

# Stratos: Virtual Middleboxes as First-Class Entities

Aaron Gember<sup>‡</sup> Robert Grandl<sup>‡</sup> Ashok Anand<sup>†\*</sup> Theophilus Benson<sup>‡</sup> Aditya Akella<sup>‡</sup>  
<sup>‡</sup>University of Wisconsin – Madison  
{agember,rgrandl,tbenson,akella}@cs.wisc.edu  
<sup>†</sup>Bell Labs, India  
ashok.anand@alcatel-lucent.com

## ABSTRACT

Enhancing application deployments in today’s clouds using virtual middleboxes is challenging due to the lack of network control and the inherent difficulty in intelligently scaling middleboxes to cope with application demands. Our position is that application and middlebox composition and scaling must occur in concert, and addressing the challenges that arise requires a framework that recognizes middleboxes as first-class entities. In this paper, we present the design, implementation, and evaluation of such a framework, called Stratos, that is targeted for immediate deployment over- or under-the-cloud today. In Stratos, tenants specify rich middlebox and application topologies using high-level abstractions, and desired middlebox traversals are realized using a novel distributed programmable data plane design. To avoid degradation in middlebox performance, middleboxes are automatically scaled using a greedy heuristic to meet application demands. Under-the-cloud deployments using network-aware flow distribution and placement provide further performance and scaling benefits. Our evaluation of an over-the-cloud deployment atop EC2 and simulation of an under-the-cloud deployment shows Stratos provides significant benefits to both tenants and providers.

## 1. INTRODUCTION

Cloud computing can enable enterprises to deploy applications flexibly and at scale while eliminating the costs and overhead of physical infrastructure management. For some applications migrating to a public or private cloud is a great choice. However, many such applications remain locked-in to existing data centers due to a mismatch between cloud capabilities and enterprise requirements.

One major roadblock is the lack of ground-up support for network-based services. Enterprises often rely on load balancers, intrusion detection and prevention systems, WAN optimizers, and a host of other network services appliances, or middleboxes, to ensure application security and improve performance. Private data centers enable the inclusion of dedicated appliances at network choke points to provide these services, albeit with significant configuration overhead. In contrast, today’s clouds offer little ability to enrich the network between servers with network-based services (§ 2).

At the basic level, cloud tenants desiring to include third-party services or middleboxes with their application must

retrofit middlebox functionality into generic virtual servers, realizing the necessary traffic redirection and splitting between virtual middleboxes (VMbs) using cumbersome configurations. Even if these issues were addressed, the performance offered by individual VMbs, and in aggregate by a VMb chain, may not meet the application requirements for two as-yet-unresolved reasons. First, tenants may have to re-provision (scale up or down) VMbs dynamically, to account for application elasticity and, potentially, to mesh with the “pay-as-you-go” model of cloud computing. Second, tenants may experience poor performance due to the provider’s placement of the tenant’s VMbs within the cloud data center, which may negate some of the benefits of deploying VMbs.

Our position is that addressing these issues in concert, and paving the way for rich application deployments in the cloud, requires a fresh view-point that *recognizes virtual middleboxes as first-class entities*. Middleboxes should be distinguished from generic compute resources and the appropriate support infrastructure needs to be developed, akin to cloud providers establishing storage systems as separate service offerings. Our view is not restricted to provider-offered services, e.g., Amazon Elastic Load Balancing [1], or to third-party-offered software, e.g., in the form of VM images; we believe that tenants should be able to mix and match both.

To realize this goal, we design Stratos. Stratos’s centralized management plane allows a cloud tenant to flexibly compose, manage or dynamically alter virtual topologies that contain arbitrary middleboxes. Stratos’s novel programmable data-plane then allows the framework provider (either a third-party vendor or cloud provider) to configure the desired functionalities for the tenant. Stratos incorporates a novel greedy heuristic to dynamically scale tenant topologies by deploying the appropriate number of VMb replicas that optimize the tenant’s performance at low cost. When application elasticity causes middleboxes to become bottlenecks, the heuristic identifies and removes bottlenecks by “scaling up”, based on a controlled exploration of the scaling space with decisions based solely on application reported performance; knowing the low-level details of the functions of individual middleboxes is not necessary. Similarly, when demand is low, the heuristic scales down.

The above make Stratos amenable to an over-the-cloud (OtC) deployment, where some of the key benefits of Stratos can be realized without the active participation of the cloud provider. We also present an under-the-cloud (UtC) version of Stratos, where the cloud provider implements specific placement and traffic distribution mechanisms to sup-

\*Ashok Anand completed this work while at the University of Wisconsin – Madison.

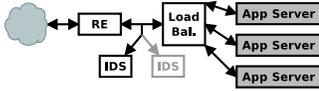


Figure 1: Example middlebox and server topology

port Stratos and further enhance its benefits. In doing so, the provider treats a tenant’s entire deployment chain as a single unit, as opposed to making decisions on a per-VM basis as is common today. We present a heuristic for smart “rack-aware” distribution of traffic among middlebox instances (subsets of which could be deployed in different racks). We also present a scheme for intelligent localized placement of a tenant’s initial topology and later middlebox instances added due to scaling. Both of these reduce inter-rack bandwidth demands reducing the likelihood of the network becoming a bottleneck. This improves overall tenant performance and makes the scaling heuristic more effective, and it also improves provider utilization.

We prototype Stratos atop Amazon’s EC2 cloud [1] to evaluate our OtC approach. We also build a simulator to study the UtC version. Our evaluations and simulations show:

- Arbitrarily rich tenant deployments can be realized using our distributed programmable data plane without performance degradation under modest loads.
- Our greedy scaling heuristic adequately eliminates middlebox bottlenecks providing suitable end-to-end performance in a 45% more cost-effective manner compared to naive scaling approaches.
- Network-aware flow distribution and middlebox instance placement in UtC deployments enables up to 30% more tenants to have their demands fully served, allows 45% of tenants to use the optimal number of middleboxes instances, and leaves up to an additional 20% of VM slots unused for providers to support future tenants.

## 2. MOTIVATION

Middleboxes play a key role in enterprises and private data centers [32] with an application server’s traffic often traversing multiple middlebox appliances to meet security, performance, or other objectives. As an example, traffic may enter the data center through a WAN optimizer or redundancy elimination middlebox (RE), be mirrored to an intrusion detection system (IDS), directed to a load balancer, and assigned to one of several application servers (Figure 1). With enterprises migrating their applications to the cloud, a wide-variety of cloud-provided appliances (e.g., Amazon’s Elastic Load Balancer [1]) or third-party VM images have emerged to supply the desired middlebox functionality. Despite this, we don’t see enterprises deploy rich applications with complex sets of VMb in the cloud (see [8] for anecdotes). Using examples of network services we tried to deploy on EC2, we elaborate on the major challenges involved.

**Composition:** Cloud providers don’t currently enable tenants to control network topology or routing between VMs [1, 9], and third-party network overlay services [12] only facilitate topologies containing directly addressed endpoints (in

contrast, on-path middleboxes are frequently transparent). As a result, tenants are forced to install VMb images on generic VMs and piece together tunnels, traffic splitters, and other software to forward the desired traffic along the desired paths. Finding the right third-party tools and correctly configuring these tools and the operating system is hard. For example, implementing the relatively simple set of middlebox traversals shown in Figure 1 required several days of trial-and-error to construct a working setup in Amazon EC2. The setup relies on several third-party tools and configurations strewn across the VMs.

Manual, distributed configuration makes managing the deployment (i.e., dynamically add or remove existing or new functionality) challenging. A change in middlebox traversal—to add new functionality or route around failed middleboxes—requires modifying configurations on several VMs. For example, adding a new replica of a TCP optimization middlebox between the WAN optimizer and load balancer requires configuration changes to the WAN optimizer(s), load balancer(s), and the new TCP optimizer. Distributed configurations also fail to provide administrators with a complete view of the implemented middlebox topology. This makes it difficult to verify that security policies and optimization objectives are correctly implemented: e.g., is a copy of all traffic being directed to one of the IDSs?

Recent proposals [16, 14] where providers enable tenants to specify topologies and control routing among their VM instances ease some of this pain. But, tenants need to specify *all* constituent VMs (e.g., all replicas of a middlebox), the detailed topology, and traffic splitting policies on a per-VM-pair basis, which can be complicated at scale. Also, they don’t accommodate off-path middleboxes (e.g., IDS). In all cases, identifying and responding to bottlenecks is hard.

**Chain Elasticity:** Imposing middlebox traversals on application traffic causes middlebox performance to effect end-to-end application performance. Even if application servers are capable of handling the current request load, an upstream middlebox may be a bottleneck. For example, consider a simple setup where a single client generates 300 web requests per second for a web page served by a single web server (see § 7 for details on this setup). Without any middleboxes between the client and server, the average request latency is 4ms. Forcing traffic to traverse two resource limited middleboxes—an intrusion prevention system and an RE system—increases this latency to 838ms, indicating one or both of the middleboxes is a bottleneck. In the cloud, where applications are frequently scaled based on load, being aware of and addressing such bottlenecks is essential to realizing the benefits of application elasticity.

Aside from a few cloud provided appliances (e.g., Amazon Elastic Load Balancer [1]), tenants are largely responsible for identifying and addressing middlebox bottlenecks themselves. Identifying *which* middlebox is bottlenecked is complicated. First, generic monitoring, e.g., cloud provided traffic counters, are insufficient for gauging the per-

formance of middleboxes which do not operate with *one packet in, one packet out* semantics, e.g., a WAN optimizer. Second, bandwidth between middlebox and/or application server VMs may be the bottleneck, as opposed to the capacity of the middlebox itself. The likelihood of encountering an oversubscribed link increases with the number of middleboxes traffic must traverse. More middleboxes also complicates the task of determining where the bottleneck resides.

**Provider support:** The extent to which the above two issues enable rich enterprise applications in the cloud depend on the level of support offered by the cloud provider. A key issue pertains to how the provider manages the resource footprint of a tenant’s deployment. Today, providers provision individual VMs, and don’t explicitly manage the compute and network resources consumed by the tenant’s network services deployment. While solutions to traversals and scaling can operate in such an environment, the benefits tenants see in this “agnostic” setting can be limited. This is especially true for scaling, e.g., if network becomes bottleneck, the scaling benefits would be limited by network.

With growing demand for network services, we believe that cloud providers will be eventually forced to build better support for managing the static and dynamic resource consumption of a tenant’s deployment as a whole. The provider must consider at least two issues to ensure optimal support for end-to-end tenant performance and enable effective scaling decisions: (1) How to *place* a tenant’s initial topology and later middlebox instances added due to scaling, and (2) How to *distribute* traffic among tenant middlebox instances.

We show how Stratos addresses the challenges outlined about in the next three sections.

### 3. NETWORK SERVICE COMPOSITION

Stratos enables cloud tenants to easily manage, monitor, and verify the middlebox traversals they desire without any of the complexity discussed earlier. Instead of composing middlebox and application server topologies through a smattering of third-party tools and configurations, tenants define *logical topologies* using a set of *high-level abstractions*. These topologies are then automatically transformed into a set of forwarding rules that define how application traffic flows between server and middlebox instances. The rules are installed into a programmable virtualized data plane that ensures traffic follows the correct paths. Through this combination of high-level abstractions, a virtualized data plane, and centralized control, tenants are able to deploy rich application and middlebox topologies with even less effort than would be required in a private data center. We begin by presenting the topology abstractions tenants use to compose middlebox and server topologies. Then, we discuss our programmable data plane design: a programmable software switch distributed across VMs (OtC) or hypervisors (UtC).

#### 3.1 Topology Abstractions

In Stratos, tenants specify middlebox traversal require-

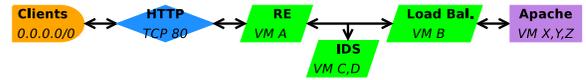


Figure 2: Example chain with middlebox traversals

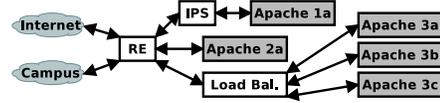


Figure 3: Large server and middlebox topology

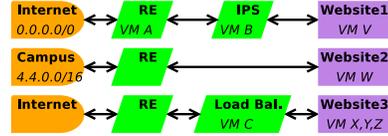


Figure 4: Several chains with repeat elements

ments using a set of high-level topology abstractions that allow tenants to easily codify existing topologies, such as the simple example in Figure 1, and more complex ones.

We use the notion of a *chain* as the basic abstraction for describing the direction specific traffic flows should take. A chain begins with a source of traffic (e.g., Internet clients), contains a sequence of one or more middleboxes the traffic should traverse (e.g., IDS and load balancer), and ends with a destination (e.g., a set of web servers). Chains are defined using four types of elements: *external*, *service*, *middlebox*, and *select*. External and service elements represent two different types of traffic sources/destinations: External is used for hosts outside of the cloud, defined by a set of IP subnets. Service is used for hosts inside the cloud, defined by a set of VMs running a specific application. A specific type of middlebox functionality, which is provided by a set of one or more VMs<sup>1</sup>, is represented by a middlebox element. Lastly, selects are used to limit the traffic traversing (part of) the chain to a specific subset of flows. A select defines a set of header fields and values that identify the traffic subset. The example shown in Figure 2 codifies the path taken by application traffic in Figure 1. Note: each element has a name and a set of subnets, header fields, or VMs it represents.

The set of middlebox traversals specified by a chain only applies to communications between the external subnets and/or service VMs specified in the elements at the start and end of the chain. In the example above, only traffic destined for the web server VMs will traverse the RE, IDS, and load balancer; all other traffic originating from the Internet and destined for some other VM will be forwarded normally. This avoids the need for tenants to specify chains for services whose traffic requires no middlebox traversals.

A tenant’s entire set of required middlebox traversals is specified using one or more chains. For example, the larger logical topology shown in Figure 3 can be specified using three chains, as shown in Figure 4. All of a tenant’s chains are considered in aggregate when programming the data plane (§ 3.2) and making scaling (§ 4) and placement (§ 5) deci-

<sup>1</sup>A cloud provider may also offer middlebox functionality as an add-on service, e.g., Amazon Elastic Load Balancer [1].

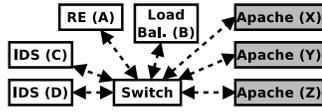


Figure 5: Data plane using a single central software switch. The same external, service, and middlebox elements may exist on multiple chains if they should handle traffic from multiple applications; this does not affect programming of the data plane, but it complicates scaling and placement. Also, chains may be specified as bi-directional or uni-directional, to allow requests and replies to traverse the same or different sequences of middleboxes.

## 3.2 Programmable Data Plane

The flow of traffic codified in tenant-defined chains is realized through the use of a programmable data plane. Such a data plane provides several advantages: (i) the flexibility it provides maps well to the flexibility we desire to provide to tenants, (ii) changes in traffic forwarding can be made easily, allowing us to accommodate potentially frequent changes, e.g., due to scaling, and (iii) it can be automatically managed by a *central controller* capable of transforming tenant-specified topologies into simple forwarding rules, avoiding the need for tenants to directly manage routing policies.

A programmable data plane, and accompanying controller, capable of forwarding traffic between middleboxes and application servers can be realized in several different ways. We first consider following two approaches.

1. *Programmable hardware switches*: The most intuitive approach is to push programmability into the data center network itself and install forwarding rules in each hardware switch along a physical path between two VMs<sup>2</sup>, similar to CloudNaaS [16] and PLayer [24]. However, the controller requires significantly more complexity to compute and install forwarding rules across a large scale network topology, and programmable switching may not be feasible in all cloud data center switches, especially core switches, due to speed constraints and limited rule storage capacity—solutions like DIFANE [38] may be able to help. Moreover, this data plane design is only feasible for an UtC deployment of Stratos in a cloud data center where all hardware switches are programmable, e.g., OpenFlow compatible [27]. Although this may be true in the future, we want to design a framework that tenants or cloud providers could deploy today.

2. *Central programmable software switch*: The second approach we consider lies at the opposite end of the spectrum: a single central programmable software switch for each tenant (Figure 5). With this approach, all communication between a tenant’s VMs occurs via the tenant’s software switch—running in a regular VM (OtC) or on a provider-managed server (UtC). Each of a tenant’s VMs establish a single tunnel to the switch, and all traffic is sent/received via the tunnels based on simple forwarding rules derived from the tenant-

<sup>2</sup>We use the term “VM” to refer to both application servers and middleboxes, as we expect both to run on generic virtual servers.

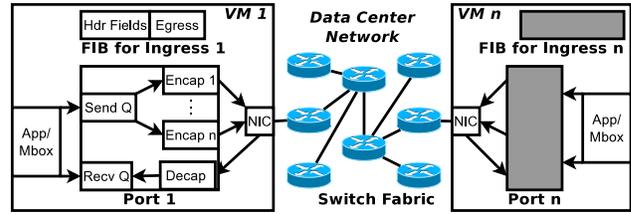


Figure 6: Distributed programmable switch

specified chains. No routing or hardware changes are required in the data center network.

We initially adopted the single-central-switch approach due to its simplicity: having a single point of forwarding significantly simplifies the configuration and monitoring of middlebox traversals. However, bandwidth limitations and lack of reliability outweigh these benefits. Experiments show that a software switch running OtC in a large Amazon EC2 instance can become a bottleneck with as few as three pairs of small instances communicating across the switch and with only one pair of large instances, due to the bandwidth constraints of the VM running the switch. We could use multiple software switches, tunneling traffic from a few VMs to each switch, but this does not guarantee the bandwidth bottleneck is eliminated; we still may require high bandwidth for communication between switches, which in the worst case devolves into using one software switch per-VM. Moreover, if a software switch fails, then all communication between a tenant’s VMs ceases. So we discard this approach.

### 3.2.1 Distributed programmable software switch

In Stratos, we adopt a programmable data plane design that overcomes the limitations of other approaches by dividing the functionality of a *single* programmable switch *across multiple* entities, an approach we refer to as *distributed programmable switch*. The same design applies in both OtC and UtC deployments of Stratos. The only difference is that an OtC deployment implements the necessary switch functionality in each of the tenant’s VMs, while an UtC deployment pushes this functionality to the hypervisor.

Our design is based on the principles employed by a simple hardware switch with three main components: ports, a switch fabric, and a forwarding information base (FIB). Ports receive packets from the network, perform a lookup in the FIB using packet header fields to determine the egress port(s), and place the packets on an appropriate queue. The switch fabric transfers packets from a queue at the ingress port to an output queue at the egress port, and the egress port sends the packets into the network.

Our programmable distributed switch functions the same as this simple hardware switch, except all three components no longer exist on a single piece of hardware. Instead, we move the logic for each switch port to each of a tenant’s VMs (OtC), or to the hypervisor on which the VM resides (UtC), and we rely on the existing data center network to serve as the switch fabric. The FIB is divided among the ports, giving each port the entries necessary for its proper functioning.

This division of functionality is shown in Figure 6.

The switch fabric, i.e., the network, transfers packets across it using normal network routing. Thus, Stratos need not be concerned with the “scheduling” of the fabric, as the paths and ordering of transfers across the fabric will be chosen by the IP routing scheme already in place in the network. However, the switch fabric does need to know the egress port, i.e., the VM or hypervisor to which the packet should be sent. The ingress port communicates this information by encapsulating a packet in an IP packet destined for the VM/hypervisor where the egress port resides.

A *central controller* transforms the tenant-specified chains into entries for the FIB. The FIB entries follow the Open-Flow standard [27], with each entry containing the Ethernet, IP, and TCP/UDP header fields on which to match, the ingress port, and an action to apply—typically, forward out a specific egress port. Dividing the entries among the ports is as simple as adding entries with a given ingress port to the FIB at that port. Although, the FIB entries are divided between ports, the Stratos controller still treats the FIB as if it was a single table on a single switch.

## 4. ELASTICITY

The ability to dynamically scale server capacity is needed is a major advantage of deploying applications in the cloud. However, for application scaling to achieve the desired outcome, middleboxes traversed by application traffic must also be scaled to avoid becoming a performance bottleneck.

The central challenge (§2) is deciding *which* middlebox(es) to scale. Several factors complicate this decision. First, different types of middleboxes require different CPU and memory resources. Several VMs of the same size providing different middlebox functionality will be able to handle varying traffic volumes in a given time frame. Second, middlebox instances (VMs) will be placed at different locations and traffic between them will be routed over different network links; congested links will result in lower volumes of traffic than their uncongested counter-parts. Finally, tenants may construct paths with several sequential middlebox traversals and use middleboxes from the cloud provider [8] or bring their own third-party provided middleboxes [3, 4, 11] to the cloud. Thus, a functionality-independent metric, e.g., per-packet processing time, must be used to decide which middlebox is the bottleneck. Unfortunately, the complexity of most middleboxes, and the presence of confounding network factors like TCP flow control, make it difficult to identify and monitor such a metric.

In this section, we first consider several simple approaches to identify which middleboxes to scale, but we show that these solutions are ineffective or impractical. Instead, we design a greedy heuristic that leverages a measure of end-to-end application performance to make scaling decisions. We conclude by discussing changes in packet forwarding that must occur to take advantage of scaled out middleboxes.

## 4.1 Bottleneck Identification

There are several simple approaches we could leverage for deciding which middlebox(es) to scale, but these solutions are either inefficient or impractical:

- **Scale all middleboxes** The simplest solution for a bottlenecked path is to scale all middleboxes on that path. This guarantees that bottleneck will be eliminated, or at the least reduced, assuming all middleboxes on the path are successfully scaled. However, this potentially wastes significant resources and imposes unneeded costs, especially, when only one middlebox is bottlenecked.
- **Rely on functionality-specific monitoring** Using a custom metric for each type of middlebox based on its semantics—e.g., current number of connections being tracked by an IPS—provides a detailed view of how well each middlebox is performing. However, utilizing these metrics to inform scaling decisions requires knowledge of middlebox semantics. In OtC case, the tenants may not know the middlebox semantics. Even in UtC case, the cloud provider may not know the middlebox semantics, since it may use third-party middleboxes. Furthermore, the performance of different types of middleboxes may not be comparable directly.
- **Rely on per-packet processing time estimates** The average per-packet processing time required at each middlebox provides a common, semantic-independent metric. If a path is determined to be bottlenecked, the middlebox with the greatest increase in per-packet processing time is likely the culprit. However, not all middleboxes follow a one packet in, one packet out convention, e.g., a WAN optimizer. It is unclear how to calculate a meaningful per-packet processing time for middleboxes which do not follow this convention.
- **Scale based on CPU load** Alternatively, we could leverage CPU utilization as a common metric that is independent of functionality. However, purely scaling based on CPU utilization ignores other potential causes of middlebox bottlenecks: high memory utilization, e.g., in TCP optimization middleboxes waiting for out or order packets, or congested network links, which are undetectable through metrics related to the middlebox instance itself.

Ultimately, a tenant is concerned (*i*) their application’s performance and (*ii*) the cost of running their middleboxes. The former concern motivates a heuristic that bases scaling decisions on an application-reported performance metric, scaling out middleboxes when performance noticeably degrades. We believe most applications already track such metrics, e.g., requests/s, and could easily export them to our framework. The later concern, can be addressed through efficient scaling decisions, i.e., only keeping middlebox instances when they enhance application performance.

### 4.1.1 Scaling Heuristic

We design a greedy heuristic that leverages both an application-reported metric and the tenant-specified chains to efficiently examine the space of potential middlebox scaling. The scaling process can be initiated by a significant change in application performance for a sustained period of time.

Our heuristic performs a set of scaling trials, scaling each middlebox in a tenant-specified chain one (VM) instance at a time. We begin by adding a new instance of the first middlebox in the chain, monitoring for changes in the application-report metric over a fixed time window. If application performance improves beyond some threshold, then the new instances is permanently added to the tenant’s topology. We continue to add new instances of the middlebox as long as we see improvement. No improvement means that the middlebox is not a bottleneck, so we discard the new instance. We then move to the next middlebox in the chain and repeat the process. After scaling any later middlebox in the chain, any of the earlier middleboxes may have become a new bottleneck, causing application performance to remain degraded. In this case, we make another pass over the chain. Scaling terminates either when we reach the end of the chain and application performance is suitable, or when a pass over the chain does not result in any new instances being permanently added. Scaling may also terminate early if adding more instances would exceed a budget limit set by the tenant.

Scale down occurs in a similar fashion. We start at the beginning of a chain and remove one instance at a time from the tenant’s topology. If no significant drop in application-reported metric occurs, then the middlebox instance must not be needed to maintain the current level of performance, so we permanently discard the instance. Otherwise, the instance is re-added to the topology and we move on to the next middlebox in the chain. To prevent a constant loop of scale up and scale down, we wait for some time before attempting scaling on a chain again.

We choose to scale only one middlebox at a time because, in most cases, no two middleboxes will be simultaneously and equally bottlenecked. In the rare event we encounter this case, we can perform a “scale all” trial where we add a new instance of every middlebox at the same time. If application performance improves, we add all the instances to the tenant’s topology, and then perform a scale down pass over the chain to remove instances which were added unnecessarily.

The heuristic readily extends to large tenant topologies consisting of many chains because scaling occurs with each chain individually. Hence, the maximum time required for scaling is constrained by the length of the chain being scaled, which anecdotal evidence suggests will usually be only a few elements. Moreover, multiple chains can be scaled in parallel.

## 4.2 Distributing flows among middleboxes

When our heuristic scales a particular type of middlebox on a chain, traffic must be redistributed across middle-

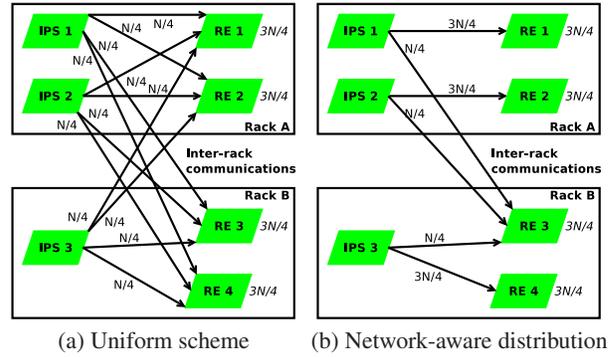


Figure 7: Network-aware flow distribution reduces inter-rack link usages

box instances to leverage the new capacity. Some middleboxes may be stateful, so packets belonging to the same flow should still traverse same middlebox instances.

We first discuss the uniform scheme for flow distribution among middlebox instances, and then we discuss the network-aware scheme for flow distribution. Finally, we discuss how we achieve such distribution in practice and how we transition to new flow redistribution after scaling.

**Uniform Flow Distribution.** In the uniform scheme, each intermediate middlebox instance splits flows equally among all middlebox instances corresponding to the subsequent middlebox element in the chain. Figure 7a shows an example of how flows are distributed among multiple middlebox instances after scaling. The logical topology is a chain connecting IPS to RE and then RE to servers. After scaling, we have 3 IPS instances and 4 RE instances. So if  $N$  flows arrive at each IPS instance, then these  $N$  flows gets distributed equally among the RE instances. Thus each RE instance gets  $\frac{3N}{4}$  flows. This solution can be used OtC.

**Network-aware Flow Distribution.** When Stratos is deployed UtC, the cloud provider can distribute flows in a network-aware fashion, as it knows where VMs are placed. In Figure 7a, the uniform scheme results in  $\frac{3N}{2}$  flows traversing the over-subscribed inter-rack links. Figure 7b shows an alternate scheme, where only  $\frac{N}{2}$  flows traverses the inter-rack links, while each RE instance still gets  $\frac{3N}{4}$  flows as before. Thus, smart flow distribution can significantly reduce inter-rack flows, thereby reducing the likelihood of the network becoming a bottleneck. This improves the performance experienced by tenants and enables the cloud provider to support more tenants.

We now describe a linear program formulation for the problem of distributing flows among the subsequent middlebox instances in the chain. Assume there are  $1..M$  middleboxes in the chain, and  $r^{th}$  middlebox has  $k_r$  instances ( $m_{r,1}..m_{r,k}$ ). For each intermediate middlebox instance  $m_{r,i}$ , we introduce a variable  $x_{r,i,j}$  representing the fraction of traffic distributed to subsequent middlebox instance  $m_{r+1,j}$ , such that  $\sum_j x_{r,i,j} = 1$  and  $0 \leq x_{r,i,j} \leq 1$ .

If there are  $N$  flows entering the chain, then each middlebox instance in the  $r^{th}$  stage should get  $\frac{N}{k_r}$  flows for load

balancing.<sup>3</sup> So,  $\frac{N}{k_r} \times \sum_i x_{r,i,j} = \frac{N}{k_{r+1}}$ .

For every connection between  $m_{r,i}$  and  $m_{r+1,j}$ , we specify a constant  $c_{r,i,j}$ , which is 1 if the connection traverses inter-rack links and 0 if the connection is intra-rack.<sup>4</sup> Then the objective function is to minimize the total inter-rack communications, i.e.,  $\min(\sum_{r,i,j} \frac{N}{k_r} \times c_{r,i,j} \times x_{r,i,j})$ . The solution,  $x_{r,i,j}$ , gives the fraction of flows departing middlebox instance  $m_{r,i}$  that should be passed to middlebox instance  $m_{r+1,j}$ .

We obtain the similar formulation for middleboxes  $M..1$  in the reverse chain, and use variable  $y_{r,i,j}$  to represent the fraction of traffic distributed to the subsequent middlebox instance  $m_{r-1,j}$  in the reverse path.

In case of bi-directional chain, the forward path and reverse path of any flow should traverse same set of middleboxes; this requires a new constraint that number of flows traversing from middlebox instance  $m_{r,i}$  to  $m_{r+1,j}$  should be same as the number of flows traversing from middlebox instance  $m_{r+1,j}$  to  $m_{r,i}$ , i.e.,  $\frac{N}{k_r} \times x_{r,i,j} = \frac{N}{k_{r+1}} \times y_{r+1,j,i}$ .

Now instead of solving  $x_{r,i,j}$  and  $y_{r,i,j}$  separately, we optimize them together, i.e.,  $\min(\sum_{r,i,j} \frac{N}{k_r} \times c_{r,i,j} \times x_{r,i,j} + \sum_{r,i,j} \frac{N}{k_r} \times c_{r,i,j} \times y_{r,i,j})$  and get the solutions for  $x_{r,i,j}$  and  $y_{r,i,j}$ .

**Achieving Flow Distribution in Practice.** We now discuss how flows get distributed across different middlebox instances as per the fractional solution. In one approach, the first packet of every new flow from a middlebox goes to the Stratos controller (§3.2), and the controller selects the next middlebox instance for that flow. The subsequent packets of the same flow traverse same set of middlebox instances. For selecting a middlebox instance, the controller keeps a count of flows<sup>5</sup> spread across middlebox instances. When a new flow comes, the controller chooses a middlebox instance such that the flow distribution approaches the fractional solution. This works fine for large flows, but the overhead of the first packet traversing through the controller is large for small flows. Instead, we take an alternate approach where the subsequent middlebox instance for a new flow is decided at the *vswitch* itself on the host where the VMb resides. This eliminates the overhead of going to the controller for every new flow. Once the decision is made, an entry for the new flow is inserted into the *vswitch* so that the subsequent packets traverse the same set of middleboxes.

In the case of bi-directional chains, requests and responses should traverse the same set of middleboxes. To handle this, whenever we add an entry for the new flow in the *vswitch*, we also add an entry for the reverse path of the new flow in the *vswitch*.

<sup>3</sup>We assume the capacity of all instances of a given middlebox  $m_{r,1} \dots m_{r,k}$  are equivalent.

<sup>4</sup>To provide a more optimal solution,  $c_{r,i,j}$  could instead quantify the link bandwidth available for the tenant's traffic.

<sup>5</sup>We can also extend it to do more sophisticated load balancing, e.g., based on the weighted sum of flows, where weights are assigned based on the number of packets in each flow.

When a new middlebox instance is added for scaling, the fractional solution is recomputed. The challenge here is to move from the existing fractional solution to the new fractional solution quick enough, so that the middlebox bottleneck is resolved quickly. Initially, we assign most of the new incoming flows to the added middlebox instance while being careful not to make it a bottleneck. We keep count of flows (new + existing) on all middlebox instances, and as the flow distribution comes closer to the new fractional solution, the new incoming flows are distributed as per solution.

## 5. PLACEMENT

Where we place the middleboxes can significantly affect the performance of applications in the cloud. If not placed carefully, they may cause the network to be a bottleneck, worsen the performance of applications and reduce any scaling benefits. We discuss the importance of middlebox placement below.

- **End-to-end application performance.** The placement of application and middlebox VMs within a cloud data center determines which network links an application's traffic must traverse. If we don't place middlebox VM in the same rack as the application VM, it will cause application traffic to traverse oversubscribed inter-rack links that can impact end-to-end performance.
- **Effectiveness of middlebox scaling.** The volume of traffic a middlebox instance can process depends on both the capacity of the VM and the available bandwidth. A middlebox VM which is reachable only via oversubscribed links will be bandwidth limited, causing it to process less of a tenant's traffic. Thus, the performance benefit of this additional middlebox instance will be reduced, and the added cost for the instance may outweigh the benefits it provides.
- **Network performance across tenants.** When a tenant uses more bandwidth along a shared link, e.g., to direct traffic to a middlebox in a different rack, less bandwidth will be available for other tenants. Placing a tenant's VMs in a way that reduces the use of shared links, e.g., placing both the middlebox and application VMs in the same rack, reduces the likelihood of a tenant degrading the performance experienced by others.

When Stratos is deployed OtC, tenants have little control over placement. They are subject to the existing placement decisions of cloud providers, which tend to focus on the placement of individual VMs without regard to the traffic between them. There are some limited options for control over placement in OtC deployments. Recent research [31] has uncovered choices that provide some influence over placement in Amazon EC2. First, VMs of the same size fall within the same set of subnets and are likely to be more closely placed than VMs of different sizes; a new VM should therefore be made the same size as the VMs which immediately precede or follow it on tenant-specified paths. Second, VMs

which are launched at about the same time are more likely to be placed nearby; hence, VMs on the same path should be launched as close together in time as possible. While it may be possible to leverage these techniques to exercise control over placement, it is not clear how generic they are.

UtC deployments of Stratos allow for many more placement options, resulting in substantially more performance and scaling benefits. The key is to consider a tenant’s topology as a whole when making placement decisions. This includes consideration of which VMs traffic flows between and the volume of traffic between them—both of which can be garnered from the tenant-specific chains. Moreover, placement should consider the fact that middleboxes (and applications) will likely be scaled in the future, requiring the placement of VMs that communicate with already placed VMs. We first present an algorithm for the initial placement of a tenant’s VMs; then we present an algorithm for placing new middlebox instances.

### 5.1 Initial VM Placement

The initial VM placement is important for the performance of applications, as well as, for realizing the scaling benefits. For example, consider that  $IPS \rightarrow RE \rightarrow server$  is the tenant-specified chain and there is one VM instance for each type,  $IPS$ ,  $RE$  and  $server$ . Initially, VM instances for  $IPS$  and  $RE$  are placed in one rack and the VM instance for  $server$  is placed in another rack. If we add a new  $RE$  instance, no matter where we place the new instance, it would cause the inter-rack flow traversals. If the available bandwidth between the two racks is low, the inter-rack flow traversals would cause network to be bottleneck and reduce the performance benefits of adding the new  $RE$  instance.

However, if we place VM instances in the same rack and the rack has enough available slots, adding a new  $RE$  instance would not cause any inter-rack flow traversals. Even when we have to place VMs in different racks, as long as there is enough available bandwidth between them, the network would not become bottleneck in scaling middleboxes. Thus, it is important to place VMs carefully in the beginning.

We now discuss our initial placement algorithm (overview in Figure 8). The algorithm takes as input the tenant-specified chains which describe the flow of traffic between middlebox and application VMs (§ 3.1). We derive the VM-to-VM communication matrix based on these interactions. We also take into consideration the fact that some VMs may require more scaling than others. For example, a middlebox is likely to be scaled if it is shared by multiple applications in the tenant-specified paths. To accommodate such potential *future* scaling easily, we “reserve” few extra VMs for such middleboxes in the VM-to-VM communication matrix.

We assume the available bandwidth of all links in the data center topology could be available as input. The available bandwidth is based on some metric (e.g., mean, maximum or  $k^{th}$  percentile) of link usages over a certain time window. The choice of the specific metric depends on how

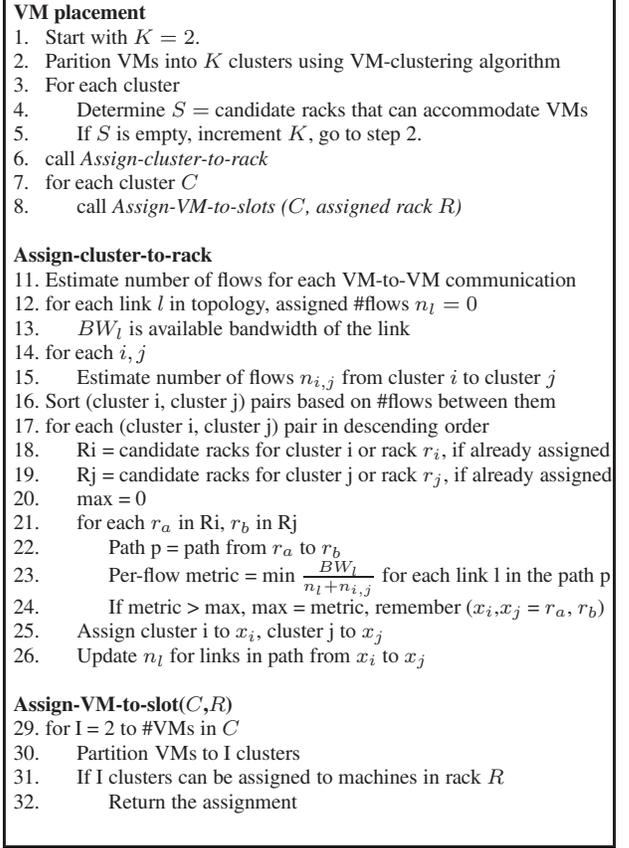


Figure 8: VM-placement algorithm

aggressively we want to place VMs such that the network does not become a bottleneck.

**VM Placement :** We adapt the classical min-cut algorithm in a manner similar to [28] and partition VMs into  $K$  clusters. The clustering tries to minimize the number of communications across clusters so that when clusters are assigned to racks, there are few inter-rack communications. After obtaining the clusters, they are assigned to racks in such a way that the inter-rack flows get high available bandwidth. The routine *Assign-cluster-to-rack* is called for this purpose. Then, for each rack, the VMs are assigned to slots in the physical machines by calling the routine *Assign-VM-to-slot*.

**Assign-cluster-to-rack :** We first estimate the number of flows for each VM-to-VM communication assuming the uniform flow distribution (§ 4) and equal number of flows per chain. While this is a simplifying assumption, we believe that it is helpful in achieving the goal of minimizing the number of VM communications traversing links with low available bandwidth.

We first assign cluster-pairs having large number of flows across them. The intuition here is to first provide high available bandwidth to these large number of flows, so that later on, it is not the case that enough bandwidth is not available for them. For a given such cluster pair, we explore their candidate racks and find the rack pair with high available band-

width on the path between the racks. We approximate this metric by looking for situations where there are few flows already assigned on the links in the path and the links have high available bandwidth. This is shown in line 23 in Figure 8 when there is a single inter-rack path<sup>6</sup>.

We then pick the next cluster-pair (in descending order of number of flows across them) and assign them to clusters in the similar manner. At the end of this routine, we have an assignment of clusters to racks.

**Assign-VM-to-slot:** After assignment of clusters to racks, we assign VMs within the cluster to slots in the racks. In case there is just one slot per machine, we randomly pick a slot and assign it to a VM. If there are more than 1 slot per machine, we follow a similar procedure as above, namely, of partitioning the VMs into multiple clusters so that VMs that communicate more among each other can be assigned to the same machine (details omitted for brevity).

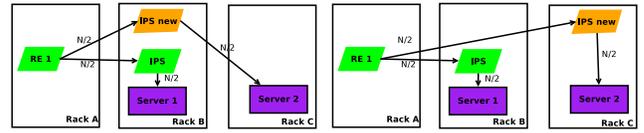
Note that the above description applies for cases where we don't have the bandwidth demand estimates for tenant's VM-to-VM communications. If the tenant can provide this information then the cloud provider can place VMs more effectively. In such a scenario, we modify **Assign-cluster-to-rack** algorithm as follows: For assigning a pair of clusters to a pair of racks, we find the minimum available bandwidth along the path, and see if it can accommodate the VM-to-VM communications between them. We select the rack-pair that maximizes the number of VM-to-VM communications whose bandwidth demands can be accommodated. After each assignment, we adjust the available bandwidth of each link appropriately.

## 5.2 Placement of New Middlebox Instances

The cloud provider can place new VMs (due to scaling) in a network-aware fashion so that the scaling benefits can be realized more effectively. Figure 9 shows an example, where a new VM for IPS has to be placed in either rack B or rack C. If it is placed in rack B, the total number of inter-rack flows is  $\frac{3N}{2}$  using network-aware flow distribution (§ 4.2), where  $N$  flows enter the RE middlebox. However, if it is placed in rack C, the total number of inter-rack flows is only  $N$ . Thus, appropriate VM placement can significantly reduce inter-rack flows benefitting both the tenant's scaling exercise as well as the provider's ability to support more tenants.

We first consider the case for single tenant, when (s)he adds a new VM. If the middlebox instance can be accommodated in the same rack as its "input VMs" (those supplying it traffic) and "output VMs" (those receiving its traffic), we place the new middlebox instance in the same rack. However, if the new middlebox instance cannot be accommodated in the same rack, we select a candidate rack (rack with free slots) that has the maximum available bandwidth to the

<sup>6</sup>In case there are  $P > 1$  paths between two racks,  $\frac{1}{P}$  of VM-flows between the two racks would be assigned to each path. Following this, we estimate the number of flows going through each link, find the bandwidth share of each flow on the link, and find the minimum value among all flows on all links. This is not shown in the figure.



(a) New instance in Rack B (b) New instance in Rack C  
Figure 9: Better placement with scaling can significantly reduce inter-rack traffic

rack for input and output VMs. When the input and output VMs are in different racks, we consider each candidate rack and estimate the inter-rack VM traffic using network-aware flow distribution, assuming that the new VM is placed in the candidate rack. We select the rack that minimizes the inter-rack VM traffic (or maximizes the bandwidth available to inter-rack flows). We also consider the input and output traffic ratio for the middlebox instance. This ratio can be learned based on past usages; for example, with 50% redundancy in traffic, a WAN optimizer would have an input:output ratio of 1:2 (traffic entering (leaving) a WAN optimizer is compressed (decompressed)). Based on this ratio, we can assign weights to input and output flows and minimize the weighted sum of inter-rack flows.

We handle the case of multiple tenants carefully. Multiple tenants can scale their middleboxes in parallel. This may create issues when the increased traffic due to their scaling can create network bottlenecks. Consider an example with tenants  $T1$  and  $T2$ , where both of their VMs are placed in rack  $A$  and rack  $B$ . Both tenants perform scaling trials in parallel. Consider that based on current bandwidth usages, our algorithm places their new VMs in rack  $B$  and that causes inter-rack traffic between rack  $A$  and rack  $B$ . This can become problematic, when due to increased inter-rack traffic, the network becomes bottleneck and both tenants don't see any improvement in their application performance. As a result, both tenants end up not being able to satisfy their traffic demands.

We handle this issue as follows. When tenant  $T1$  makes a request for addition of a new VM and our algorithm makes a placement decision of the new VM, we also estimate the increase in bandwidth usage of inter-rack links and use these estimates to determine the available bandwidth; the actual increase in bandwidth usage would happen when the new VM is actually placed, but other tenants' requests for scaling might have come before that. So when tenant  $T2$  makes a request for a new VM, we know that placing a new VM in the rack  $B$  can create network bottleneck, so we try to find other placement. It may happen that both tenants make request for new VMs at the same time. In that case, we choose an ordering between  $T1$ 's request and  $T2$ 's request and serve them one after another.

## 6. IMPLEMENTATION

We implement a prototype of Stratos capable of running OtC on Amazon EC2. Each tenant VM includes two pieces of software to realize the port functionality of our distributed

programmable switch: Click [25] and Open vSwitch [7]. Click holds the responsibility for encapsulating packets for transmission across the switch fabric (i.e., network) to other ports (i.e., VMs). We use `FromHost` and `ToDevice` Click elements to send/receive traffic via tap interfaces. Each port instance has a tap interface for every other port in the switch. Packets sent to one of the tap interfaces are encapsulated in an IP/UDP packet destined for the IP of the port (VM) that tap interface represents.<sup>7</sup> There is also a tap interface for reaching the host on which the port resides; the host is configured to use this tap for reaching all EC2-internal IPs. Open vSwitch performs the FIB lookup and queuing functionality of the port. The FIB consists of OpenFlow [27] rules, installed by the controller. The rules determine which queue (i.e., tap) a packet should be sent to, resulting in the traffic being sent to the host or across the fabric to a port.

Our central controller runs NOX [21] and interfaces with the Open vSwitch instances using the OpenFlow protocol [27]. We wrote our own NOX module that takes as input a topology specification file. The specification defines each middlebox and application instance, indicating the distributed switch port to which they are “connected”. Paths are specified as a sequential list of elements. The controller transforms this topology specification into a set of forwarding rules which are installed in each Open vSwitch instances.

## 7. EVALUATION

We evaluate Stratos using both our prototype implementation in Amazon EC2 (to study OtC deployment) and a set of simulations (to study UtC deployment). We explore how well each of the components of Stratos—composition, elasticity and placement—contributes towards our goal of allowing cloud tenants to flexibly compose virtual topologies including middleboxes and dynamically scale to match current demands. We also show how our mechanisms can help improve provider utilization (UtC).

### 7.1 OtC Deployment

We evaluate an OtC deployment of Stratos using our prototype implementation (§6) in Amazon EC2 [1]. First, we evaluate the end-to-end application performance achievable with our programmable distributed switch (§3.2.1). Second, we evaluate the effectiveness of our greedy scaling heuristic (§4.1.1) on individual tenant-specific chains.

#### 7.1.1 Composition

**Methodology.** We evaluate the performance of Stratos deployed OtC using the tenant topology shown in Figure 4. The website VMs run Apache 2.2.17 [2] with a default configuration and serve either a 100KB or 10KB text file. External client subnets are represented by three EC2-resident VMs (which we refer to simply as “clients”) generating web

<sup>7</sup>We use Ethernet-in-UDP encapsulation because EC2 does not allow IP-in-IP or GRE.

Task	Time
Topology transformation	14ms
Port setup	23.33s (average 2.12s per port)
Rule installation	54ms (average 5ms per port)

Table 1: Distributed programmable data plane setup costs

requests using httpperf 0.9 [6]. The middlebox VMs run either a redundancy elimination module written in Click [13] (RE), Suricata 1.2.1 [11] with the default ruleset (IPS), or Balance 3.54 [3] (Load Balancer). All of the VMs are large Amazon EC2 instances. Each VM also functions as a port in our distributed switch, as described in §6. The controller runs on a micro Amazon EC2 instance and installs forwarding entries in the ports based on the specified chains.

**Setup Cost.** We first measure the setup overhead associated with our distributed programmable switch. The setup cost includes the time required to transform the topology specification into forwarding rules, launch a port on each VM, and install appropriate forwarding rules in each port (Table 1). The total setup time of the distributed programmable switch for our example topology is  $\approx 23.4$ s, a fairly low overhead.

Note that the time required to launch VMs, and install any required software, is not included in this setup time. Since a port of the programmable distributed switch cannot be setup until after a VM is launched, the small per-port overhead of  $\approx 2$ .1s will be incurred after a VM is launched.

**Application Performance Impact.** Next, we evaluate the effect our programmable data plane deployed OtC has on application performance. For this evaluation, we remove all middlebox processing from our deployment and allow the clients to directly communicate with the web servers to request/receive the 100KB text file. We measure the reply throughput under increasing client request rates both with and without the use of our programmable distributed switch. We observe that the performance with and without the programmable distributed switch is equivalent—all requests are served—for loads across all three clients totaling  $\geq 750$  requests/s. With a total load of 900 requests/s, 7% of the requests cannot be served when using the programmable distributed switch while all requests can be served without it. However, increasing the request load even further (1050 requests/s) causes not all requests to be served even without the programmable distributed switch. Thus, the maximum application load that can be handled with our programmable distributed switch is only slight less than the maximum load possible without, implying that our data plane setup imposes minimal additional overhead.

**Middlebox Bottlenecks.** We empirically show how the presence of middleboxes may limit application performance. We generate increasing volumes of client requests, and measure the achieved request throughput. We assume no scaling is employed. Figure 10 shows the achievable throughput across different request loads when all three clients use the same request rate. Note that Client 2 requests the 100KB text file, while Clients 1 and 3 request the 10KB text file; Client 2 therefore imposes about 10x as much traffic load at

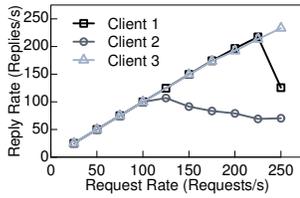


Figure 10: End-to-end application performance with equal client load with programmable distributed switch OtC

the same request rate.

We observe that under small request load ( $\leq 100$  requests/s), the reply throughput matches the request rate for all three clients, but higher request loads start to bottleneck middleboxes. The RE middlebox is the first middlebox to become bottlenecked because it is shared by all three clients. However, the bottleneck has a greater impact on Client 2, which has larger flows, compared to Clients 1 and 3: with 150 requests/s, Client 2 only has 61% of requests served, while Clients 1 and 3 still have almost all of the requests served; Clients 1 and 3 only start to be modestly affected with more than 200 requests/s. The 50% drop in performance for Client 1 with 250 requests/s is a result of the IPS middlebox becoming bottlenecked. The significant effects of middlebox bottlenecks on end-to-end performance motivates the need for scaling, which we evaluate in the next subsection.

The only difference between an OtC and an UtC deployment of our programmable distributed switch is that the port functionality is pushed into the hypervisor. This avoids CPU overhead in tenants' VMs, pushing the CPU load to the hypervisor. Hence, we expect tenants may be able to serve slightly higher maximum loads with an UtC deployment.

### 7.1.2 Elasticity

We now evaluate how well our greedy scaling heuristic (§4.1.1) can address bottlenecks such as the ones exemplified above in an OtC deployment of Stratos.

**Methodology.** We use a single chain for our scaling evaluation, consisting of client, IPS, RE, and server. Similar to the setup above, Apache serves a 100KB text file, and client requests are generated by httpperf. Both requests and replies pass through an IPS [11] and a RE [13] middlebox. Apache and httpperf run on non-bottlenecked large EC2 instances; all middleboxes are small instances, a size purposely chosen to induce bottlenecks. We induce middlebox scaling by increasing the request rate from 25 to 100 requests/s over a 3 minute period; scale down is induced by decreasing the request rate to 50 requests/s.

**Results.** Our first concern is the ability to detect the presence of a bottleneck in order to initiate scaling. Response time is a simple and effective metric for this purpose. When the request rate increases from 25 to 50 requests/s, causing middleboxes to be a bottleneck, the response time increases from 5ms to 2742ms; when the bottleneck is eliminated, the response time drops back to 77ms.

The behavior of our scaling heuristic is shown Figure 11a, which depicts application throughput and the number of IPS

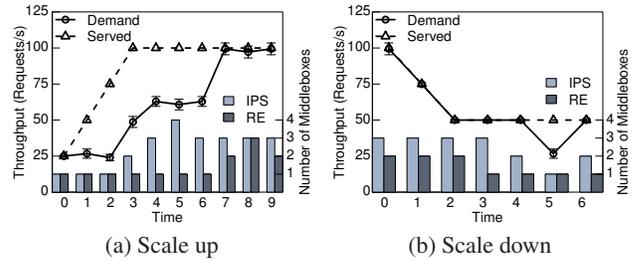


Figure 11: Greedy scaling heuristic

Approach	Max Throughput	Rqsts/s/cost	Time
Stratos scaling	99 requests/s	19.8	9min
No scaling	23 requests/s	11.5	0min
All scaling	97 requests/s	10.8	3min

Table 2: Comparison of middlebox scaling approaches

and RE middleboxes over time while our heuristic is running. With only 1 IPS and 1 RE middlebox, performance is quite poor ( $\approx 25$  requests/s). The first scaling action, increasing to 2 IPSs, increases throughput by 50% (to  $\approx 50$  requests/s), and a second scaling, to 3 IPSs, increases throughput an additional 23% (to 63 requests/s). Scaling to 4 IPS is tried, but the 4th IPS is not kept due to the increase in throughput being not significant enough. The algorithm then moves on to scale the number of RE middleboxes, increasing to 2 REs for 37% increase in throughput (to 99 requests/s). Scaling to 3 REs is tried, but the 3rd RE adds no improvement in performance. In summary, we scale from 1 IPS and 1 RE middlebox to 3 IPS and 2 RE middleboxes over a period of 9 minutes, increasing application performance by 75%. The resulting throughput, 99 requests/s, closely matches the throughput of a setup with no middleboxes.

We also evaluate how well Stratos is able to scale down when request load decreases. Figure 11b shows the application throughput and number of middleboxes over time when we cut the request load in half. Stratos first scales down to 1 RE and then scales down the IPSs to 2, ending with a request throughput that is exactly the same as the throughput with no middleboxes. The number of IPSs is not scaled back to 1 as this causes a 46% decrease in request throughput.

We now compare our scaling algorithm against two alternative approaches: *no scaling* and *scaling all middleboxes*, i.e., scale up creates one new instance of each middlebox. Table 2 shows the maximum achievable throughput (showing end-to-end performance), the ending requests/s/unit-cost (showing cost-effectiveness), and the time required to scale for all three approaches (showing delay overhead). We observe that Stratos scaling is 42–45% more cost effective than no scaling and all scaling. No scaling and all scaling are both faster than Stratos scaling, but this time savings is offset by their inefficiency in the long run.

**Summary.** An OtC deployment of Stratos enables tenants to realize rich topologies including middleboxes using our programmable distributed switch without a significant degradation in end-to-end application performance. Moreover, our greedy scaling heuristic is able to adequately eliminate middlebox bottlenecks, providing suitable end-to-end per-



Figure 12: Chain used for simulation

formance in a cost-effective manner.

## 7.2 UtC Deployment

We next evaluate the enhancements that can be made to end-to-end application performance and the effectiveness of scaling in an UtC deployment of Stratos through improved placement and flow distribution. We briefly describe the simulator we developed, then we show the benefits improved placement (§5) and flow-distribution (§4.2) provides to both tenants and cloud providers.

**Methodology.** We developed a simulator to evaluate our placement and flow-distribution heuristics at scale for an UtC deployment of Stratos. The simulator takes as input a data center topology of racks and switches, a tenant chain with elements and initial instances, the number of tenants, and a fixed application demand (in Mbps). First, the simulator places all of the tenants’ initial instances in racks either by randomly selecting a rack for each VM (*rand*) or by using our initial VM placement heuristic (*pack*). We do not leave space in racks for future scaling. Second, the simulator runs our greedy scaling heuristic for one tenant at a time. Scaled instances are placed either by randomly selecting a rack (*rand*) or by using our network-aware middlebox instance placement heuristic (*aware*). Traffic is distributed amongst middlebox instances using either *uniform* or *network-aware* flow distribution. Each tenant runs the greedy scaling heuristic until its full demand is satisfied or no further performance improvement can be made. We assume intra-rack links are very high capacity.

For our UtC evaluation, we run our simulator to place 20 tenants within a 50-rack data center. The data center is arranged in a tree topology with 7 VM slots per rack and a capacity of 500Mbps on each network link. All tenants use the same chain (Figure 12) which initially consists of 9 VMs; thus every tenant is forced to spread their VMs across racks. The capacity of each instance of the WAN optimizer, IPS, and cache is fixed at 100, 120, and 220Mbps, respectively. The application demand between each client and server pair is 180Mbps, for a total traffic demand of 360Mbps.

### 7.2.1 Placement

We first use our simulator to evaluate our placement heuristics (§5), comparing our initial placement (*pack*) and new middlebox instance placement (*aware*) against random placements in either case (*rand*). We assume traffic is distributed amongst middlebox instances using uniform distribution (§4.2); we evaluate smarter distribution in the next subsection.

**Tenant View: Aggregate Performance.** Since end-to-end application performance is paramount, we first look at the tenant demand that can be served under three different combinations of initial and middlebox instance placement schemes: *rand/rand*, *pack/rand*, and *pack/aware*. Figure 13 shows a CDF of the percentage of demand that is able to be served

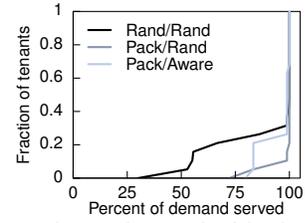


Figure 13: Tenant demand served with placement schemes

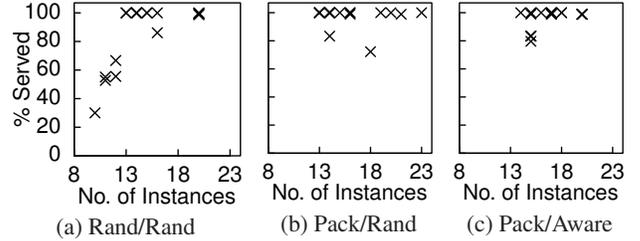


Figure 14: Demand served vs. instances used for each tenant

for each tenant. We observe that almost all demand is served for about 75% of tenants with our *pack* and *aware* placement heuristics. Moreover, no tenant has more than 20% of its demand unsatisfied. In contrast, tenants may have up to 70% of their demand unsatisfied with *rand/rand*, although 70% of tenants have almost all of their demand served.

Using our initial VM placement heuristic with random instance placement (*pack/rand*) results in more tenants having almost all of their demand served compared to using *pack/aware*. In both cases, the traffic flows between the tenants initial instances over only a few localized links, but traffic exchanged with new middlebox instances is only localized with *aware* instance placement. This increase in localized traffic, makes it more likely the edge links of racks used by the tenant will become bottlenecked, and causing fewer tenants to have their demand fully served. Adding bandwidth-awareness to instance placement, similar to our awareness of bandwidth in initial placement, can reduce such localized bottlenecks by spreading new instances among more racks, and effectively more links. This effect can also be diminished using network-aware flow distribution in combination with *aware* placement, which we show later.

**Tenant View: Cost vs. Performance.** In the optimal case, just enough middlebox instances are added for a tenant to provide sufficient processing capacity to meet demand. Due to system dynamics, however, some tenants may use more or fewer instances than ideal and may still not be able to meet demand. We now examine this issue in detail. Figure 14 shows the percent of demand served versus the number of instances used, with a data point for each tenant. The optimal number of instances required to meet demand is 13: 2 clients, 4 WAN optimizers, 3 IPSs, 2 caches, and 2 servers.

We first focus on tenants whose demand was not fully served. Two cases arise: (i) *Tenant has fewer than the ideal number of instances:* In this case, any new middlebox instance is unable to improve end-to-end performance (i.e., increase served demand) because the bandwidth available to reach the new instance (from the instances that precede or

follow it in the chain) is insufficient; this causes the new instance to be discarded and scaling to quit. This case only occurs with *rand/rand*, for 25% of tenants. Random placement distributes instances amongst many racks, forcing almost all traffic to traverse network links, thus increasing the presence of network bottlenecks that cannot be overcome by using more instances. (ii) *Tenant has more than the ideal number of instances*: Although tenants may have added enough middlebox instances to provide sufficient processing capacity (compared to the ideal amount needed to match the demand), the network bandwidth may be insufficient to handle the tenant demand when a tenant attempts to fully utilize the capacity of a middlebox instance. For example, at the start of scaling, a tenant will add a new WAN optimizer instance, increasing the middlebox capacity to 200Mbps. The end-to-end demand that can be served now becomes limited by the IPS, which can only process 120Mbps with a single instance. Thus, during the scaling experiment for the WAN optimizer, only 60Mbps of traffic was sent to each WAN optimizer instance. When the IPS is scaled, the traffic going to each IPS will increase to 100Mbps, which the network may not be able to handle. Even though sufficient middlebox processing capacity is available, the network bandwidth available to some instances is insufficient. This can cause more middlebox instances to be added, with the tenant eventually giving up. This case occurs for 5%, 10%, 25% of the tenants with *rand/rand*, *pack/rand*, and *pack/aware*, respectively. The higher occurrence of this case with smarter placement occurs due to higher instance locality—instances are placed in just a few nearby racks—causing localized network bottlenecks that only manifest when attempts are made to utilize the full processing capacity of middlebox instances.

We now consider tenants whose demand was (almost) fully met. Demand can be fully met when enough non-network-bottlenecked middlebox instances are launched to provide at least as much processing capacity as the optimal number of instances provides. This occurs for 70%, 90%, and 75% of tenants with *rand/rand*, *pack/rand*, and *pack/aware*, respectively. Fewer tenants fully meet the demand with our scaled VM placement heuristics, compared to *pack/rand* due to the localized network bottlenecks that occur, as discussed above. Such bottlenecks can be avoided with smarter flow distribution, as we show in the next subsection.

### 7.2.2 Flow Distribution

In addition to optimizing the placement of new middlebox instances, we also seek to optimize the distribution of flows among instances to reduce the likelihood of encountering network bottlenecks. We compare the *uniform* and *network-aware* methods of flow distribution (§4.2) with both *rand* and *aware* placement of new middlebox instances.<sup>8</sup>

**Tenant View: Performance.** Figure 15a shows the percent of tenant load that is able to be served with the two different distribution mechanisms. We observe that all load

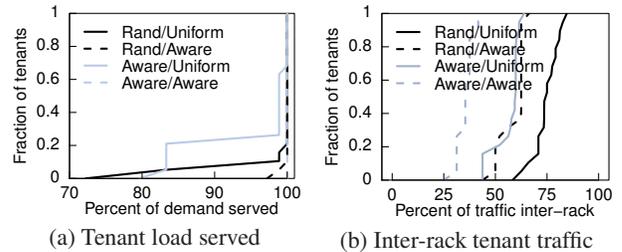


Figure 15: Tenant outcomes of distribution schemes

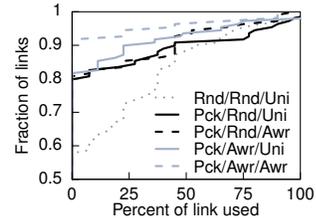


Figure 16: Network link utilization

is able to be served for all tenants with both *aware* placement and *aware* distribution. In contrast, with uniform flow distribution, network-aware placement is unable to provide the maximum benefit, with 25% of tenants having less than (almost) their full demand served. Using *aware* distribution with *rand* placement also provides benefits, decreasing the max unserved demand by 89% (from 28% to 3% unserved). **Tenant View: Cost vs. Performance.** We also see a reduction in the number of middlebox instances required to meet tenant demand: With *aware* distribution and placement, 45% of the tenants use the optimal number of instances (13) and the remaining tenants only use one more than optimal.

We affirm the performance improvement and reduction in middlebox instances required is a result of a decrease in inter-rack network bandwidth by examining the link bandwidth used by individual tenants. Figure 15b shows a CDF of the percent of tenants’ traffic which crosses inter-rack links for each tenant. *Aware* distribution results in a significant decrease in inter-rack traffic for the median tenant with both placement algorithms: 69% with *aware* placement and 20% with *rand*. Moreover, *aware* distribution with *rand* placement is able to decrease the inter-rack traffic to levels equivalent to using *basic* distribution with *aware* placement.

### 7.2.3 Provider View

Our UtC placement and distribution heuristics also benefit providers by reducing the network and rack utilization.

First, we compare network link utilization across all combinations of placement and distribution we consider. Figure 16 shows a CDF of the percent of link bandwidth used for each link in the data center. The highest fraction of network links (43%) are utilized with *rand* initial placement, because all of tenant’s instances are spread across the data center causing significant inter-rack communication. Using *pack*, instead of *rand* initial placement reduces the traffic traversing core links, but this is only beneficial when edge links are also less utilized. A reduction in both core and edge link utilization only occurs with the localized placement and

<sup>8</sup>Initial placement uses our *pack* heuristic.

	Uniform Distribution	Aware Distribution
Rand/Rand Place	87%	-
Pack/Rand Place	90%	78%
Pack/Aware Place	97%	77%

Table 3: VM slot utilization across all racks

minimized inter-rack communication that occurs with *pack* initial placement, *aware* instance placement, and *aware* distribution. In this case, the smallest fraction of network links (only 10%) are utilized, leaving the most network capacity for supporting additional tenants.

The spare network capacity available with *aware* instance placement and distribution is accompanied by the largest number of unused VM slots in racks. Table 3 shows the percent of slots used under all evaluated combinations of placement and distribution. With *aware* instance placement and distribution, tenants perform less unnecessary scaling, as noted earlier, leaving an additional 20% of slots unused compared to *aware* instance placement and *basic* distribution and an additional 13% of slots unused compared to *rand* instance placement and *basic* distribution.

**Summary.** Using our network-aware initial placement and distribution heuristics leaves 90% of links and 23% of VM slots unutilized, allowing a provider to support more additional tenants than under other schemes.

## 8. RELATED WORK

**Networked services in the cloud:** Recent proposals [8, 5, 12, 16] and third party middleware [12] have begun to incorporate limited support for middleboxes. CloudNaaS [16], CloudSwitch [5] and VPNCubed [12] aim provide flexible composition of virtual topologies; however, they don't have the mechanisms for scaling of networked services. Embrane [8] introduces a framework that allows for the flexible scaling of networked services. However, it is limited to provider-offered middleboxes, and does not allow composing them with each other or with third-party VMbs.

Studies have looked at the properties of clouds that impact application performance [36, 26], introduce security violations [31], and that affect application reliability [34]. Others have sought to enrich the networking layer of the cloud by adding frameworks that provide control over bandwidth [14, 22], security [23, 18, 29], and performance of virtual migration [37]. All of these are largely complementary to Stratos, but understanding how to integrate these systematically into Stratos is a non-trivial issue that we leave for future.

**Middleboxes in enterprises and datacenters:** Issues in deployment and management of middleboxes have been examined in the context of enterprise [32] and data-center [24] networks. But the focus is on composition in physical infrastructures and thus performance challenges introduced by the lack of tight control in clouds are not addressed.

**VM Placement:** Oversubscription within current data center networks and its impact on application performance and link utilizations have been widely studied [36, 26, 15]. While many have sought to overcome the performance issues by

modifying routing [17, 30] or altering the data center's architecture [20, 19, 35], only a few recent works [16, 28] have explored using VM placement as a solution. Our placement approach is novel in that it utilizes both knowledge about the communication patterns between the VMs and the future scaling demands to determine placement. Unlike prior work, which focuses on placing individual VMs in isolation, our work focuses on discovering groups of related VMs with dense communication patterns and collocating them.

**Scaling in the Cloud:** Recent studies [33, 1, 10] have viewed elasticity within the cloud as scaling the number of virtual machines in each tier of a tenant's hierarchy. While many of these studies rely on CPU utilization to determine scaling, we use a more comprehensive approach that examines the impact on end-to-end application performance and helps identify scaling strategies that may not be apparent from per VM CPU monitoring (e.g., when multiple MBs show high resource utilization, when memory is the bottleneck resource, or when there is network contention).

## 9. CONCLUSION

Enhancing application deployments in today's clouds using virtual middleboxes is challenging due to the lack of network control and the inherent difficulty in intelligently scaling middleboxes to cope with application demands. Overcoming the challenges in a systematic way requires a new ground-up framework where middleboxes are elevated to a first class citizen in the cloud, akin to other cloud resources such as storage. In this paper, we presented the design, implementation, and evaluation of one such framework called Stratos that is targeted for immediate deployment OtC or UtC today. We developed a variety of mechanisms for efficiently realizing arbitrary middlebox and application compositions and providing suitable end-to-end performance through greedy scaling and network-aware flow distribution and placement. Our evaluation of a prototype OtC deployment of Stratos atop EC2 and simulation of an UtC deployment showed the benefits Stratos provides to both tenants and providers.

## 10. REFERENCES

- [1] Amazon web services. <http://aws.amazon.com>.
- [2] Apache. <http://apache.org>.
- [3] Balance. <http://www.inlab.de/balance.html>.
- [4] CloudOpt. <http://cloudopt.com>.
- [5] CloudSwitch. <http://www.cloudswitch.com>.
- [6] httpperf. <http://hpl.hp.com/research/linux/httpperf>.
- [7] Open vSwitch. <http://openvswitch.org>.
- [8] Powering virtual network services. <http://embrane.com>.
- [9] Rackspace cloud. <http://rackspace.com/cloud>.
- [10] Right Scale. <http://www.rightscale.com>.
- [11] Suricata. <http://openinfosecfoundation.org>.
- [12] VPN-Cubed. <http://cohesiveft.com/vpncubed>.
- [13] A. Anand, C. Muthukrishnan, A. Akella, and R. Ramjee. Redundancy in network traffic: Findings and implications. In *SIGMETRICS*, 2009.
- [14] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards Predictable Datacenter Networks. In *SIGCOMM*, 2011.
- [15] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *IMC*, 2010.
- [16] T. Benson, A. Akella, A. Shaikh, and S. Sahu. CloudNaaS: A cloud networking platform for enterprise applications. In *SoCC*, 2011.
- [17] T. Benson, A. Anand, A. Akella, and M. Zhang. Microte: fine grained traffic engineering for data centers. In *CoNEXT*, 2011.

- [18] C. Dixon, H. Uppal, V. Brajkovic, D. Brandon, T. Anderson, and A. Krishnamurthy. ETTM: a scalable fault tolerant network manager. In *NSDI '11*.
- [19] N. Farrington, G. Porter, S. Radhakrishnan, H. H. Bazzaz, V. Subramanya, Y. Fainman, G. Papen, and A. Vahdat. Helios: a hybrid electrical/optical switch architecture for modular data centers. *SIGCOMM*, 2010.
- [20] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VI2: a scalable and flexible data center network. In *SIGCOMM*, 2009.
- [21] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: towards an operating system for networks. *ACM SIGCOMM Computer Communication Review*, 38(3), 2008.
- [22] 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 *Co-NEXT '10*.
- [23] F. Hao, T. Lakshman, S. Mukherjee, and H. Song. Secure Cloud Computing with a Virtualized Network Infrastructure. In *HotCloud*, 2010.
- [24] D. A. Joseph, A. Tavakoli, and I. Stoica. A policy-aware switching layer for data centers. In *SIGCOMM*, 2008.
- [25] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *TOCS*, 18:263–297, 2000.
- [26] A. Li, X. Yang, and S. K. M. Zhang. Cloudemp: Comparing public cloud providers. In *IMC '10*, Melbourne, Australia, 2010.
- [27] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.
- [28] X. Meng, V. Pappas, and L. Zhang. Improving the scalability of data center networks with traffic-aware virtual machine placement. In *INFOCOM*, 2010.
- [29] L. Popa, M. Yu, S. Y. Ko, S. Ratnasamy, and I. Stoica. CloudPolice: taking access control out of the network. In *HotNets '10*.
- [30] C. Raiciu, S. Barre, C. Plunke, A. Greenhalgh, D. Wischik, and M. Handley. Improving datacenter performance and robustness with multipath tcp. In *SIGCOMM*, 2011.
- [31] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *CCS*, 2009.
- [32] V. Sekar, S. Ratnasamy, M. K. Reiter, N. Egi, and G. Shi. The middlebox manifesto: enabling innovation in middlebox deployment. In *HotNets*, 2011.
- [33] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes. Cloudscale: elastic resource scaling for multi-tenant cloud systems. In *SoCC*, 2011.
- [34] K. V. Vishwanath and N. Nagappan. Characterizing cloud computing hardware reliability. In *SoCC*, 2010.
- [35] G. Wang, D. G. Andersen, M. Kaminsky, M. Kozuch, T. S. E. Ng, K. Papagiannaki, and M. Ryan. c-Through: Part-time optics in data centers. In *Proc. ACM SIGCOMM*, New Delhi, India, Aug. 2010.
- [36] G. Wang and T. S. E. Ng. The impact of virtualization on network performance in Amazon EC2 data center. In *INFOCOM*, 2010.
- [37] T. Wood, K. K. Ramakrishnan, P. Shenoy, and J. van der Merwe. CloudNet: dynamic pooling of cloud resources by live WAN migration of virtual machines. In *VEE '11*.
- [38] M. Yu, J. Rexford, M. J. Freedman, and J. Wang. Scalable flow-based networking with difane. In *SIGCOMM*, 2010.