ProActive Routing in Scalable Data Centers with PARIS

Dushyant Arora

Master's Thesis

Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Master of Science in Engineering

Recommended for Acceptance
by the Department of Computer Science
Princeton University

Adviser: Jennifer Rexford

June 2013

© Copyright by Dushyant Arora, 2013.

All rights reserved.

Abstract

Modern data centers must meet many challenges and expectations: (a) They must scale to a large number of servers, while offering high bisection bandwidth and flexible placement of virtual machines. (b) They must allow tenants to specify network policies and realize them by forwarding traffic through the desired sequences of middleboxes. (c) They must allow tenants to bring their own IP address space to the cloud to ease transition of enterprise applications and middlebox configuration. The traditional approach of connecting layer-two pods through a layer-three core constrains VM placement. More recent "flat" designs are more flexible but have scalability limitations due to flooding/broadcasting or querying directories of VM locations. Rather than reactively learn VM locations, our PARIS architecture has a controller that *pre-positions* IP forwarding entries in the switches. Switches within a pod have complete information about the VMs beneath them, while each core switch maintains complete forwarding state for part of the address space. PARIS offers network designers the flexibility to choose a topology that meets their latency and bandwidth requirements. PARIS also allows tenants to bring their own IP address space and utilizes lightweight virtualization using Linux namespaces to offer middlebox service in a manner that truly reflects the pay-as-you go model of cloud computing. It utilizes MPLS label forwarding and aggregation in the network core and uses source routing at the network edge to realize middlebox policies specified by tenants. Finally, we evaluate our PARIS prototype built using Openflow-compliant switches and NOX controller. Using PARIS we can build a data center network that can support up to 500K VMs.

Acknowledgements

I would like to thank the Department of Computer Science for giving me this amazing opportunity to learn at such a great place under the guidance of brilliant researchers and scholars. I would like to thank my parents for their love and encouragement. I would also like to thank my advisor, Prof. Jennifer Rexford, for her constant guidance and support.

I would like to thank Eder Leão Fernandes for writing the only OpenFlow 1.3 implementation out there and for the quick bug fixes. This thesis would not have been possible without his help. I would also like to thank members of Cabernet Research Group especially Theophilus Benson and Srinivas Narayana for great discussions and insightful comments.

Contents

Abstract iii

Acknowledgements iv

List of Tables vi

List of Figures vii

1 Introduction 1

1.1 Related Work 4

1.2 ProActive Routing In Scalable DCs 6

2 Architectural Principles 7

3 PARIS Architecture 8

3.1 No-Stretch PARIS 9

3.2 High-Bandwidth PARIS 11

3.2.1 Valiant Load Balancing in Core Layer 12

3.3 Generalized Core-Layer Topologies 13

3.4 Fine-Grained Rules for Popular VMs 14

3.5 Elements and Dynamics 14

3.5.1 Network Elements 14

3.5.2 External Traffic 15

3.5.3 Network Dynamics 15

4 Multi-tenancy 16

5 Middlebox support 18

5.1 Addressing and Forwarding 20

5.2 Handling Middlebox Failure 23

5.3 OpenFlow Standard Modifications 24

5.4 Discussion 25

6 Evaluation 25

7 Conclusion 28

List of Tables

1 Comparison of recent data center network architectures. 4

2 Run time analysis of MB_{find}. 28

List of Figures

1 Common data center hierarchical topology. 3
2 No-Stretch topology with core-layer switches aggregating /16 prefixes. 8
3 Forwarding information stored in Aggregation Switch A1 of Figure 2. 10
4 High-Bandwidth topology with a full mesh in the core layer. 11
5 Middleboxes attached to network switches in an off-path configuration. 20
6 Flow entries stored inside a firewall's flow table. 22
7 CDF of sender bandwidth for No-Stretch and High-Bandwidth PARIS. 24
8 CCDF of sender bandwidth for No-Stretch and High-Bandwidth PARIS. 25
9 Scalability of No-Stretch PARIS. 26
10 Scalability of High-Bandwidth PARIS. 28

1 Introduction

Recent years have seen rapid growth in migration of enterprise applications to public clouds and this trend will continue to rise in the coming years [1]. Similar trends have been observed for private clouds as well [2]. This is driving creation of large data centers with thousands of switches that interconnect tens of thousands of servers running hundreds of thousands of Virtual Machines (VMs) [3, 4]. Operating at this large scale puts tremendous pressure on the design of network topologies, routing architectures, and addressing schemes. With this knowledge in mind, we come up with the following set of requirements that a data center network must satisfy:

1. **Scalability:** This is the first and most important requirement. Data center networks should be able to support hundreds of thousands of virtual machines and thousands of tenants.

2. **Virtual machine migration:** The data center network should not put any restrictions on host mobility. Any host/VM should be allowed to migrate to any physical server within the data center. Seamless host mobility helps lower power consumption by dynamically consolidating VMs on fewer servers and powering off underutilized servers. Apart from this, virtual machine migration also helps in dealing with hardware failures, malicious attacks, and system upgrades.

3. **Multi-pathing:** Big Data and High Performance Computing applications have led to the creation of low oversubscription, high-capacity network interconnects like Clos [5], Fat-tree [6] etc. The data center network should be able to leverage these high-capacity interconnects by spreading traffic across the redundant links in these topologies and achieve high bisection bandwidth.

4. **Easy manageability:** Network management must be automated for scalability. A data center network has a large quantity of network switches. Maintenance of switches is costly and configuring them is an error-prone task [7].

5. **Low cost:** Network architects must make use of off-the-shelf commodity components for building the network to keep costs low.

6. **Middlebox support:** Middleboxes are network appliances that provide services other than packet forwarding. Common examples of these appliances are Firewalls, Load Balancers, Network Address Translators, Intrusion Detection Systems etc. Enterprises (small and large) make use of these middleboxes in their networks and should continue to be able to use these

services when they migrate to the cloud. Cloud providers provide load balancing and limited packet filtering services today. However, none provide the flexibility to tenants to specify their network policies eg. forward all traffic going to destination port 80 through a firewall followed by a load balancer.

7. **Multi-tenancy:** There should be no restriction on a tenant's choice of IP address. This lowers the barrier for migrating legacy applications to the cloud. This also minimizes changes required in middlebox configuration, which might have hardcoded IP addresses. Bring-your-own-IP-address (BYOIP) is already supported by public clouds (eg. Amazon Virtual Private Cloud [8], Windows Azure Virtual Network [9]). Private clouds don't need to support multi-tenancy but they must satisfy the remaining requirements.

Traditional approaches of building layer-2 Ethernet switch networks or connecting islands of Ethernet LANs using layer-3 switches, fail to achieve these data center goals.

Layer-2 networks built using Ethernet switches are not scalable because they use flooding-based Source MAC Learning which does not scale beyond a couple of hundred hosts [10]. We can limit broadcasts in layer-2 networks by using VLANs but they are limited (4094) and require careful configuration. Another fundamental limitation of layer-2 networks is that they do forwarding on flat MAC addresses. As a side effect of Source MAC Learning, each Ethernet switch may have to store forwarding information for all hosts in the network. Typical commodity data center switches can store ∼64K MAC entries. Thus, the layer-2 table size of the switches puts an upper bound on the size of the network. Layer-3 networks on the other hand are easier to scale. Figure 1 shows a common data center topology with layer-2 switches in the edge layer and layer-3 switches in the aggregation and core layer. The data center address space is divided into subnets and each edge layer switch stores forwarding information for hosts belonging to a subnet. The aggregation switches store forwarding information for all subnets within its pod and aggregate the subnets before advertising them to the core layer. Switches in other pods store these aggregated IP prefixes and hence, the amount of information stored inside switch forwarding information base reduces.

Virtual machine migration requires layer-2 adjacency; since migrating VMs should not change their IP addresses as doing so will break their pre-existing TCP connections. Unlike MAC addresses, IP addresses are topologically significant. This restricts mobility of VMs to within a subnet in layer-3 networks, while layer-2 networks provide seamless host mobility. Seamless host mobility is possible at layer-3 as well [11] but it requires modification of end-host application.

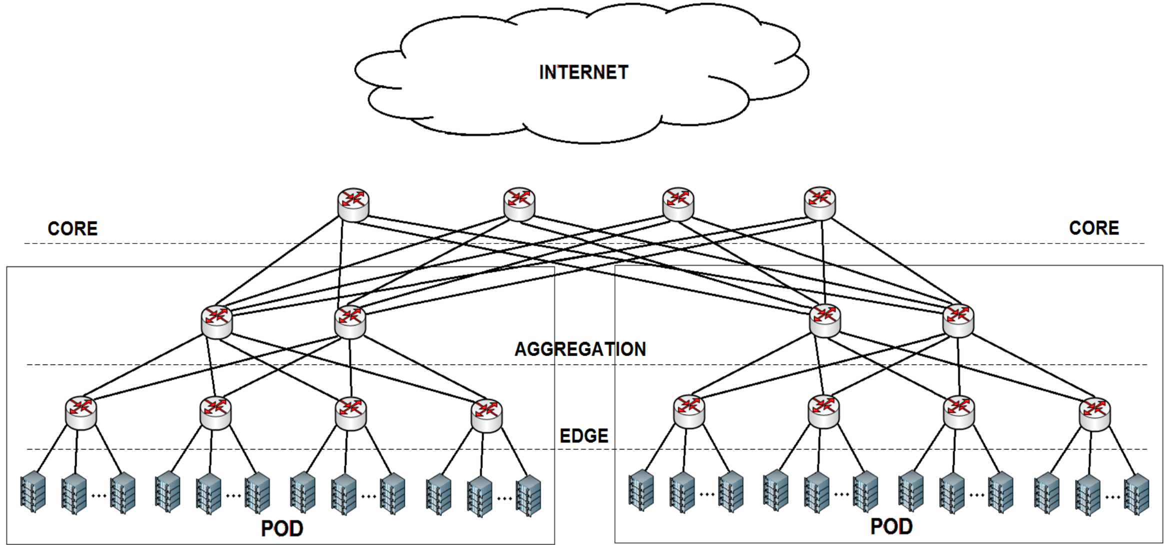

Figure 1: Common data center hierarchical topology.

Multi-pathing can be achieved in layer-3 networks through Equal-Cost Multi-path (ECMP [12]) routing which load-balances traffic over multiple paths of equal cost. Routing protocols like Open Shortest Path First (OSPF) and Intermediate System to Intermediate System (ISIS) explicitly allow ECMP routing. Data center switches supporting 32-way and 64-way ECMP are common these days. However, in layer-2 networks multi-pathing is a problem because Spanning Tree Protocol (STP) limits available bandwidth by disabling links, to avoid forwarding loops. STP also puts a lot of load on the root switch which becomes a single point of congestion and failure. This necessitates deployment of a high port-density and high bandwidth switch as the root switch, which increases the cost of network deployment.

Layer-3 networks require significant configuration effort which is error-prone [13, 14]. For each layer-3 switch, network operators have to configure its interfaces, configure the routing protocol, and configure route summarization on the switches. Also, since the network size is big, operators typically partition it into routing areas for scalability. This further complicates configuration. Finally, network administrators have to partition the data center address space into subnets and configure switch interfaces and DHCP servers with this subnet information. For layer-2 networks, the switch parameters must be configured properly so that STP chooses the desired high capacity switch as the root. Besides this, Ethernet switches require zero-configuration and provide plug-and-play functionality.

Architecture	Scalability	Host Mobility	Multi-pathing	Manageability	Low cost
Ethernet	×	✓	×	✓	✓
IP with Ethernet	✓	∼	✓	×	✓
TRILL	×	✓	✓	✓	✓
SPAIN	×	✓	✓	✓	✓
PAST	×	✓	✓	✓	✓
VL2	∼	✓	✓	×	✓
PortLand	∼	✓	✓	✓	✓
SEATTLE	×	✓	✓	×	×
No-Stretch PARIS	✓	✓	∼	✓	✓
High-BW PARIS	✓	✓	✓	✓	✓

Table 1: Comparison of recent data center network architectures.

Multi-tenancy can be easily supported in layer-2 networks, unless the tenant VMs have the same MAC address. Supporting multi-tenancy in layer-3 networks requires tunneling/encapsulation, since two tenant VMs can have the same IP address. Middlebox policies can be supported in an ad-hoc fashion in both networks. There are two ways we can do this—Tunneling [15] packets from one middlebox to another or by installing policy rules on all switches [16]. The latter approach is not scalable, since it consumes a lot of space on switch TCAMs, which are limited in size. The former approach is scalable but inflexible.

1.1 Related Work

There has been a lot of work in this problem space but none of the existing solutions provide all the desired properties. Table 1 compares recent data center network architectures with PARIS.

TRILL [17], SPAIN [18], and PAST [19] solve the limitations of the spanning tree protocol in layer-2 networks and achieve high aggregate bandwidth by spreading traffic across the links of the network topology. Since they are Ethernet-based solutions, they are able to support seamless host mobility and maintain plug-and-play functionality as well. However, since forwarding is still done on flat MAC addresses, the size of MAC table becomes a bottleneck and limits scalability.

TRILL runs link-state routing protocol (ISIS) in the network between TRILL Switches or RBridges (Routing Bridges) for dissemination of end-host and switch reachability information. Core RBridges need to know the MAC addresses of only the Edge RBridges, however, Edge RBridges learn MAC address forwarding information for all hosts in the network. Since Rbridges run link-state routing protocol among themselves, traffic through the core can take advantage of multi-pathing. RBridges still rely on flooding for unknown destination Ethernet frames and flood broadcast traffic (ARP, DHCP).

SPAIN achieves multi-pathing by calculating multiple disjoint spanning trees (each tree is assigned a VLAN ID) and spreading traffic across them through an end host "driver". However, switches now have to store multiple table entries (for each (VLAN ID, destination MAC) pair) and resort to flooding in case MAC table overflows. PAST builds a spanning tree for each host in the network which is more scalable but it still requires one Ethernet table entry per routable address in each switch. Also, it does multiple costly spanning tree computations each time a link/switch fails.

Another approach, taken by VL2 [20], NVP [21] and PortLand [22], is to use indirection to separate a host's location from its identity. Each host is given both a location address and an identity address. VL2 uses layer-3 addresses — topologically significant Locator Address (LA) and flat Application Address (AA) while PortLand uses layer-2 addresses — topologically significant Pseudo MAC (PMAC) address and flat Actual MAC (AMAC) address. The topologically significant address provides scalability and multi-pathing, while the flat address provides host mobility. Both schemes use indirection to translate between these addresses and this translation is cached at the end host. However, this location-identity split scheme has a few disadvantages. The scheme relies on directory servers that store mapping from a VM's address to its location. These directories introduce scalability challenges of their own, requiring many servers to handle the load of queries and updates. In addition, host/edge switches incur delay to query the directory, and overhead to cache the results and encapsulate packets, particularly for the high volume of traffic entering the data center from the Internet.

Monsoon [23], a pre-cursor to VL2, works at layer-2 and uses MAC-in-MAC tunneling instead of IP-in-IP encapsulation. Both VL2 and Monsoon use *Valiant Load Balancing* [24], a technique we leverage in our solution as well. Similar to PortLand, MOOSE [25] also proposes making MAC addresses hierarchical but it does not talk about multi-pathing.

SEATTLE [26] proposes using consistent hashing for scaling Ethernet by distributing directory information over the switches and directing traffic through an intermediate switch on a cache miss at the ingress switch. However, reactive caching can lead to large forwarding tables, unless the traffic matrix is sparse—a reasonable assumption in enterprise networks but not necessarily in data centers. SEATTLE cannot be implmented on commodity data center switches, since it requires switch modifications. Manageability is also an issue, since all switches need to be configured to run OSPF among themselves.

VL2 does not talk about how to achieve multi-tenancy and none of the above approaches talk about middlebox support.

1.2 ProActive Routing In Scalable DCs

In this paper, we present PARIS, a more scalable way to provide "one big virtual switch" in both public and private clouds.

We saw that networks built using flat MAC addresses provide seamless host mobility and require zero configuration but are not scalable and cannot support multi-pathing. Networks built using hierarchical IP addresses provide scalability and multi-pathing but they do so at the cost of host mobility and manageability. A natural question to ask is whether we can combine the best of both these approaches. PortLand [22] and MOOSE [25] explored one point in this design space by making *MAC addresses hierarchical*. In this work, we take the alternative route by building a network based on *flat IP addresses* to satisfy all our design requirements.

Since we treat IP addresses as flat addresses, running distributed routing protocols among the switches will not be scalable. Our design has a controller that *pre-positions forwarding state* in the network, based on knowledge of each VM's address and location. We leverage Software Defined Networking (SDN) enabled by OpenFlow [27] for this purpose. SDN separates switch control plane from its data plane, giving us control over what goes inside switch flow tables. Rather than encapsulate packets at the network edge, the bottom few levels of switches store a forwarding entry for *all VMs beneath them*. While clearly not scalable to the entire network, commodity switches have enough forwarding-table space for the IP addresses of all VMs in the same pod. To avoid a state explosion further up the hierarchy, the controller *partitions the forwarding state across the core switches*, so each core switch maintains fine-grained forwarding state for a portion of the data center's IP address space. However, partitioning the forwarding state across the core switches may lead to lower path diversity. Fortunately, this challenge is surmountable through careful design of the core-layer topology and by maintaining fine-grained forwarding entries for popular destinations.

The main contributions of this work are as follows:

- We come up with two data center forwarding schemes—No-Stretch PARIS and High-Bandwidth PARIS which are scalable, support host mobility and easy manageability, and provide good bisection bandwidth through multi-pathing.
- We implement multi-tenancy and middlebox support for both these schemes.
- We evaluate our PARIS prototype built using NOX [28] OpenFlow controller and emulated OpenFlow switches over Mininet-HiFi.

The next section illustrates the main architectural principles underlying our design. We begin by looking at the first five data center network requirements and describe forwarding schemes (No-Stretch PARIS and High-Bandwidth PARIS) that satisfy them in Section 3. Section 4 extends our design to public clouds, where we can have multiple tenants; each with their own IP subnets. Section 5 extends PARIS to provide support for middlebox policies. We present the evaluations in Section 6 and Section 7 concludes the paper.

2 Architectural Principles

In rethinking how to design scalable data-center networks, we identify four main principles:

Flat layer-three network: Having the entire data center form one IP subnet simplifies host and DHCP configuration, and enables seamless VM migration. However, forwarding on MAC addresses introduces scalability challenges, since the address space is large and flat—forcing the use of broadcasting/flooding which has the side effect of each switch learning location of all hosts in the network. In contrast, IP addresses are easily aggregated, with switches forwarding traffic based on the longest-matching prefix.

Proactive installation of forwarding state: Installing forwarding-table entries before the traffic arrives reduces packet latency and avoids the overhead of learning the information reactively. However, injecting flat addresses into routing protocols (e.g., OSPF or IS-IS) leads to large forwarding tables in every switch. Instead, a logically-centralized controller can pre-position the necessary forwarding-table entries in each switch, based on a network-wide view of the topology and the locations and addresses of VMs. This enables much smaller tables.

Complete forwarding information within a pod: Storing a forwarding-table entry for every VM in every switch would not scale. Yet, a switch could easily store the forwarding information for all VMs in the same pod. Today's low-end switches have enough space for storing forwarding information for thousands of servers. OpenFlow-enabled switches have separate tables for wildcard and exact-match entries [19]. Wildcard entries are stored in a TCAM, which is smaller in size (∼4K-24K), while exact-match entries are stored in conventional memory like SRAM, which is much larger (∼16K-64K layer-3 table). Assuming we have 48 port switches at the edge layer, each connected to a physical server hosting 64 VMs, a quick back-of-the-envelope calculation shows that each pod aggregation switch in Figure 1 will store ∼13K entries—easily storable in layer-3 table. Future switches will have larger tables, to accommodate multi-core servers hosting even more VMs. Storing

complete information enables short paths and good path diversity within a pod, and default routes up the hierarchy to the rest of the network.

Partitioned forwarding state in the core: Moving up the hierarchical topology, the number of VMs lying "below" a switch grows exponentially. We can no longer store complete forwarding information inside the core-layer switch. So, we divide the data center address space and store full forwarding state for a portion of the IP address space in each core switch. For example, we can divide a /14 address space into four /16 *Virtual Prefixes* [29] and a core switch with a layer-3 table size of 64K can store forwarding information for an entire /16 virtual prefix. Since we treat IP as a flat address within a pod, VMs with IP address within this prefix may be spread across multiple pods.

3 PARIS Architecture

In this section, we first discuss how PARIS achieves scalability and multi-pathing by proposing new forwarding schemes and core-layer topology. Next, we describe how to reduce stretch and increase path diversity in PARIS by having fine-grained forwarding entries for popular destinations in core switches. Finally, we look at the PARIS architecture components and network dynamics. We initially assume a private cloud setting where all VMs belong to the same institution and have unique IP addresses. Later, we extend the forwarding schemes for multi-tenant scenario in Section 4.

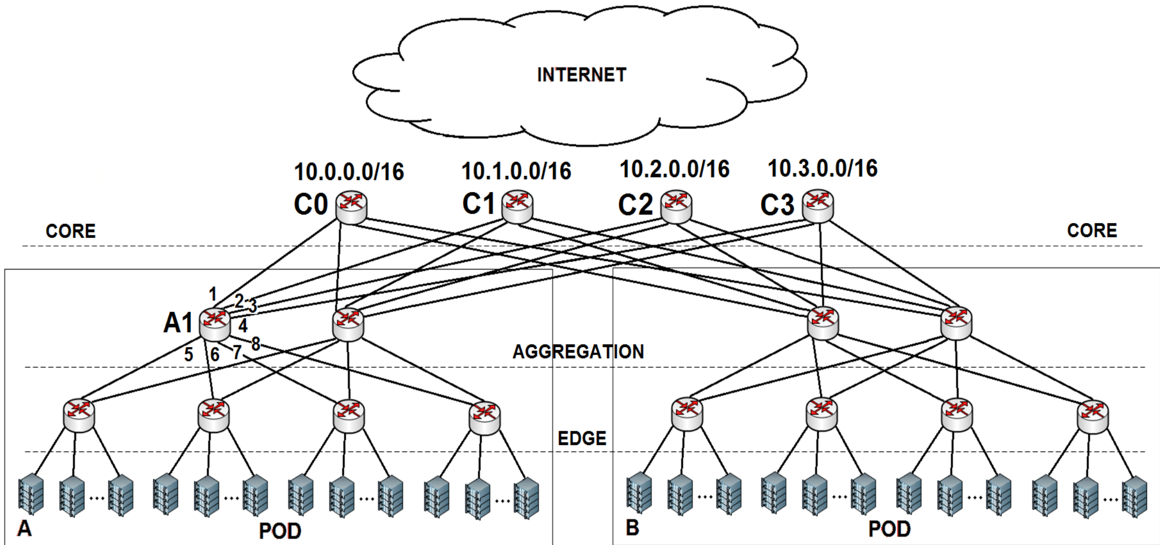

Figure 2: No-Stretch topology with core-layer switches aggregating /16 prefixes.

3.1 No-Stretch PARIS

Based on the architecture principles we described in Section 2, we come with No-Stretch PARIS forwarding scheme as shown in Figure 2. In No-Stretch PARIS packets always take the shortest path from source VM to destination VM, hence the name. Each edge switch stores reachability information of all directly-connected VMs, i.e., all VMs beneath it. Similarly, each aggregation switch stores reachability information of all hosts in its pod. The core-layer switches store reachability information for a portion of the data center address space. Each *Virtual Prefix* (VP) has an *Appointed Prefix Switch* (APS) in the core layer, responsible for storing reachability information for all IP addresses that lie within that virtual prefix. Figure 2 shows a data center network with 10.0.0.0/14 host address space, where each switch in the core layer is aggregating a /16 virtual prefix. We say, for example, that C1 is an APS for virtual prefix 10.1.0.0/16 and stores reachability information for all VMs with IP address that lies within the virtual prefix 10.1.0.0/16. The actual virtual prefix length depends on the layer-3 table size of the core switches. The data center address space size together with layer-3 table size determines the number of core-layer switches.

Since each core switch stores reachability information to only a subset of the address space, each aggregation switch must be connected to every core switch in a complete bipartite graph configuration in order to reach all the hosts in the network. In order to reach destination VMs in other pods, all aggregation switches store low-priority flow entries which match on virtual prefixes. According to OpenFlow specification [30], packets match flow entries in priority order, i.e., from the highest priority to the lowest priority entry with the first entry which matches being used for forwarding. Figure 3 shows flow entries installed on a pod aggregation switch in No-Stretch PARIS. Traffic to a destination VM attached to a different edge switch in the same pod, matches a default priority exact-match entry (non-shaded), instead of a low-priority virtual prefix match entry (shaded).

Parameterized Construction: Assume a physical server can host V VMs, and let k be the port density of all switches in Figure 2. Since each core-layer switch is connected to all the aggregation-layer switches, we can have a maximum of $k - 4$ core switches, k aggregation switches and $2k$ edge switches for the pod design of Figure 2. The maximum number of VMs supported by this topology will be $2k(k - 2) \times V$. Using this formula, if we have 64 port switches in our data center network we can support a maximum of $\sim$500K VMs.

We try to take advantage of equal-cost paths to spread traffic, wherever possible. In Figure 2, traffic from an edge switch going to an aggregation or core-layer switch has two equal-cost next

DESTINATION IP	OUT PORT
10.1.0.2	5
10.0.0.4	7
10.0.0.58	8
10.2.1.5	6
10.3.5.68	6
10.1.1.8	7
. . . .	
10.0.0.0/16	1
10.1.0.0/16	2
10.2.0.0/16	3
10.3.0.0/16	4

Figure 3: Forwarding information stored in Aggregation Switch A1 of Figure 2.

hops. OpenFlow specification starting from v1.1 allows spreading of traffic across multiple ports of a switch through *select* group type entry in switch group table. Commodity switches have a group table size of ∼1K [19], enough for our purpose. The implementation of traffic splitting algorithm is switch dependent and can be done on a per-flow or per-packet granularity. So, in order to spread traffic across the two equal-cost paths we install on each edge switch in Figure 2, a single low-priority flow entry which spreads traffic across the two ports connecting it to its pod aggregation switches. This low-priority entry is matched in case no default-priority entry in edge switch matches, i.e., if the destination VM is not directly connected to the edge switch.

Traffic between two VMs A and B (10.2.0.24), located in different pods, matches the low-priority flow entry on A's edge switch and will be forwarded to one of its pod aggregation switches. Since B is located in a different pod, aggregation switches in A's pod do not have forwarding information for it. Instead, the packet will match the low-priority virtual prefix match entry in A's pod aggregation switch. In this example, it will be forwarded to the APS aggregating virtual prefix 10.2.0.0/16 in the core layer—C2. The APS can forward the traffic to any of the two aggregation switches in B's pod. The controller randomly chooses one of them during network bootstrap and installs a forwarding entry for it in C2. Once the packet reaches the destination pod aggregation switch, it will have the necessary forwarding information for forwarding the traffic to B's edge switch, which ultimately forwards the traffic to B.

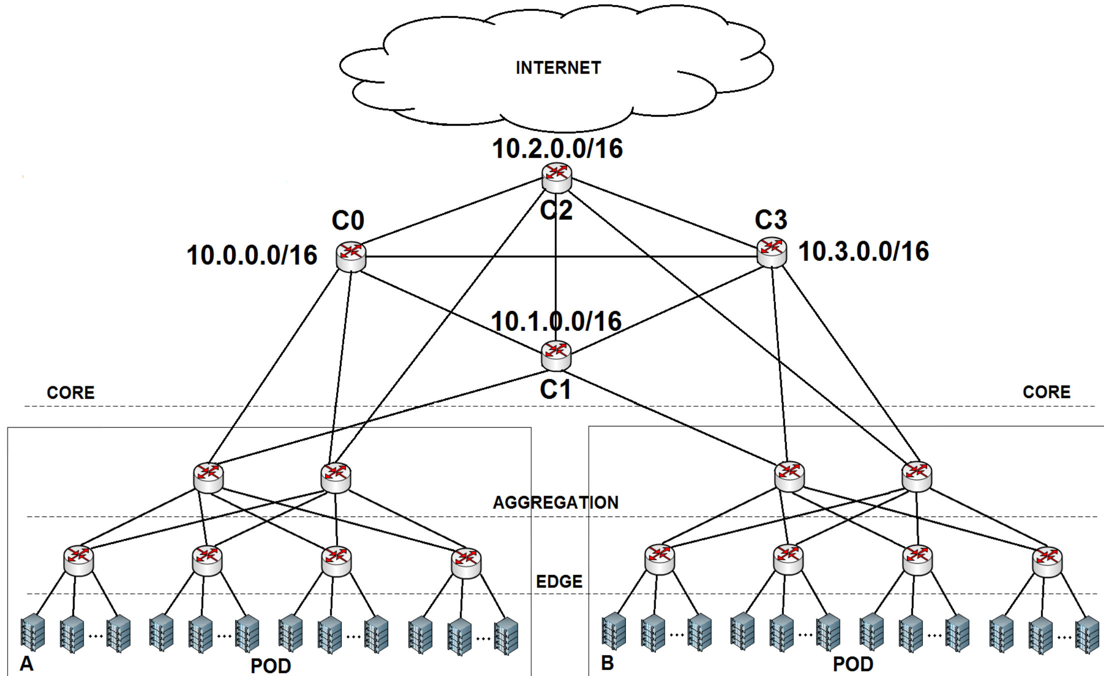

Figure 4: High-Bandwidth topology with a full mesh in the core layer.

While sending northbound traffic, No-Stretch PARIS takes advantage of equal-cost paths that exist within a pod but it does not take advantage of multiple equal-cost paths that exist between pods. Instead, it does coarse-grained traffic division at virtual prefix scale. We overcome this limitation by modifying the core-layer topology in the next section.

3.2 High-Bandwidth PARIS

Data center traffic measurement studies have shown that majority of traffic in a data center stays within a data center (i.e. "east-west" traffic) and this traffic will continue to rise in the coming years [31]. It is incumbent on a data center network to support high inter-host communication bandwidth through multi-pathing. We saw in No-Stretch PARIS that partitioning of forwarding state across core switches leads to lower path diversity. Also, the network size that No-Stretch PARIS can support is limited by the port density of the switches. Fortunately, careful design of the core-layer topology can overcome these limitations. In this section, we propose High-Bandwidth PARIS that provides higher bisection bandwidth and makes PARIS more scalable.

Instead of having disjoint switches in the core layer, we connect them to form a full mesh (see Figure 4). We choose a full-mesh topology because it has low diameter and multiple paths for fault tolerance and load balancing.

Parameterized Construction: Let's say we wish to construct a data center with 2^h VMs using k-port switches, with pod design as shown in Figure 4. We assume core-layer switches have a routing table size of 2^r. For connecting these VMs, we will need $\lceil \frac{2^h}{(k-2)\times V} \rceil$ edge switches, $\lceil \frac{2^{h-1}}{(k-2)\times V} \rceil$ aggregation switches and 2^{h-r} core switches, where V is the number of VMs each physical server attached to an edge switch can host. Each core-layer switch will be connected to $(2^{h-r}-1)$ other core switches. Each aggregation switch is connected to $\min(k-4,\, E)$ randomly-selected core switches where,

$$E = \left\lfloor \frac{(k - 2^{h-r} + 1) \times (k-2) \times 2V}{2^r} \right\rfloor \quad (1)$$

Since aggregation and core switches no longer need to be connected in a bipartite graph configuration, we can support more hosts in High-Bandwidth PARIS.

The aggregation-layer switches no longer have low-priority virtual prefix match entries shown in Figure 3. Instead, they have a low-priority flow entry which spreads traffic across all the connected core switches, in case none of the default entries match. This provides us more multi-pathing compared to No-Stretch PARIS. As before, each core-layer switch is an APS for a virtual prefix. In order to reach all the hosts in the data center, each APS has low-priority virtual prefix match entries for all virtual prefixes being aggregated by other APSs in the core layer.

Traffic between two VMs A and B (10.2.0.24), located in different pods, flows through a maximum of three core switches (two hops). The source pod aggregation switch hashes the five tuple of the packet and forwards it to one of the core-layer switches (say C0). The ingress core switch looks at the destination IP address of the packet and forwards it to the appropriate APS (C2). The APS then tunnels the traffic through the core layer to the destination pod aggregation switch, if it is not directly connected to it. Tunneling rules have higher priority. For tunneling the packets, we can use MPLS or VLAN headers. In this example, no tunneling is needed since C2 is directly connected to the destination pod aggregation switch. Traffic is directly forwarded to the aggregation switch in B's pod. In case C2 is connected to both aggregation switches in B's pod, the controller randomly chooses one of them during network bootstrap and installs a forwarding entry for it in C2.

3.2.1 Valiant Load Balancing in Core Layer

Measurement studies have shown that traffic pattern inside a data center is highly volatile [20]. *Valiant Load Balancing* (VLB) is a randomization technique for handling traffic variations under the hose model [32]. It facilitates efficient link utilization by bouncing the traffic off a random switch

before forwarding it to its destination. We can further increase multi-pathing in High-Bandwidth PARIS by implementing VLB in the core layer.

In a full-mesh network of N nodes, where a node can send traffic at a maximum rate r to another node, the link capacity of mesh links required to support any traffic matrix using VLB is only $r(\frac{2}{N})$ [33]. For implementing VLB in our topology, the link capacity required between any pair of core switches will be $r(\frac{4}{N})$, where in the first phase a packet entering the core layer is forwarded to an APS after bouncing it off a randomly selected core switch, and in the second phase it is forwarded to an egress core switch from the APS after again bouncing it off a random core switch.

By implementing VLB, we can reduce the bandwidth requirement of the internal links in the core layer, and also support arbitrary traffic matrices. However, a packet may now travel four hops instead of two in the core layer, i.e., we trade-off stretch for throughput. This is a reasonable trade-off in data centers since they have very low network latency.

In conclusion, High-Bandwidth PARIS is more scalable, has higher path diversity and can provide higher bisection bandwidth compared to No-Stretch PARIS. We verify these properties through evaluations in Section 6.

3.3 Generalized Core-Layer Topologies

So far we have seen No-Stretch and High-Bandwidth variants of PARIS, with disjoint and fully-connected core-layer topologies. These are two special examples of core-layer topologies with different path diversity, stretch, and fault-tolerance properties. We will now look at other graph configurations for connecting switches in the core layer.

If we have a sparse graph in the core layer with a small number of internal edges, we will have more spare ports for supporting multi-pathing between the aggregation and core-layer switches. But a sparse graph will have a larger diameter and lower path diversity within the core layer. We seek a core-layer topology with high connectivity for fault tolerance and load balancing, low diameter, and low vertex degree. These properties are satisfied by *Expander Graphs* which have $\mathrm{O}(\log n)$ diameter, where n is the number of vertices in the graph. There are many constructions [34] for different families of expander graphs which have varying vertex degree and diameter. Some of them are LPS graph, Paley graph, Margulis construction for expander graphs, Hypercube graph and superconcentrators. The network designer can choose a suitable expander graph topology depending upon the latency, "east-west" bandwidth, and reliability needs of the data center.

3.4 Fine-Grained Rules for Popular VMs

Traffic to popular destinations may experience stretch in High-Bandwidth PARIS, and low bisection bandwidth in No-Stretch PARIS. To alleviate these problems, the OpenFlow-based controller can install fine-grained forwarding entries (/32) for popular destinations in the core-layer switches.

In No-Stretch PARIS, if the controller installs fine-grained forwarding entries for popular destinations on all core-layer switches, the aggregation switches can install high priority flow entries for spreading traffic across the core layer and achieve higher bisection bandwidth for these destinations. For High-Bandwidth PARIS, installing individual destination rules on all the core-layer switches, instead of aggregating them at APSs, ensures that all the traffic to these destinations always takes the shortest path through the core layer.

Traffic measurements for identifying popular destinations incur little overhead. We can collect traffic statistics in the virtual switches inside the hypervisor or in the commodity switches, if they support it. OpenFlow-enabled switches keep traffic statistics for all flow entries, by default.

3.5 Elements and Dynamics

In this section, we discuss various architecture components of PARIS and how they interact with each other. Also, we describe how the network handles external traffic and copes with failure.

3.5.1 Network Elements

Controller/Fabric Manager: The controller has complete visibility into the network topology and knows the address and location of all VMs. Using this initial information the controller performs the following tasks: (i) tracking switch-level topology and host location, (ii) optimally placing and updating forwarding information in switches after startup/failure, and (iii) monitoring network traffic to perform traffic engineering. We use NOX [28] OpenFlow controller for this job in our evaluations.

Switches: We do not run any intra-domain routing protocol between the switches. In order to learn about topology changes, the switches must support neighbor discovery via protocols like LLDP and send notification events to the controller. OpenFlow-enabled commodity switches provide all the features we need to build No-Stretch and High-Bandwidth forwarding schemes.

Hosts: Hosts send a variety of broadcast traffic which needs to be managed in order to make the network scale and save precious bandwidth. We place each VM in its own /32 subnet with a default route to its aggregation switch, so that it no longer sends ARP broadcasts. In location-identity split schemes, the first packet of each new flow is sent to the controller for address-mapping. Measurement

studies have shown that more than 99% of flows in a data center network are latency-sensitive “mice” flows that last for tens of milliseconds [20]. The additional latency incurred for translation lookup is detrimental to these flows. Moreover, extra hardware is needed to create a scalable directory service. A network with 2 million VMs requires about 667 directory servers for handling address-mapping queries [35]. Unlike other approaches [22, 20, 26], we don’t need these directory servers.

Host DHCP messages are intercepted by the edge switches and forwarded to the controller, which can assign any unallocated IP address to any VM. This is a one-time lookup and it greatly simplifies host configuration. For handling multicast traffic, we can create virtual prefixes for multicast addresses and aggregate forwarding information for multicast groups in the core layer. The switches forward IGMP join messages to the controller which installs relevant forwarding information on the core, aggregation, and edge switches to ensure delivery of multicast traffic.

3.5.2 External Traffic

Architectures that use packet encapsulation or header rewriting to separate host location and identity [23, 22, 21] must perform these action on a large volume of diverse traffic arriving from the Internet. For example, Nicira’s network-virtualization platform has special gateway servers for this purpose. VL2 [20] uses an externally visible Location Address for servers which need to be reachable from the Internet. So, it doesn’t need special gateway servers but it restricts the movement of these externally visible servers as a result. In contrast, PARIS does not extend or modify the packet header, greatly simplifying the handling of traffic entering the data center from the Internet.

For handling external traffic in No-Stretch PARIS, we need a border router which is attached to all the core-layer switches and which stores forwarding information for all the virtual prefixes. For High-Bandwidth PARIS, we do not need this border router as all the core-layer switches are connected to each other in a full-mesh topology.

3.5.3 Network Dynamics

Since the controller has a network-wide view, it plays a crucial role in coping with network dynamics.

Switch Dynamics: If an edge switch fails, the VMs attached to that edge switch become unreachable. Unless there are redundant edge switches, there is nothing that can be done to restore reachability. If one of the links connecting an edge switch to the aggregation-layer switches goes down or one of the aggregation switches goes down, the group entry in edge switch can detect that one of its ports is down and start forwarding traffic to its live port. If a core-layer switch fails, the virtual prefix being aggregated by it can be sub-divided into smaller sub-prefixes and stored on

other core-layer switches until a new core-layer switch comes up. This provides graceful degradation and load balancing properties to the architecture. In both No-Stretch and High-Bandwidth PARIS having redundant core switches can help increase fault tolerance and also provide multi-pathing benefits.

Link Dynamics: Since we have multiple paths between any pair of hosts, simple re-routing of traffic by the controller can restore reachability after a link failure. The High-Bandwidth topology is more resilient to link failures than No-Stretch topology.

Host Dynamics: When a VM migrates to a new pod, the controller installs forwarding information on the new edge and aggregation switches for reaching the migrated VM. It also updates the APS entry, so that the APS forwards the packets to the new pod aggregation switch. Finally, the controller deletes forwarding entries from old edge and aggregation switches. The controller can orchestrate VM migration or it can learn that the VM has moved through gratuitous ARP from the migrated VM.

4 Multi-tenancy

Our architecture so far was geared towards private clouds, where all VMs have unique IP addresses. However, it can be easily extended to provide multi-tenancy support, where a tenant can request any (possibly overlapping) IP address.

To uniquely identify each VM in the data center network, we allocate each tenant a unique *Tenant ID*. In PARIS, we use MPLS Label Value field for this purpose. MPLS is a "layer 2.5" protocol with a 20-bit field called Label Value. We allocate each tenant a unique Label Value starting from 16.[1] We tag all outgoing packets of a tenant VM with the appropriate Tenant ID and remove this tag from all incoming packets. MPLS Label Value along with IP address, now uniquely identifies a VM.

We choose MPLS Labels over VLAN Identifiers because using VLAN Identifiers as Tenant IDs would allow us to support only 4094 tenants. The number of tenants inside a public cloud can be large and MPLS Labels allow us to support over a million tenants. MPLS support was added to OpenFlow starting from v1.1.

The edge, aggregation, and core-layer switches continue to store the information that they were storing previously. The edge and aggregation switches store forwarding information for all VMs beneath them and the APSs aggregate virtual prefixes as before. The only difference is that these switches now match on both MPLS Label Value and destination IP address of an incoming packet.

[1] Label Values 0-15 are reserved.

OpenFlow specification as of v1.3 does not allow switches to match on both MPLS Label Value and destination IP address simultaneously. To get around this limitation, we make use of two flow tables—Flow Table 1 and Flow Table 2. OpenFlow specifications starting from v1.1 allows switches to have multiple flow tables. A flow entry which previously matched on the destination IP address of a VM and forwarded that packet out of a switch port, now matches on both the destination IP address and MPLS Label Value of the tenant that owns the VM. Packet processing will now be done as follows:

- In Flow Table 1, we have entries that match on valid MPLS Label Values. On a successful match we store the MPLS Label Value inside a Metadata register (introduced in OpenFlow Specification v1.2 [36]), pop-off the MPLS header, and send the packet down the switch pipeline to match on entries stored in Flow Table 2.
- Inside Flow Table 2, we store entries that match on both the Metadata register (which contains the MPLS Label Value) and the destination IP address of the packet. If there is a successful match, we push a MPLS header on the packet, set its MPLS Label Value to the value inside the Metadata register, and forward it out of the switch port.

We also need to somehow push a MPLS header (with Label Value = Tenant ID) on to a packet coming out of a tenant VM and pop this header off before forwarding it back to a tenant VM. We use the host virtual switch running within the hypervisor for performing this task. The flow table of a host virtual switch is stored in DRAM, which is both scalable and has low cost. This allows us to store millions of flows entries in the host virtual switch flow tables. Since the controller knows all the tenant-VM mappings, it installs flow entries inside host virtual switches. It installs entries for pushing a MPLS header and setting its MPLS Label Value as the VM Tenant ID, for all packets coming from a VM before forwarding them to the edge switch. The controller also installs flow entries for popping off the MPLS header of packets going to a VM, after matching on both its MPLS Label Value and destination IP address. Also, if the destination VM is directly connected to the source edge switch, the controller installs high priority flow entries inside the host virtual switch so that it forwards traffic to the destination VM without pushing and popping MPLS headers.

Since the switches now match on MPLS Label Value along with destination IP address, the number of forwarding rules stored inside each switch increases. Specifically, we now need switches with multiple flow tables and the number of flow entries stored inside each switch increases by the number of tenants. So, the virtual prefix size now not only depends on the routing table size but also on the number of tenants we wish to support in the data center network. Traffic in both No-Stretch

and High-Bandwidth PARIS is forwarded as before but now using the combination of MPLS Label Value and VM IP address to uniquely identify each VM.

5 Middlebox support

Middleboxes are layer 4-7 devices that inspect, filter, and modify network traffic. These services are highly valued by IT organizations and enterprises migrating applications to the cloud expect support for these services [37]. However, so far cloud providers have provided only limited support for them [38, 39]. In this section, we propose new addressing and forwarding schemes that extend PARIS to enable middlebox support for tenants. Tenants specify middlebox policies and configuration for each class of middlebox device they need, and the network infrastructure ensures that these policies are implemented correctly and efficiently at all times. We provide support for policies of the form: [Tenant ID, Predicate specifying packet fields that need to be matched] → Sequence of middleboxes. For example the following policy:

$$\text{[Tenant ID:16, dest_IP:10.1.0.17, dest_port:80]} \rightarrow \text{Firewall} \rightarrow \text{Load Balancer} \tag{2}$$

states that all packets from VMs belonging to tenant 16 having destination IP address 10.1.0.17 and destination port 80 should traverse a firewall followed by a load balancer.

In thinking about how to support middlebox policies in PARIS, we use the following design principles:

Smart Edge Simple Core [40]: OpenFlow started out by identifying a common set of matching headers and actions among vendor flow tables. The OpenFlow protocol allowed network operators to install entries in the switch data plane that exploited this common functionality. Thus, it was successful in making the network plane programmable and could be supported across many vendor switches. However, as host requirements are evolving, the OpenFlow standard is expanding the common set of features to support more services. This complicates the switch hardware and also slows down packet lookups, since the hardware now has to perform matching on larger number of bits. Further, since forwarding decision is made at each switch hop, things slow down further.

A much simpler way to implement the same functionality is to keep the core simple and let it do what it does best: forward packets through simple table lookups. We store the network intelligence in the programmable network edge. The flow entries in the network edge specify how packets should be forwarded and the network core simply does what it is told. We follow this philosophy in our implementation. The edge in our case is implemented in software switches inside hypervisors. An

added advantage of storing network policies in software is that we are no longer limited by the limited TCAM size of hardware switches.

Separate Policy from Reachability: We borrow this principle from [16]. Ad-hoc practices for ensuring middlebox traversal, like removing links to create choke points, are hard to configure and maintain. By taking middleboxes off the network path, we ensure efficiency and fault tolerance. No packet traverses unnecessary middleboxes and middlebox failure no longer leads to network partition. Since data centers have low network latency, a small increase in packet path length due to off-path middleboxes is acceptable.

Use Source Routing: Source routing helps us realize the previous two design principles. Another alternative [15] could be to store network policies inside switches directly attached to middleboxes and tunnel packets from one middlebox to another. This approach violates our first design principle and is inflexible. It is hard to provide strong consistency guarantees [41] in this approach, in case the middlebox policies are changed. Providing eventual consistency may violate correctness of policies and can lead to security vulnerabilities [16].

In case of source routing, changing policies is fast and simple. To modify a policy we simply change the flow entry installed on the host virtual switch in the hypervisor. All packets now either follow the old policy or the new policy.

We use MPLS label stack for implementing source routing in PARIS. The controller accepts policies from tenants and proactively installs entries on the host virtual switches to implement them.

Share Middleboxes: Most middleboxes in enterprise networks operate at moderate to low utilization [42]. This prompted us to allow sharing of middleboxes among tenants using a pay-per-use model which amortizes the Capex cost of hosting middleboxes for the cloud provider and lowers the Opex cost for tenants. This also allows small enterprises who cannot afford to install middleboxes in their network to leverage their services in the cloud.

Isolation and monitoring are prerequisites for allowing sharing of middleboxes. We provide isolation by running each tenant's middlebox instance in a Linux process and network namespace container. We can configure the amount of CPU fraction allocated to each tenant's instance and the core on which the instance runs. We allow monitoring of tenant's instance with the help of an OpenFlow virtual switch running inside the middlebox. The virtual switch allows us to poll for tenant usage statistics which can be used for billing and can also help in making scaling [43] and source routing decisions.

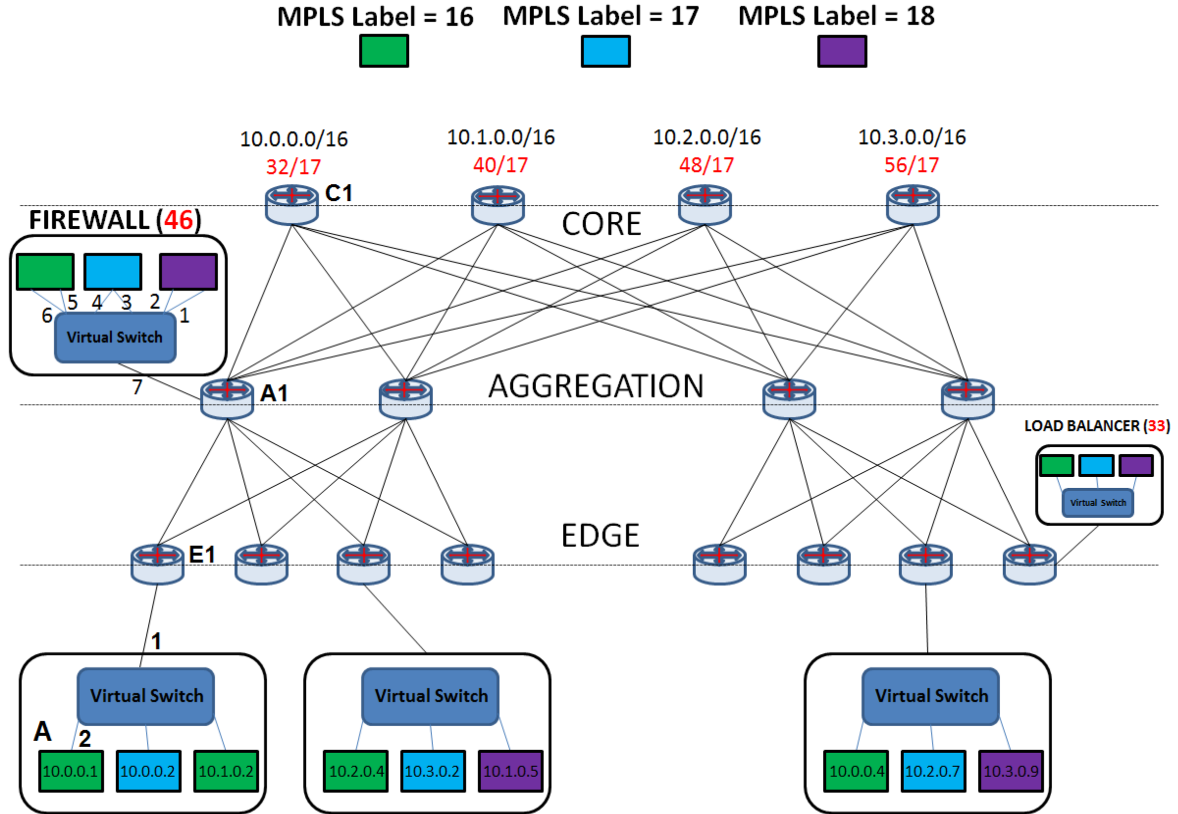

Figure 5: Middleboxes attached to network switches in an off-path configuration.

5.1 Addressing and Forwarding

Since our architecture uses source routing for implementing middlebox policies, we need a way to address the middleboxes inside the data center. A large enterprise today has thousands of middleboxes—at par with the number of switches in the network [44]. In our implementation, we make use of MPLS Label Value for addressing middleboxes by assigning each middlebox a unique Label Value. We used the MPLS Label Value field for supporting tenants too in Section 4. The MPLS Label Value address space supports over a million unique values. We can easily share this space among the tenants and the middleboxes.

Giving each middlebox a unique MPLS address has a few more advantages. It allows us to support transparent middleboxes like firewalls and caches. Also, since MPLS is a "layer-2.5" protocol, we don't need to make any modifications to enterprise middlebox configuration which might have hardcoded IP/MAC addresses. This lowers the barrier for migrating enterprise networks to the cloud.

In Section 2, we learnt the design principles for implementing scalable forwarding on flat addresses in a data center network. MPLS Label Values are flat addresses, so we apply the same principles here. Figure 5 shows how a middlebox is attached in an off-path configuration to switches in our data center topology. Edge switches store forwarding information (MPLS Label Value) for all middleboxes directly attached to them or attached to aggregation switches in their pod. Aggregation switches store forwarding information for all middleboxes directly attached to them or attached to any of the edge switches beneath them. Next, we create virtual prefixes called *MPLS Prefixes* for MPLS Label Value address space. Each MPLS prefix has an appointed switch in the core layer which stores forwarding information for all middleboxes with MPLS Label Value within this MPLS prefix. The controller calculates these prefixes during network bootstrap, making sure that the each core-layer switch is storing roughly the same number of MPLS Label Values. We do matching on MPLS Label Values in the Flow Table 1 of the hardware switches.

Now that we can address each middlebox in the data center network using MPLS Label Value and forward packets addressed to it, we use this capability to implement tenant middlebox policies. The controller accepts middlebox policies (Eq. 2) from a tenant and runs MB_{find} algorithm to find specific middlebox instances for each middlebox class specified in the policy. The MB_{find} algorithm runs Dijkstra's algorithm on a modified graph topology and has a run time complexity of $\mathrm{O}\big(Ek \log Vk\big)$, where E, V are the number of links and network appliances in the data center network and k is the number of middleboxes in the tenant policy. We run this algorithm for each host virtual switch attached to a VM belonging to the tenant, whose middlebox policy we are implementing.

For finding the sequence of middleboxes to use for implementing the tenant middlebox policy, we construct a modified graph from the original data center network topology with links connected in such a way that it forces Dijkstra's algorithm to find the shortest path that goes through the desired sequence of middleboxes. We can modify the topology to further optimize MB_{find} algorithm. We can remove links which don't have enough free bandwidth or remove overloaded middlebox instances from the graph. We use the host virtual switch as the source and the last middlebox in the policy as the destination for Dijkstra's algorithm. Now, we run the original Dijkstra's algorithm in this modified graph and find the shortest path between the host virtual switch and the destination middlebox which goes through the desired sequence of middlebox classes.

Once, we know the specific instances for each middlebox class in the policy, we install flow entries in the host virtual switch for incoming packets from the tenant VMs. The flow entries push a MPLS Label stack on all incoming packets which match the policy predicate, before forwarding them to the edge switch. This MPLS Label stack contains multiple MPLS headers, with MPLS Label Values set

FLOW ENTRY	OUT PORT
eth_type: 0x8847, MPLS_BOTTOM_LABEL: 16	5
eth_type: 0x8847, MPLS_BOTTOM_LABEL: 17	3
eth_type: 0x8847, MPLS_BOTTOM_LABEL: 18	1
in_port:2	7
in_port:4	7
in_port:6	7

Figure 6: Flow entries stored inside a firewall's flow table.

to addresses of specific middlebox instances, in accordance with the desired sequence of middlebox classes that the policy specifies. The Label Value of the bottommost MPLS header in this stack is set equal to the Tenant ID. The MPLS Label Stack is light-weight, since each MPLS header is only 4 bytes long. This allows us to support long policies efficiently. Also, since the policies are stored inside host virtual switches, flow table memory is not a constraint.

In our current prototype we have implemented support for only two middlebox classes: firewall and load balancer. However, extending our approach for providing support to myriad classes of middleboxes [45] should be easy. Each middlebox instance running on a server consists of an OpenFlow virtual switch and a Linux process and network namespace container for each of the tenants. The tenant middlebox configuration is installed inside this container. The virtual switch matches on the bottom-most MPLS header's Label Value to identify the correct Linux container to which the incoming packet must be forwarded.

A tenant container in a firewall has two virtual interfaces connected to the virtual switch (see Figure 5). Each container has a Linux bridge [46] running inside it, which forwards all incoming packets coming on one interface out of the other. We use ebtables [47] for bridge filtering. The configuration for ebtables is provided by the tenant.

A tenant container in a load balancer has one virtual interface connected to the middlebox virtual switch. For implementing a load balancer, we use Linux Virtual Server [48] (LVS). The tenant provides the load balancer configuration which contains the slave VMs for every tenant IP the load balancer is serving. The load balancer uses IP-in-IP tunneling for forwarding the packets to the slave servers in a round robin fashion. The slave servers reply back directly to the sender VM without going through the load balancer on the way back. The load balancer can also use source NAT [49] instead of tunneling in our architecture.

We now see an example of how forwarding will take place in the topology shown in Figure 5. The figure shows the implementation of middlebox support for No-Stretch PARIS but we have built a similar prototype for High-Bandwidth PARIS as well. For this example, let's say the tenant specifies the policy given in Eq. 2. The controller will run the MB_{find} algorithm and install the policy on all relevant virtual switches. Now, when VM A sends out a packet to destination 10.1.0.17:80, it will match the following entry on its server virtual switch: [in_port: 2, dest_IP: 10.1.0.17, dest_port:80] $\rightarrow$ [PUSH_MPLS(16), PUSH_MPLS(33), PUSH_MPLS(46), out_port:1]. According to the OpenFlow standard, switches match on the field value of the outermost header, by default. So, in this case the packet will now be forwarded to middlebox with MPLS Label Value 46. Edge switch E1 has forwarding information for middleboxes attached to all aggregation switches in its pod. It will forward the packet to A1. All switches store forwarding information for directly connected middleboxes. So, A1 will forward the packet to the firewall. Figure 6 shows all the flow entries installed on the firewall's virtual switch. The firewall's virtual switch will match on the bottommost MPLS Label Value and forward the packet to the correct Linux container after popping off the outermost MPLS header. The container will forward the packet back to the virtual switch, if it is not filtered out. The middlebox virtual switch will forward it back again to A1 after looking up flow entries in its flow table. A1 now matches on the new outermost MPLS Label Value—33. The packet will eventually reach the load balancer via C1, and from there to the slave VM.

5.2 Handling Middlebox Failure

In case a middlebox fails, the controller needs to recalculate a new middlebox instance for all policies which used the failed middlebox. After recalculating the new middlebox instance, the controller installs the new policy rule as a high priority entry on host virtual switches. Once the entry gets installed, new packets coming out from the host will start using it. Since we are using source routing, at no point during this operation will we violate the correctness of the policy. This process of recomputing new middlebox instance gives us the new optimal path through the network but it can be slow.

Another alternative is to make use of *fast failover* group type in switch group table supported by OpenFlow standard starting from v1.1. In case the hardware switch supports it, we can make use of this group type to store precomputed backup routes on a switch and use them in case the link to a middlebox fails. This scheme might lead to increase in packet latency, since the packet might take a non-optimal path through the network but it has fast response time.

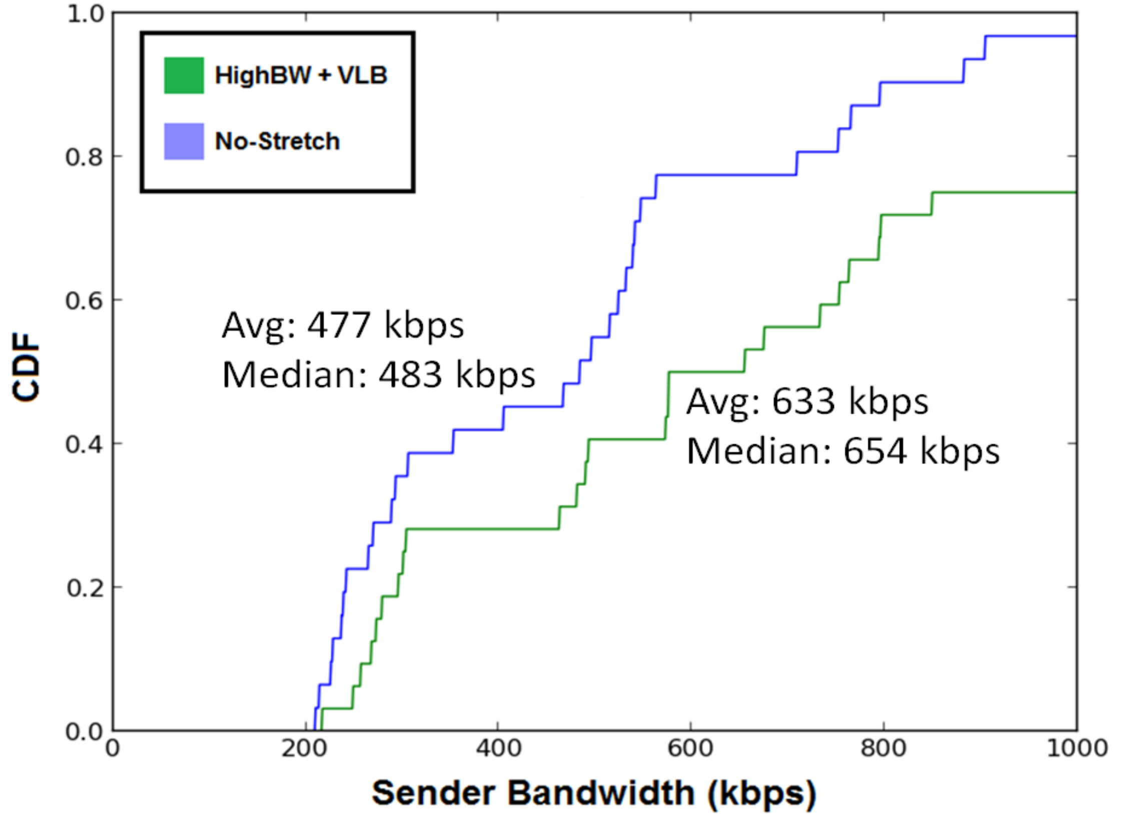

Figure 7: CDF of sender bandwidth for No-Stretch and High-Bandwidth PARIS.

5.3 OpenFlow Standard Modifications

We make two modifications to the OpenFlow standard for implementing our middlebox support architecture:

- Since we are using MPLS prefixes in the core layer, we need to allow masking of MPLS Label Values just like IP addresses. OpenFlow standard currently allows masking of source and destination MAC address, source and destination IP address and the Metadata register. Masking of MPLS Label Value is easy to implement in hardware using TCAMs. We don't see any reason why it should be hard for the standard to include this modification. Also, if the number of middleboxes is small enough that each core-layer switch can store forwarding information for all middleboxes, we can go without implementing MPLS prefixes.

- We need to perform matching on MPLS Label Value of bottommost MPLS header instead of outermost MPLS header in middlebox virtual switches. This this is done only inside the software switch and we require no hardware support for this. This modification is non-intrusive, since data center networks provide us the luxury of modifying end-host software as we see fit.

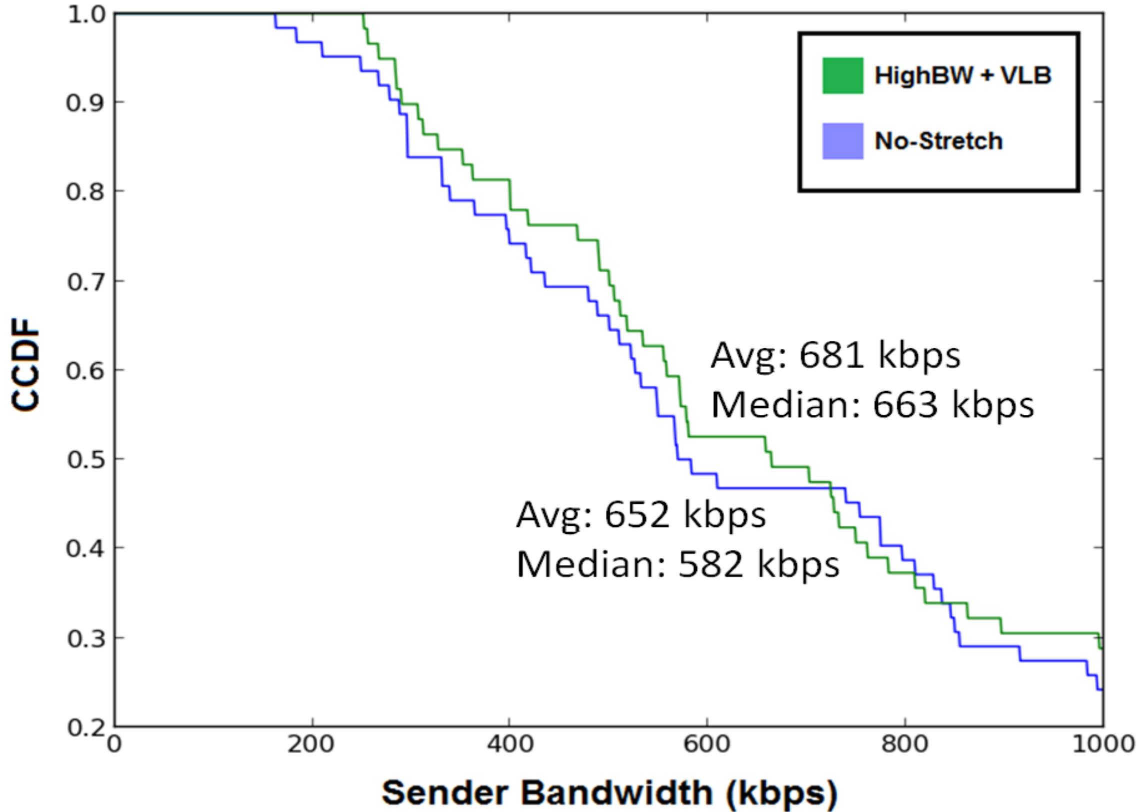

Figure 8: CCDF of sender bandwidth for No-Stretch and High-Bandwidth PARIS.

5.4 Discussion

In Section 4 and Section 5 we saw how to implement multi-tenancy and middlebox support for PARIS, while preserving its scalability, host mobility, multi-pathing and easy configurability. The No-Stretch and High-Bandwidth forwarding schemes that we saw in Section 3 can be used in a private data centers without requiring external gateway servers. The last two requirements, however, require the use of gateway servers for handling external traffic. The gateway servers help to implement middlebox policies and support multi-tenancy by tagging packets with tenant MPLS Label Value. Source routing at end host and gateway servers makes it easier for us to support both flow affinity and symmetry properties desired by certain middlebox classes.

6 Evaluation

We built a prototype of our architecture using NOX [28] OpenFlow controller and OpenFlow 1.3 compatible user-space software switches [50]. Our system has ∼6300 LOC in C++ and ∼3800 LOC in Python.

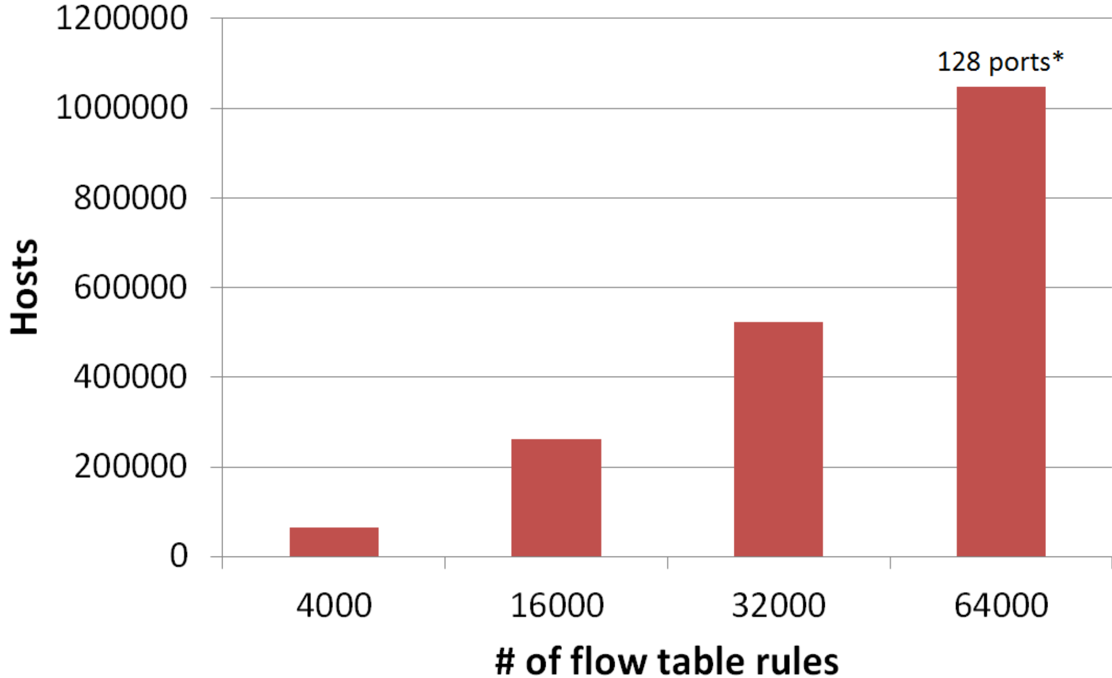

Figure 9: Scalability of No-Stretch PARIS.

In this section, we will evaluate the scalability and multi-pathing properties of our architecture. In the first set of evaluation experiments, we show that High-Bandwidth PARIS is able to achieve higher inter-host communication bandwidth compared to No-Stretch PARIS. This is because it is able to spread traffic across the aggregation-core layer links and it uses VLB inside the core layer. We use Mininet-HiFi [51] for simulating a data-center environment.

For our first experiment we implement a data center network with 32 hosts, 16 edge switches, 8 aggregation switches, and 4 core switches. In High-Bandwidth PARIS, we have a full-mesh in the core layer with each aggregation switch connected to all core switches. The network has no over-subscription at any layer. We connect the edge switches to the hosts via 1Mbps links and connect the switches to each other using 10Mbps links. The link bandwidths were scaled down compared to those in traditional data centers to make the experiments run faster. Our measurements showed that the intra-pod RTT is approximately 61μs for both schemes but the inter-pod RTT for No-Stretch PARIS and High-Bandwidth PARIS is approximately 106μs and 126μs respectively. These results reflect the fact that RTT increases as the average path length increases. Since the packet has to travel a longer path through the core layer in High-Bandwidth PARIS, it has slightly higher inter-pod RTT.

For our simulations, we use a random traffic pattern, where each host sends traffic to one randomly selected host for 20s. We use iperf [52] for generating TCP traffic between the hosts. Figure 7 shows the CDF of sender bandwidth for both No-Stretch and High-Bandwidth PARIS. Since there is no oversubscription in the network, a sender can at maximum achieve a bandwidth of 1000kbps if there is no flow-collision in the switch layers. As expected, given the random traffic pattern, we see that High-Bandwidth PARIS achieves higher average sender bandwidth compared to No-Stretch PARIS.

We now bias the topology in favor of No-Stretch PARIS and run the experiment again. This time, we create a topology with 64 hosts, 32 edge switches, 16 aggregation switches and 8 core-layer switches. In No-Stretch PARIS, each aggregation switch will be connected to all 8 core switches but in High-Bandwidth PARIS we connect each aggregation switch to 4 random core switches. So, the degree of multi-pathing is reduced. We run the experiment again and we find that High-Bandwidth PARIS still achieves higher average sender bandwidth compared to No-Stretch PARIS. Figure 8 shows the complementary CDF for this experiment. We can leverage other solutions like Hedera [53] to make sure flows don't collide in the data center network.

Though small in scale, our simulations demonstrate the unique properties of No-Stretch and High-Bisection PARIS.

Next, we run simulations to demonstrate the scalability of No-Stretch PARIS and High-Bandwidth PARIS. We assume each tenant has 512 VMs and each server hosts 64 VMs in the data center network. Also, $\sim$40% of network appliances are middleboxes. We use 64x10Gbps switches for constructing the network. We vary the flow table size of the switches and find out the number of hosts we can support in the data center network. Figure 9 and Figure 10 show this simulation for No-Stretch PARIS and High-Bandwidth PARIS respectively. No-Stretch PARIS can support up to 500K hosts with switches that can store 32K flow entries and can go up over a million hosts if we use 128 port switches that can store 64K flow entries. For High-Bandwidth PARIS, the flow table size needed to support 500K is higher because we need to store more rules to perform tunneling in the core layer. For this purpose we can use external TCAMs [35].

Finally we run an experiment to measure the run time of MB_{find} algorithm. We calculate the time for finding the middlebox instances for a single policy (Eq. 2) for all the hosts of a tenant. Table 2 shows our results.

Hosts	Vertices	Edges	Time
16384	704	2368	0.27s
65536	2592	9344	4.04s
131072	4368	9792	11.51s

Table 2: Run time analysis of MB_{find}.

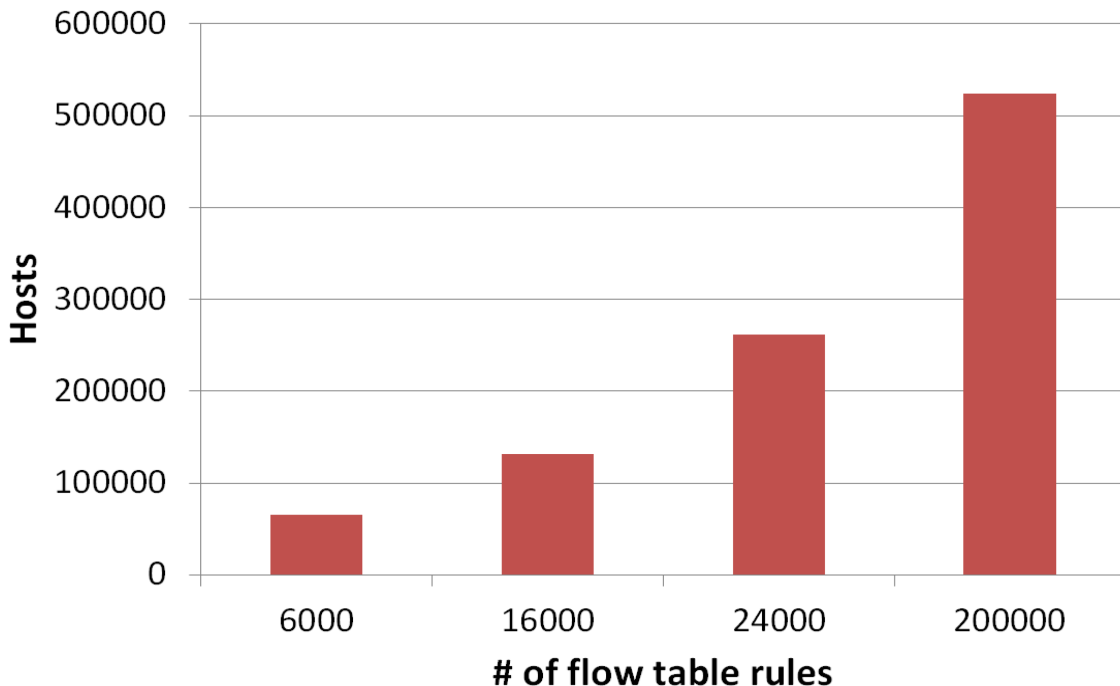

Figure 10: Scalability of High-Bandwidth PARIS.

7 Conclusion

In this work, we demonstrate how Proactive Routing on flat IP addresses can be used to build scalable and flexible data center networks. We eliminate flooding/broadcasting of packets and avoid querying directories for VM location. Instead, we use a controller to pre-position forwarding state in each switch layer. We propose a new core-layer topology which provides us with increased multi-pathing and bisection bandwidth. We also show how, through installation of fine-grained forwarding entries for popular destinations, we can further improve performance. We then extend our architecture for supporting multi-tenancy and middlebox policies. Our solution for multi-tenancy requires OpenFlow v1.2 supporting switches with two flow tables and our solution for supporting middlebox policies requires some modifications to the OpenFlow standard. Finally, we evaluate our architecture on Mininet-HiFi using NOX controller and user-space software OpenFlow switches.

References

[1] Cloud Computing Takes Off: Market Set to Boom as Migration Accelerates. http://www.morganstanley.com/views/perspectives/cloud_computing.pdf.

[2] 2012-13 IOUG Cloud Computing Survey. http://www.ioug.org/d/do/2897.

[3] Amazon data center size. http://huanliu.wordpress.com/2012/03/13/amazon-data-center-size/.

[4] M. Arregoces and M. Portolani. *Data Center Fundamentals*. Cisco Press, 2003.

[5] C. Clos. A study of Non-blocking Switching Networks. *Bell System Technical Journal*, 32:406424, 1953.

[6] C. E. Leiserson. Fat-Trees: Universal Networks for Hardware-Efficient Supercomputing. *IEEE Transactions on Computers*, 34:892–901, 1985.

[7] Z. Kerravala. Configuration management delivers business resiliency. The Yankee Group, Nov 2002.

[8] Amazon Virtual Private Cloud. http://aws.amazon.com/vpc/.

[9] Windows Azure Virtual Network. http://msdn.microsoft.com/en-us/library/windowsazure/jj156007.aspx.

[10] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel. The Cost of a Cloud: Research Problems in Data Center Networks. *ACM SIGCOMM Computer Communications Review*, January 2009.

[11] E. Nordström, D. Shue, P. Gopalan, R. Kiefer, M. Arye, S. Y. Ko, J. Rexford, and M. J. Freedman. Serval: An End-Host Stack for Service-Centric Networking. In *Networked Systems Design and Implementation*, 2012.

[12] D. Thaler and C. Hopps. Multipath Issues in Unicast and Multicast Next-Hop Selection. RFC 2991, November 2000.

[13] What's Behind Network Downtime? http://www-935.ibm.com/services/tw/gts/pdf/200249.pdf.

[14] Network Downtime, the Configuration Errors. http://networkworld.com/documents/whitepaper/wpnetworkdowntimetheconfigurationerrors.pdf.

[15] T. Benson, A. Akella, A. Shaikh, and S. Sahu. CloudNaaS: A Cloud Networking Platform for Enterprise Applications. In *ACM SOCC*, 2011.

[16] D. A. Joseph, A. Tavakoli, and I. Stoica. A Policy-aware Switching Layer for Data Centers. In *ACM SIGCOMM*, 2008.

[17] IETF TRILL working group. http://www.ietf.org/html.charters/trill-charter.html.

[18] J. Mudigonda, P. Yalagandula, M. Al-Fares, and J. C. Mogul. SPAIN: COTS Data-Center Ethernet for Multipathing over Arbitrary Topologies. In *Networked Systems Design and Implementation*, 2010.

[19] B. Stephens, A. Cox, W. Felter, C. Dixon, and J. Carter. PAST: Scalable Ethernet for Data Centers. In *ACM CoNEXT*, 2012.

[20] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network. In *ACM SIGCOMM*, 2009.

[21] Network Virtualization Platform (NVP). http://nicira.com/en/network-virtualization-platform.

[22] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. PortLand: A Scalable Fault-Tolerant Layer 2 Data Center Network Fabric. In *ACM SIGCOMM*, 2009.

[23] A. Greenberg, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. Towards a Next Generation Data Center Architecture: Scalability and Commoditization. In *Workshop on Programmable Routers for the Extensible Services of Tomorrow*, 2008.

[24] L. G. Valiant and G. J. Brebner. Universal schemes for parallel communication. In *ACM Symposium on Theory of computing*, 1981.

[25] M. Scott, A. Moore, and J. Crowcroft. Addressing the Scalability of Ethernet with MOOSE. In *DC CAVES Workshop*, 2009.

[26] C. Kim, M. Caesar, and J. Rexford. Floodless in SEATTLE: A Scalable Ethernet Architecture for Large Enterprises. In *ACM SIGCOMM*, 2008.

[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 Communications Review*, January 2008.

[28] 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 Communications Review*, July 2008.

[29] H. Ballani, P. Francis, T. Cao, and J. Wang. Making Routers Last Longer with ViAggre. In *Networked Systems Design and Implementation*, 2009.

[30] OpenFlow Switch Specification Version 1.1.0. http://www.openflow.org/documents/openflow-spec-v1.1.0.pdf.

[31] Cisco Global Cloud Index: Forecast and Methodology, 20112016. http://www.cisco.com/en/US/solutions/collateral/ns341/ns525/ns537/ns705/ns1175/Cloud_Index_White_Paper.html.

[32] N. G. Duffield, P. Goyal, A. Greenberg, P. Mishra, K. K. Ramakrishnan, and J. E. van der Merwe. A Flexible Model for Resource Management in Virtual Private Networks. In *ACM SIGCOMM*, 1999.

[33] R. Zhang-Shen and N. McKeown. Designing a Predictable Internet Backbone Network. In *ACM SIGCOMM HotNets Workshop*, 2004.

[34] Expander graphs and their applications. http://www.math.ias.edu/~boaz/ExpanderCourse/.

[35] A. Tavakoli, M. Casado, T. Koponen, and S. Shenker. Applying NOX to the Datacenter. In *ACM SIGCOMM HotNets Workshop*, 2009.

[36] OpenFlow Switch Specification Version 1.2.0. http://www.opennetworking.org/images/stories/downloads/openflow/openflow-spec-v1.2.pdf.

[37] 2012 Cloud Networking Report. http://www.webtorials.com/main/resource/papers/webtorials/2012-CNR/2012-CNR-Complete.pdf.

[38] Amazon Elastic Load Balancing. http://aws.amazon.com/elasticloadbalancing/.

[39] Rackspace Cloud Load Balancers. http://www.rackspace.com/cloud/load-balancing/.

[40] M. Casado, T. Koponen, S. Shenker, and A. Tootoonchian. Fabric: A Retrospective on Evolving SDN. In *ACM Hot topics in Software Defined Networks*, 2012.

[41] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for Network Update. In *ACM SIGCOMM*, 2012.

[42] 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 *ACM SIGCOMM*, 2012.

[43] A. Gember, A. Krishnamurthy, S. St. John, R. Grandl, X. Gao, A. Anand, T. Benson, A. Akella, and V. Sekar. Stratos: A Network-Aware Orchestration Layer for Middleboxes in the Cloud.

[44] J. Sherry and S. Ratnasamy. A Survey of Enterprise Middlebox Deployments. Technical Report UCB/EECS-2012-24, EECS Department, University of California, Berkeley, 2012. `http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-24.html`.

[45] B. Carpenter and S. Brim. Middleboxes: Taxonomy and Issues. RFC 3234, February 2002.

[46] Linux Bridge. `http://www.linuxfoundation.org/collaborate/workgroups/networking/bridge`.

[47] Linux Ethernet bridge firewalling. `http://ebtables.sourceforge.net/`.

[48] Linux Virtual Server. `http://www.linuxvirtualserver.org/`.

[49] C. Kopparapu. *Load Balancing Servers, Firewalls, and Caches*. John Wiley & Sons Inc., 2002.

[50] OpenFlow 1.3 Software Switch. `https://github.com/CPqD/ofsoftswitch13`.

[51] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown. Reproducible Network Experiments using Container Based Emulation. In *ACM CoNEXT*, 2012.

[52] Iperf. `http://iperf.sourceforge.net/`.

[53] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *Networked Systems Design and Implementation*, 2010.