented with prior solutions for single-resource isolation. DRFQ [23] achieves per-appliance isolation for general middleboxes, while Pisces [62] and
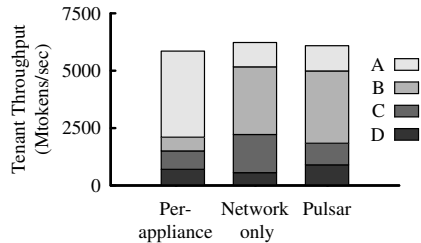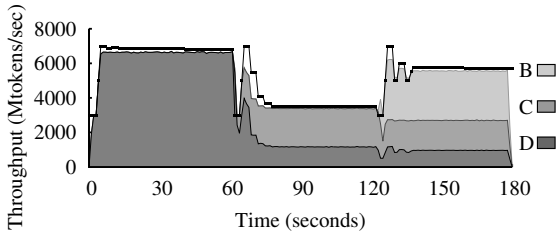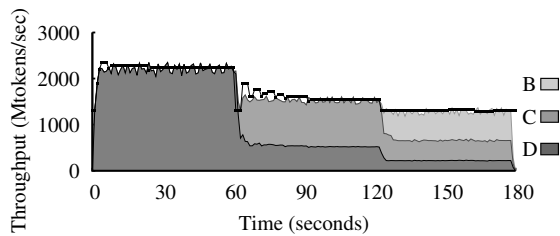
Figure 13—Prior isolation mechanisms fail to meet tenants' guarantees. Per-appliance isolation violates C and D's guarantee, while network only isolation violates D's guarantee.

(a) Key-value store

(b) Filestore

Figure 14—Pulsar's capacity estimation. The solid black line represents the estimated capacity. The guarantees of the three tenants have a ratio of 3:2:1 which is preserved throughout.

IOFlow [66] focus on storage appliances. With such per-appliance isolation, the filestore, key-value store, and encryption appliances are shared in proportion to tenant guarantees but not the network. Figure 13 shows that with per-appliance isolation, tenants B and D miss their guarantees by 63% and 12% respectively. Note that even though tenant D is bottlenecked at the filestore, it is sharing network links with tenant A whose aggressiveness hurts D's performance. We also compare against network-only isolation, as achieved by Hadrian [14]. Figure 13 shows that with this approach, tenant C is still able to hog the filestore bandwidth at the expense of tenant D, resulting in D's guarantee being violated by 30%.

6.2 Capacity estimation

We evaluate Pulsar's capacity estimation algorithm with an experiment that involves three tenants, B–D, whose workloads are tabulated in Figure 10. Unlike the previous experiments, we focus on one appliance at a time, and change the setup so that all three tenants use the ap-
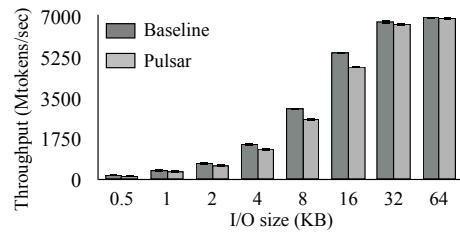

Figure 15—Baseline (no rate limiting) and Pulsar's throughput

pliance being evaluated. The appliance guarantees for the tenants are 600, 400, and 200 MT/s, respectively.

To experiment with varying workloads, we activate tenants one at a time. Figure 14(a) shows the estimated capacity and tenant throughput for the key-value store appliance. In the first phase, tenant D operates in isolation. The capacity estimate starts at a low value (3000 MT/s) and increases through the binary search phase until the appliance is fully utilized. After 8 seconds, the capacity estimate stabilizes at 6840 MT/s. Tenant C is activated next. Its VMs generate small 512 B requests that are more expensive for the key-value store to serve than they are charged, so the appliance's capacity reduces. The controller detects this workload change and the capacity estimate is reduced until it stabilizes at 3485 MT/s. Finally, when tenant B is activated, the appliance's actual capacity increases as the fraction of small requests (from C's VMs) reduces. The controller searches for the increased capacity and the estimate stabilizes at 5737 MT/s. Note that the guarantees of all three tenants are met throughout. Using H-DRF as the MRA_G mechanism ensures that spare resources are given based on tenants' guarantees, preserving the 3:2:1 ratio. In all three phases, the estimate converges within 15 seconds.

Figure 14(b) shows capacity estimation for the filestore. As tenants are added, their workloads increase the percentage of small WRITEs, leading to a decrease in the appliance's capacity. The root cause for the lower capacity is that the cost function that we chose undercharges all small requests and incorrectly charges WRITEs the same as READs (cf. Fig 4(b)). To account for this mismatch, the capacity estimate is consistently refined and converges to a value that ensures the appliance is neither being underutilized nor are tenants' guarantees being violated. We validate our observations by re-running the experiments with more accurate cost functions. The result is a capacity estimate that remains constant despite workload changes. We also experimented with the encryption appliance and the HDD-filestore, and the estimation results are similar. We omit them for brevity.

6.3 Data- and control-plane overheads

We first quantify the data-plane overhead of our rate enforcer. We measure the throughput at an unmodified Hyper-V server and compare it to the throughput when
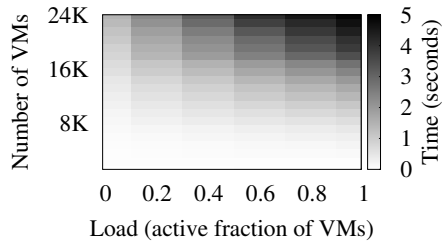
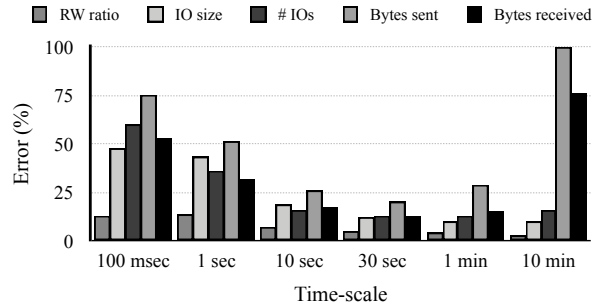Figure 16—Average time for computing rate allocations

Figure 17—The choice of control interval affects the accuracy of the utilization vector (§4.2) for estimating future demand. The error reduces when the control interval is ≈10–30 seconds.

our rate limiter is enabled. To show worst-case overheads we use the in-memory key-value store since that achieves the highest baseline throughput. 12 VMs are used to generate a workload with the same number of PUTs and GETs. We vary the request size from 512 B to 64 KB, thus shifting the bottleneck from the key-value store's CPU for small IO requests to the network for larger requests.

Figure 15 shows the average throughput from 5 runs. The worst-case reduction in throughput is 15% and happens for small request sizes (<32 KB). This overhead is due mostly to data structure locking at high 40+ Gbps speeds. The overhead for requests larger than 32 KB is less than 2%. The CPU overhead at the hypervisor was less than 2% in all cases.

In terms of control-plane overhead, the network cost of the controller updating rate allocations at the servers is 140 bytes/flow, while the cost of transmitting statistics to the controller is 256 bytes/flow per control interval. For example, for 10,000 flows this would mean 10.4 Mbps of traffic from the controller to the rate limiters and 19.52 Mbps of traffic to the controller. Both numbers are worst-case—if the rate allocation or the statistics collected by the rate enforcer do not change from one interval to the next, then no communication is necessary. The latency (including work done by both the controller and the hypervisor) for setting up a rate limiter for a given flow is approximately 83 μs. In general, these numbers indicate reasonable control plane overheads.

6.4 Controller scalability

We evaluate controller scalability through large-scale simulations. Flow demand estimation and appliance capacity estimation incur negligible costs for the controller. Local allocations are parallelizable, and involve much fewer flows and resources than global allocations. We thus focus on quantifying the cost of computing global allocations. We simulate a datacenter with a fat-tree topology [9] and 12 VMs per physical server. Tenants are modeled as a set of VMs with a VDC specification. Each VM sources one flow, either to another VM, or to an appliance. This results in 12 flows per server, which is twice the average number observed in practice [39, §4.1]. Based on recent datacenter measurements [49, Fig. 3], we configure 44% of flows to go to appliances while the

rest are VM-to-VM flows. For resources, we model each server's network link (uplink and downlink) and all physical appliances. Thus, for a datacenter with 2000 servers and 200 appliances, we model 4200 resources.

Figure 16 shows the average allocation time with our iterative DRF implementation as the global allocation mechanism; we vary the total number of VMs and the fraction of VMs that are active. We find that our controller can compute allocations for a medium-scale datacenter with 24K VMs and 2000 servers within 1–5 seconds. When only 20% of the VMs are active, allocation time is at or below 1.5 seconds, even with 24K VMs. However, high loads can push allocation time to as high as 4.9 seconds. The majority of the time is spent performing element-wise arithmetic operations on the 4200-entry vectors. This suggests that parallelizing these operations could provide meaningful performance benefits. We are currently experimenting with GPU implementations of different allocation mechanisms.

Repeating the experiment with H-DRF shows that costs are an order of magnitude higher. This highlights the tradeoff between a policy's expressiveness and its computational burden.

6.5 Choice of control interval

Resource allocation in Pulsar is demand-driven. Hence, the ideal control interval should capture the true demand of flows. Estimating demand for a very short future interval can be impacted by bursts in workload while estimating for a long interval may not be responsive enough (i.e., it may not capture actual workload changes). To verify this, we used our network and IO traces (§2.2) to estimate flow demand at various time-scales. Unfortunately, the traces do not have queuing and backlog information, so we cannot use Pulsar's demand estimation mechanism detailed in Section 4.2. Instead, we simply use past utilization as an indicator of future demand. Specifically, we approximate the demand for a time interval using an exponentially weighted moving average of the utilization over the previous intervals. Thus, the demand errors we report are an over-estimate.

For the network, we use two demand metrics: bytes sent and received. For the storage traffic, the metrics are the number and mean size of IOs, and the read-to-write IO ratio. Figure 17 shows the average demand estimation error across several time-scales. As is well-known [15, 39], most workloads exhibit bursty behavior at very fine timescales (below 1 sec); hence, using a very short control interval leads to large estimation errors. At large time scales (several minutes), past utilization is a poor predictor of future demand. For these workloads, a control interval of ≈10–30 seconds is best suited for demand estimation, offering a good trade-off between responsiveness and stability. While preliminary, these results indicate that Pulsar's controller-based architecture can cope with real datacenter workloads.

7 Related work

Section 2 briefly described existing work on inter-tenant performance isolation. Below we expand that description and contrast related work to Pulsar.

Appliance isolation. A large body of recent work focuses on storage isolation [17, 26, 27, 62, 66], but in all cases the network is assumed to be over-provisioned. While DRFQ [23] achieves fair sharing of multiple resources within a single appliance, it differs from Pulsar mechanistically and in scope: Pulsar decouples capacity estimation from resource allocation, and provides isolation across multiple appliances and the network. Furthermore, Pulsar provides workload-independent guarantees by leveraging an appliance-agnostic throughput metric.

Like Pisces [62] and IOFlow [66], Pulsar uses a centralized controller to offer per-tenant guarantees. However, Pisces relies on IO scheduling at the storage server, while Pulsar performs end-host enforcement without appliance modification. Moreover, Pulsar dynamically estimates the capacity of appliances, whereas IOFlow requires that they be known a priori.

Network isolation. Numerous systems isolate tenants across a shared datacenter network [11, 12, 14, 28, 36, 43, 51, 52, 60, 73]. Beyond weighted sharing [60] and fixed reservations [12, 28, 73], recent efforts ensure minimum network guarantees, both with switch modifications [11, 14, 51], and without them [36, 43, 52]. Pulsar extends the latter body of work by providing guarantees that span datacenter appliances and the network.

Market-based resource pricing. Many proposals allocate resources to bidding users based on per-resource market prices that are measured using a common virtual currency [21, 34, 63, 64, 69]. However, the value of a unit of virtual currency in terms of actual throughput (e.g., requests/sec) varies with supply and demand. Consequently, a tenant's throughput is not guaranteed. By contrast, Pulsar charges requests based on their vir-

tual cost (measured in tokens). While tokens can be seen as a virtual currency, the fact that each resource is associated with a pre-advertised virtual cost function means that a tenant's guarantees in tokens/sec can still be statically translated into guarantees in requests/sec.

Virtual Datacenters. The term VDC has been used as a synonym for Infrastructure-as-a-service offerings (i.e., VMs with CPU and memory guarantees [3, 6]). SecondNet [28] extended the term to include network address and performance isolation by associating VMs with private IPs and network throughput guarantees. Pulsar broadens the VDC definition to include appliances and ensures throughput guarantees across all resources.

8 Discussion and summary

Pulsar's design relies on cost functions that translate requests into their virtual cost. However, for some appliances, observable request characteristics (size, type, etc.) are not a good indicator of request cost. For example, quantifying the cost of a query to a SQL database requires understanding the structure of the query, the data being queried, and database internals. Similarly, the isolation of appliances that perform caching requires further work. While Pulsar implicitly accounts for caching through higher capacity estimates, it does not discriminate between requests that hit the cache and those that do not. We are experimenting with *stateful cost functions* that can charge requests based on past events (e.g., repeated requests within an interval cost less), to explicitly account for such appliances.

In summary, Pulsar gives tenants the abstraction of a virtual datacenter (VDC) that affords them the performance stability of a in-house cluster, and the convenience and elasticity of the shared cloud. It uses a centralized controller to enforce end-to-end throughput guarantees that span multiple appliances and the network. This design also allows for a simple capacity estimation mechanism that is both effective, and appliance-agnostic. Our prototype shows that Pulsar can enforce tenant VDCs with reasonable overheads, and allows providers to regain control over how their datacenter is utilized.

References

[1] Amazon AWS Products. `http://aws.amazon.com/products/`.

[2] Amazon DynamoDB capacity unit. `http://aws.amazon.com/dynamodb/faqs/#What_is_a_readwrite_capacity_unit`.

[3] Bluelock: Virtual data centers. `http://www.bluelock.com/virtual-datacenters`.

[4] Hyper-V resource allocation. `http://technet.microsoft.com/en-us/library/cc742470.aspx`.

[5] Iometer. `http://iometer.org`.

[6] VMware vCloud: Organization virtual data center. `http://kb.vmware.com/kb/1026320`.

[7] vSphere resource management guide. `https://www.vmware.com/support/pubs/vsphere-esxi-vcenter-server-pubs.html`.

[8] WANProxy. `http://wanproxy.org`.

[9] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM Conference*, Aug. 2008.

[10] M. Alizadeh, A. G. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM Conference*, Aug. 2010.

[11] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. The price is right: Towards location-independent costs in datacenters. In *Proceedings of the ACM Workshop on Hot Topics in Networks (HotNets)*, Nov. 2011.

[12] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards predictable datacenter networks. In *Proceedings of the ACM SIGCOMM Conference*, Aug. 2011.

[13] H. Ballani, D. Gunawardena, and T. Karagiannis. Network sharing in multi-tenant datacenters. Technical Report MSR-TR-2012-39, MSR, 2012.

[14] H. Ballani, K. Jang, T. Karagiannis, C. Kim, D. Gunawaradena, and G. O'Shea. Chatty tenants and the cloud network sharing problem. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Apr. 2013.

[15] T. Benson, A. Anand, A. Akella, and M. Zhang. Understanding data center traffic characteristics. In *Proceedings of the ACM SIGCOMM Workshop on Research on Enterprise Networking (WREN)*, Aug. 2009.

[16] A. Bhattacharya, D. Culler, E. Friedman, A. Ghodsi, S. Shenker, and I. Stoica. Hierarchical scheduling for diverse datacenter workloads. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC)*, Oct. 2013.

[17] J.-P. Billaud and A. Gulati. hClock: Hierarchical QoS for packet scheduling in a hypervisor. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, Apr. 2013.

[18] O. Biran, A. Corradi, M. Fanelli, L. Foschini, A. Nus, D. Raz, and E. Silvera. A stable network-aware VM placement for cloud systems. In *Proceedings of the IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, May 2012.

[19] D. Dolev, D. G. Feitelson, J. Y. Halpern, R. Kupferman, and N. Linial. No justified complaints: On fair sharing of multiple resources. In *Proceedings of the Innovations in Theoretical Computer Science (ITCS) Conference*, Aug. 2012.

[20] H. Dreger, A. Feldman, V. Paxson, and R. Sommer. Predicting the resource consumption of network intrusion detection systems. In *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, Sept. 2008.

[21] M. Feldman, K. Lai, and L. Zhang. A price-anticipating resource allocation mechanism for distributed shared clusters. In *Proceedings of the ACM Conference on Electronic Commerce (EC)*, June 2005.

[22] S. Floyd. HighSpeed TCP for large congestion windows, Dec. 2003. RFC 3649. `http://www.ietf.org/rfc/rfc3649.txt`.

[23] A. Ghodsi, V. Sekar, M. Zaharia, and I. Stoica. Multi-resource

[24] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Mar. 2011.

[25] D. Ghoshal, R. S. Canon, and L. Ramakrishnan. I/O performance of virtualized cloud environments. In *Proceedings of the International Workshop on Data Intensive Computing in the Clouds (DataCloud)*, May 2011.

[26] A. Gulati, I. Ahmad, and C. A. Waldspurger. PARDA: proportional allocation of resources for distributed storage access. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, Feb. 2009.

[27] A. Gulati, A. Merchant, and P. J. Varman. mClock: Handling throughput variability for hypervisor IO scheduling. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Oct. 2010.

[28] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang. SecondNet: A data center network virtualization architecture with bandwidth guarantees. In *Proceedings of the International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, Nov. 2010.

[29] D. Gupta, L. Cherkasova, R. Gardner, and A. Vahdat. Enforcing performance isolation across virtual machines in XEN. In *Proceedings of the ACM/IFIP/USENIX International Middleware Conference*, Dec. 2006.

[30] A. Gutman and N. Nisan. Fair allocation without trade. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, June 2012.

[31] Z. Hill, J. Li, M. Mao, A. Ruiz-Alvarez, and M. Humphrey. Early observations on the performance of Windows Azure. In *Proceedings of the International ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, June 2010.

[32] IEEE Computer Society. Virtual bridged local area networks. Amendment 13: Congestion notification, Apr. 2010. IEEE Std. 802.1Qau-2010.

[33] A. Iosup, N. Yigitbasi, and D. Epema. On the performance variability of cloud services. Technical Report PDS-2010-002, Delft University, 2010.

[34] D. Irwin, J. Chase, L. Grit, and A. Yumerefendi. Self-recharging virtual currency. In *Proceedings of the ACM SIGCOMM Workshop on the Economics of Peer-to-Peer Systems (P2PECON)*, Aug. 2005.

[35] V. Jalaparti, H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Bridging the tenant-provider gap in cloud services. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC)*, 2012.

[36] V. Jeyakumar, M. Alizadeh, D. Mazières, B. Prabhakar, C. Kim, and A. Greenberg. EyeQ: Practical network performance isolation at the edge. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Apr. 2013.

[37] C. Jin, D. X. Wei, and S. H. Low. FAST TCP: Motivation, architecture, algorithms, performance. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, Mar. 2004.

[38] C. Joe-Wong, S. Sen, T. Lan, and M. Chiang. Multi-resource allocation: Fairness-efficiency tradeoffs in a unifying framework. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, Apr. 2012.

[39] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. The nature of data center traffic: Measurements & analysis. In *Proceedings of the ACM SIGCOMM Conference on Internet Measurement (IMC)*, Nov. 2009.

[40] I. Kash, A. D. Procaccia, and N. Shah. No agent left behind: Dynamic fair division of multiple resources. In *Proceedings of the*

fair queueing for packet processing. In *Proceedings of the ACM SIGCOMM Conference*, Aug. 2012.

International Conference on Autonomous Agents and Multiagent Systems (AAMAS), May 2013.

[41] L. T. Kou and G. Markowsky. Multidimensional bin packing algorithms. *IBM Journal of Research and Development*, 21(5), Sept. 1977.

[42] K. LaCurts, J. C. Mogul, H. Balakrishnan, and Y. Turner. Cicada: Introducing predictive guarantees for cloud networks. In *Proceedings of the USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, June 2014.

[43] J. Lee, Y. Turner, M. Lee, L. Popa, S. Banarjee, J.-M. Kang, and P. Sharma. Application-driven bandwidth guarantees in datacenters. In *Proceedings of the ACM SIGCOMM Conference*, Aug. 2014.

[44] S. Lee, R. Panigraphy, V. Prabhakaran, V. Ramasubramanian, K. Talwar, L. Uyeda, and U. Wieder. Validating heuristics for virtual machines consolidation. Technical Report MSR-TR-2011-9, Microsoft Research, 2011.

[45] W. Leinberger, G. Karypis, and V. Kumar. Multi-capacity bin packing algorithms with applications to job scheduling under multiple constraints. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, Sept. 1999.

[46] A. Li, X. Yang, S. Kandula, and M. Zhang. CloudCmp: Comparing public cloud providers. In *Proceedings of the ACM SIGCOMM Conference on Internet Measurement (IMC)*, Nov. 2010.

[47] K. Maruyama, S. K. Chang, and D. T. Tang. A general packing algorithm for multidimensional resource requirements. *International Journal of Computer and Information Sciences*, 6(2), 1977.

[48] D. C. Parkes, A. D. Procaccia, and N. Shah. Beyond dominant resource fairness: Extensions, limitations, and indivisibilities. In *Proceedings of the ACM Conference on Electronic Commerce (EC)*, June 2012.

[49] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, C. Kim, and N. Karri. Ananta: Cloud scale load balancing. In *Proceedings of the ACM SIGCOMM Conference*, Aug. 2013.

[50] J. T. Piao and J. Yan. A network-aware virtual machine placement and migration approach in cloud computing. In *Proceedings of the International Conference on Grid and Cloud Computing (GCC)*, Nov. 2010.

[51] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica. FairCloud: Sharing the network in cloud computing. In *Proceedings of the ACM SIGCOMM Conference*, Aug. 2012.

[52] L. Popa, P. Yalagandula, S. Banarjee, J. C. Mogul, Y. Turner, and J. R. Santos. ElasticSwitch: Practical work-conserving bandwidth guarantees for cloud computing. In *Proceedings of the ACM SIGCOMM Conference*, Aug. 2013.

[53] K. Ramakrishnan, S. Floyd, and D. Black. The addition of explicit congestion notification (ECN) to IP, Sept. 2001. RFC 3168. `http://www.ietf.org/rfc/rfc3168.txt`.

[54] N. Roy, J. S. Kinnebrew, N. Shankaran, G. Biswas, and D. C. Schmidt. Toward effective multi-capacity resource allocation in distributed real-time embedded systems. In *Proceedings of the IEEE Symposium on Object/Component/Service-oriented Real-time Distributed Computing (ISORC)*, May 2008.

[55] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz. Runtime measurements in the cloud: observing, analyzing, and reducing variance. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, Sept. 2010.

[56] V. Sekar, S. Ratnasamy, M. K. Reiter, N. Egi, and G. Shi. The middlebox manifesto: Enabling innovation in middlebox deployment. In *Proceedings of the ACM Workshop on Hot Topics in Networks (HotNets)*, Nov. 2011.

[57] D. Shah and D. Wischik. Principles of resource allocation in networks. In *Proceedings of the ACM SIGCOMM Education Workshop*, May 2011.

[58] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. Network file system (NFS) version 4 protocol, Apr. 2003. RFC3530. `http://www.ietf.org/rfc/rfc3530.txt`.

[59] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making middleboxes someone else's problem: Network processing as a cloud service. In *Proceedings of the ACM SIGCOMM Conference*, Aug. 2012.

[60] A. Shieh, S. Kandula, A. Greenberg, C. Kim, and B. Saha. Sharing the data center network. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Mar. 2011.

[61] D. Shue and M. J. Freedman. From application requests to Virtual IOPs: Provisioned key-value storage with Libra. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, Apr. 2014.

[62] D. Shue, M. J. Freedman, and A. Shaikh. Performance isolation and fairness for multi-tenant cloud storage. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Oct. 2012.

[63] I. Stoica, H. Abdel-Wahab, and A. Pothen. A microeconomic scheduler for parallel computers. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, 1994.

[64] I. E. Sutherland. A futures market in computer time. *Communications of the ACM*, 11(6), June 1968.

[65] A. S. Tanenbaum and D. J. Wetherall. *Computer Networks*. Prentice Hall, 5th edition, Oct. 2010.

[66] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron, T. Talpey, R. Black, and T. Zhu. IOFlow: A software-defined storage architecture. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, Nov. 2013.

[67] E. Thereska, A. Donnelly, and D. Narayanan. Sierra: Practical power-proportionality for data center storage. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, Apr. 2011.

[68] C. A. Waldspurger. Memory resource management in VMware ESX server. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2002.

[69] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Nov. 1994.

[70] E. Walker. Benchmarking Amazon EC2 for high-performance scientific computing. *USENIX ;login:*, 33(5), 2008.

[71] G. Wang and T. S. E. Ng. The impact of virtualization on network performance of Amazon EC2 data center. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, Apr. 2010.

[72] H. Wang and P. J. Varman. Balancing fairness and efficiency in tiered storage systems with bottleneck-aware allocation. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, Feb. 2014.

[73] D. Xie, N. Ding, Y. C. Hu, and R. Kompella. The only constant is change: Incorporating time-varying network reservations in data centers. In *Proceedings of the ACM SIGCOMM Conference*, Aug. 2012.

[74] L. Xu, K. Harfoush, and I. Rhee. Binary increase congestion control (BIC) for fast long-distance networks. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, Mar. 2004.