Applying NOX to the Datacenter

Arsalan Tavakoli
UC Berkeley

Martin Casado and
Teemu Koponen
Nicira Networks

Scott Shenker
UC Berkeley, ICSI

1 Introduction

Internet datacenters offer unprecedented computing power for a new generation of data-intensive computational tasks. There is a rapidly growing literature on the operating and distributed systems issues raised by these datacenters, but only recently have researchers turned their attention to the datacenter's unique set of networking challenges. In contrast to enterprise networks, which usually grow organically over time, datacenter networks are carefully and coherently architected, so they provide an isolated setting in which new networking designs can be explored and deployed.

The combination of interesting intellectual challenges and lowered deployment barriers makes datacenters a rich arena for network research and innovation, as evinced by the recent flurry of research papers on datacenter networks. Of particular interest are the network designs proposed in [1, 5, 6, 9], which vary along many design dimensions but are all specifically tailored to the datacenter environment.

In the more general networking literature, in 2004 the 4D project [4] initiated a renaissance in the network management literature by advocating a logically centralized view of the network. The goal of this approach was to provide a general management plane, not specialized to a particular context (such as the datacenter). A recent development in this vein is the NOX network operating system [7]. NOX gives logically centralized access to high-level network abstractions such as users, topology, and services, and exerts control over the network by installing flow entries in switch forwarding tables. By providing programmatic access (through Python or C++) to network observation and control primitives, NOX serves as a flexible and scalable platform for building advanced network management functionality. Enterprise network management systems built on NOX have been in production use for over a year, and an early version of NOX is freely available under the GPL license at www.noxrepo.org.

This philosophical question behind this paper is whether the general-purpose approach in networking, which has served the Internet and enterprise so well, can be extended to specialized environments like the datacenter, or if special-case solutions will prevail. The more practical instantiation of this question is: How well does a general-purpose management system, like NOX, cope with the highly specific and stringent requirements of the datacenter? As we explain in this paper, we find that not only can NOX provide reasonable management of datacenter environments, it also offers operators a choice of several points in the datacenter design spectrum, rather than locking them into one specific solution.

Due to our familiarity with it, we use NOX throughout the paper as a concrete instance of a general network platform. However, the goal is to explore more broadly whether a general approach can be used in place of point solutions. Hence, this discussion should apply equally well to similar systems such as 4D [4], or Maestro [2]. We also don't make the claim that these systems are *better* than any of these point solutions. Our only goal is to demonstrate that there is still hope for the "general-purpose" philosophy that has served networking so well.

In the rest of the paper, we first present background material on datacenter networks and NOX (§2), and then demonstrate NOX's flexibility by describing implementations of existing architectures that can scale to a hundred thousand servers and millions of VMs (§3). We subsequently discuss how NOX provides basic datacenter functionality (§4) and several additional capabilities (§5). We end with a general discussion (§6).

2 Background

Before delving into how NOX can help datacenter networks, we first review datacenters, describe their networking requirements, present the VL2 and PortLand proposals, and provide some background on NOX.

2.1 Overview of Datacenters

Large-scale Internet datacenters are typically comprised of many racks of commodity PCs. Each rack has a top-of-rack (ToR) switch, and the ToRs are typically connected to an internal hierarchical network consisting of Aggregation switches and Core routers [3]. Datacenters range in size from hundreds of machines to hundreds of thousands of machines, and are used for a wide variety of tasks. To motivate our discussion of their networking requirements we focus on three particular types of datacenters. First, there is what we call *bare-metal* datacenters. These use PCs running Linux without a hypervisor to process data at large scale — various search engine datacenters are examples in this category. Second, there are *virtualized* datacenters: to provide more flexible management of

resources, these datacenters use hypervisors to support multiple VMs per host. Third, *multitenant* datacenters are a particular subset of virtualized datacenters where the VMs belong to many different customers — *cloud computing* datacenters (such as those that host Amazon's EC2 service) are representatives of this category.

2.2 Networking Requirements

We now briefly describe the set of networking requirements for these datacenters. Some of these requirements were gathered from existing literature (see, for example, [5, 9]), while others surfaced from discussions with datacenter operators and cloud service providers.

Scaling: The most pressing requirement, shared by all large datacenters, is to reach scales of hundreds of thousands of servers and millions of VMs. Two aspects that make scaling hard are forwarding table size and broadcasts, since both typically increase linearly with system size.

Location Independence: Dynamic resource provisioning in datacenters is much more efficient if resources can be assigned independent of their network location and VMs can migrate freely without interruption in service.

Service Quality: Unlike wide-area networks, datacenters are expected to achieve high levels of performance. This includes the ability to tolerate network failures, enforce network isolation between tenants, and provide load balancing to avoid network hotspots.

Additional Requirements: Two other features that are often mentioned as requirements are the ability to support middlebox interposition based on traffic classification and a general monitoring and troubleshooting capability.

2.3 Two Recent Datacenter Networking Proposals

As benchmarks for NOX, we briefly review two recent datacenter networking proposals. This is obviously not a comprehensive list, but both represent some of the best and most recent work in the research community.

PortLand: PortLand [9] uses a fat tree topology and position-based MAC addresses (PMACs) to achieve very compact routing state. ToR and Aggregation switches are grouped together in pods, and every Core switch is connected to every pod with a single link. Every VM (or host, if virtualization is not used) is assigned a PMAC of the form: *pod.position.port.vmid*. These PMACs enable PortLand switches to use longest prefix-match to reduce forwarding state. ToR switches perform the rewriting between PMACs and real MACs, so hosts and VMs need not be aware of this addressing structure. All switches maintain a full link-connectivity matrix in order to calculate routes locally. A centralized fabric manager maintains all PMAC-to-MAC mappings, as well as the full connectivity matrix.

VL2: VL2 [5] uses a Clos network topology with full bandwidth available for server-to-server traffic. VL2 implements Valiant Load Balancing (VLB) by sending traffic to random intermediate switches. This is accomplished by giving every Core switch the same anycast address and using ECMP, which randomly picks one of the shortest paths. All switches are assigned a location-specific IP address (LA), and use OSPF to build forwarding tables. All physical servers and services have application-specific IP addresses (AA). A centralized address manager (AM) maintains the AA-to-LA address mappings, and IP-in-IP encapsulation is used to route on LAs, but deliver packets to AAs. To avoid hardware modification, this encapsulation is implemented as a layer 2.5 stack on hosts, which consults the AM for the mappings before sending packets.

These designs use their topology, ECMP routing, and address-rewriting to achieve small forwarding tables, load balancing, and location-independence (however, seamless VM migration requires additional mechanisms). They deal with broadcasts by directly handling ARP and DHCP traffic, and using multicast for other broadcasts. Failures of any of the Aggregate and Core switches are handled by using alternate paths.

2.4 Overview of NOX

Both of these aforementioned designs have a centralized component to handle address mappings (and, in the case of PortLand, the connectivity matrix). NOX extends these centralized capabilities while still preserving scalability. Whereas VL2 and PortLand provide a small set of logically centralized primitives, NOX provides a logically centralized programming environment with a powerful set of abstractions.

A NOX-managed network consists of one or more *controllers* running NOX and a set of controllable switches. The controllable switches allow NOX to specify flow entries in their forwarding table. These flow entries are of the form $<\ header\ ,\ action\ >$, where the second item describes the action that should be applied to any packet that matches the header fields listed in the first item. Commonly used actions include: *forward to port x*, *forward to a controller*, *drop*, and *forward to port x with the following header fields rewritten with y*. Switches that support the OpenFlow standard [11] are examples of controllable switches, but NOX can use any similar protocol for manipulating switch forwarding entries.

NOX is described in [7], and more documentation is available at www.noxrepo.org, so we don't describe its operation in detail here. We note only the two most salient features of NOX for our discussion here. First, NOX provides complete visibility of the network

topology, including network links, network switches, and the location of all hosts. Second, NOX can operate in either pro-active or re-active mode. In the former, it uses the topology information to pre-load forwarding tables in the switches. In the latter, it has the switches send the first packet from each new flow (i.e., a packet for which the switch does not have a flow entry) to a controller, and the controller, after consulting its topology information, then installs the appropriate flow entries along the desired route. The pro-active approach minimizes flow latency, while the re-active approach allows the controller to make decisions on a flow-by-flow basis, taking QoS requirements and load conditions into account.

In larger networks, multiple NOX controllers act in parallel: each new flow notification is sent to one of the controllers, which installs the resulting flow entries without coordinating with other controllers. The only state for which NOX must maintain global consistency are the network topology and host address mapping. These change slowly enough that consistency is straightforward to achieve. Recent benchmarks on currently deployed NOX controllers [10] show an individual controller can handle at least 30K new flow installs per second while maintaining a sub-10ms flow install time. The controller's CPU is the bottleneck, and NOX's overall capacity scales roughly linearly with the number of controllers.

3 Scaling NOX to the Datacenter

The size of switch forwarding tables is the most vexing scaling requirement, so we start by estimating the size of the forwarding tables needed by NOX. Our analysis is based on the topology and empirical data from [5]. We realize that other datacenters may have different topologies and data characteristics, but our goal here is to provide an initial estimate of NOX's ability to cope with datacenter scales.

Analytical Setup: We perform our analysis for datacenters with 6K, 20K, and 100K servers, running in both bare-metal (1 application/server) and highly virtualized (20 VMs/server) mode. The network is a 3-layer Clos topology: each rack has 20 servers connected by 1Gbps links to a single ToR switch, which connects to two Aggregation switches using 10Gbps ports, and each Aggregation switch connects to all Core switches. The network is designed with no over-subscription, so the servers' NICs are the bottleneck for server-to-server traffic. As per [5], we assume each server instance has on average 10 concurrent flows (5 incoming and 5 outgoing).

To provide context for our estimates of flow-table size, we note that low-cost 1Gbps switches exist today with ACLs of up to 4K entries which can utilize low-cost external TCAMs that support roughly 250K source/destination pair flow entries. Commoditization trends suggest that low-cost 10Gbps equivalents may not be far behind (currently, we are aware of ACL sizes on the order of 1K, but we do not know whether these low-cost 10Gbps switches support external TCAMs).

NOX itself does not dictate any particular network topology or routing algorithm; it is a network control platform on which general routing algorithms can be implemented. As an example of NOX's flexibility, we now show how NOX can implement the routing schemes in VL2 and PortLand.

Implementing VL2: Recall that VL2 [5] achieves small forwarding tables and VLB through IP-in-IP encapsulation and ECMP. NOX can provide VLB without using ECMP or encapsulation; it only requires access to a single byte in the packet header. We take this byte from the source IP address since the number of hosts in a datacenter consumes only a tiny fraction of the IP address space. We will refer to this byte as the *coreID*. When a packet from a new flow arrives at the first-hop ToR switch, the switch forwards the packet to a NOX controller. The controller then installs the corresponding flow entry in the ToR switch, directing the switch to: (a) rewrite the flow's packet headers by replacing the destination IP with that of the last-hop ToR switch, and inserting the identifier for a randomly selected Core switch into the coreID, and (b) forward packets to one of its potential Aggregate switches (the controller chooses which one). The controller also installs a flow entry in the destination ToR switch, instructing it to restore the correct destination IP address and forward to the appropriate port. When the Aggregate switch receives the packet from the source ToR, it forwards the packet to the appropriate Core switch based on the coreID. The Core switch maintains per-ToR switch state (supplied to it by a controller pro-actively), and forwards the packet to the destination Aggregate switch for that destination ToR. This Aggregate switch has per-ToR state and can forward the packet to the appropriate ToR.

Figure 1 presents the results of our scalability analysis for implementing this approach, comparing NOX and VL2 [5] for similar setups. The Core switch state is similar in NOX and VL2, while aggregation switch state is far less in NOX. For ToR switches, NOX requires more state in the smaller virtualized networks. This increased state results from VL2 maintaining destination to last-hop ToR mappings at the physical servers, rather than the first-hop ToR.

Implementing PortLand: Recall that PortLand [9] uses positional MAC addresses (PMACs) that embed pod, position, port, and VM id. This enables compressed forwarding tables that can use longest prefix matching. The first and last-hop ToR switches handle the mapping between MACs and PMACs and vice versa. NOX can also maintain a similar PMAC mapping for all

	Server Only						20 VMs per server					
Physical Nodes	6000		20000		100000		6000		20000		100000	
Racks	300		1000		5000		300		1000		5000	
Agg. Switches	50		84		139		50		84		139	
Core Switches	12		24		72		12		24		72	
	VL2	**NOX**	**VL2**	**NOX**	**VL2**	**NOX**	**VL2**	**NOX**	**VL2**	**NOX**	**VL2**	**NOX**
Entries at ToR	382	120	1128	120	5231	120	762	2400	1508	2400	5611	2400
Entries at Agg.	362	24	1108	48	5211	144	362	24	1108	48	5211	144
Entries at Core	362	300	1108	834	5211	1389	362	300	1108	1000	5211	5000

Figure 1: Average number of flow entries installed at switches for VL2 and NOX.

registered hosts or VMs in the network, and preload the compressed forwarding tables at each switch. Thus, the state requirements of NOX and PortLand are the same. These requirements are substantially less than those in Figure 1, and so we omit this analysis for brevity.

These two examples show how NOX can easily support multiple (at least two) scalable topology and addressing schemes, without incurring significant additional flow-table state. NOX thus allows operators to choose between these, or other potential, designs by using the same basic management framework. There are two other scaling worries we must consider:

Controller Scalability: Since NOX is logically centralized, we must ask whether it can handle the rate of new flows. In the largest topology considered in our routing analysis, we had 2 million VMs. If we assume that each flow lasts 1 second (100MB over a 1Gbps link), this results in 20 million new flows per second (assuming a new entry per flow). Such a workload would require 667 controllers, or only 0.03% of the number of end hosts. Note that both VL2 and PortLand need a similar number of centralized controllers to handle address mappings (because they both use a mapping service with roughly the same query capacity per server).

Broadcast: Any large network must scalably cope with broadcast traffic. As with VL2 and PortLand, NOX will handle all ARP and DHCP broadcasts itself (i.e., these broadcast packets are sent to a controller, which then responds directly to the source with the appropriate response). These two protocols represent the vast bulk of broadcast usage, and for safety NOX will suppress all other broadcast messages. However, in a multitenant situation, customers may want a private L2 broadcast domain. In this case, NOX establishes flow entries for a spanning tree with broadcast rules. These entries prevent any other source from sending packets on this L2 broadcast domain. More generally, NOX can enforce security isolation and general access controls within the datacenter, since it sets up state for every new flow, and can refuse to do so for prohibited communication pairs.

4 Providing Core Functionality

We have shown how NOX supports small forwarding tables, broadcast, isolation, and aspects of location independence and load balancing. We now discuss how NOX enables other core datacenter capabilities.

Location Independence: The previous examples show that NOX can support address remapping and per-flow routing, making clear that NOX can support a general addressing scheme (meaning that servers can be located anywhere within the datacenter). We now address how NOX supports live migration of VMs.

To demonstrate NOX's ability to support live migration, we use the topology portrayed in Figure 2. Server A is sending traffic to a VM that is instantiated on Server B, and during the course of this flow, the VM migrates to Server C. We assume the proper flow entries are installed in all switches along the path. Upon receiving notification

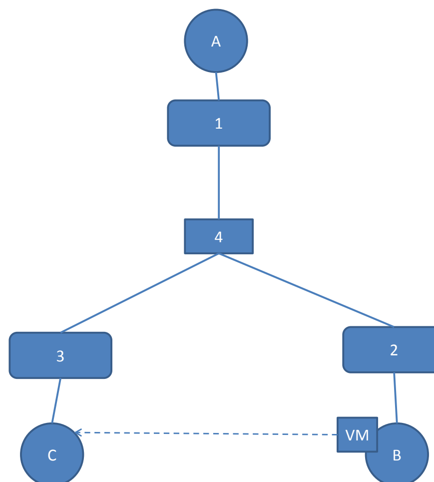

Figure 2: VM Migration Example

that the VM is moving, a NOX controller removes all flow entries for that VM from its first hop switch (Switch 2 in our topology). For all flows initiated after this removal, the first packet is sent to a controller by the flow's first-hop switch, and the controller installs the appropriate path. For any ongoing flow, which had a set of flow entries in

the network leading to Switch 2, when a packet from this flow arrives at Switch 2, it will forward the packet to a NOX controller (since the switch no longer has a flow entry for flows destined for the VM). The controller can identify the source of this flow from the packet's 10-tuple (Server A in this case) and the first-hop switch (Switch 1). It then deletes the stale flow entry from this first-hop switch, and installs an updated entry on the next new flow notification (i.e., the next time a packet from this flow arrives at Switch 1). Stale state at switches disjoint from the new path will time out, and state at other switches (such as Switch 4 in our example), will be modified to reflect the new path. As an optimization to prevent packet loss for any existing packets in flight, an entry can be installed that routes packets arriving at Switch 2, destined for the VM, to Switch 3.

Service Quality: NOX can provide fault tolerance through its ability to control routes. Existing switch mechanisms can determine link failures and notify NOX. When informed of a failed link by a switch, a NOX controller then flushes all the flow entries at that switch which use the failed link. When the next packet from each of these affected flows arrives at the switch, the packet will be forwarded to a controller which will then establish a set of flow entries along a new path (avoiding the failed link). In addition, OpenFlow provides the capability to install a second flow entry of a lower priority, allowing NOX to have previously calculated and installed back-up paths in case of failures in order to minimize latency.

In addition, NOX can utilize any QoS mechanisms provided by the network switches. NOX allows for the specification of multiple traffic categories (e.g. http traffic), and their associated QoS. When a packet from a new flow is sent to a controller, it can identify the traffic category and then reflect the category-specific QoS in the flow entries it installs along the chosen route. If QoS is not natively available to the switches, NOX provides more coarse-grained mechanisms that operate at the flow admission level. For example, NOX can rate-limit flow establishment for particular categories depending on their priority. We later discuss how NOX enables fine-grained monitoring and visibility into network state and load, enabling more accurate QoS provisioning.

NOX provides various options for achieving load balancing. As demonstrated earlier, NOX can support randomized schemes such as Valiant Load Balancing and ECMP. Alternatively, NOX can utilize real-time information about network load and the utilization of switches to install flows on uncongested links. At the application level, NOX can integrate with application-level load balancers into the network by directing the appropriate flows to them. NOX can also provide native application-level load balancing support, by directing packets sent to a particular virtual IP to one of corresponding servers.

5 Additional Capabilities

We now describe how NOX satisfies additional requirements consistently cited by operators as necessary functionality but not included in most proposed architectures.

Middlebox Traversal: Many datacenter applications require the traversal of middleboxes, which provide custom processing at layers 3-7. Similar to [8], NOX can support their use by having operators specify *classification-action* pairs, in which "classification" specifies the flow header for each category and an "action" is an ordered middlebox traversal sequence. For each new flow notification, NOX determines whether it matches any of the traffic categories, and if so, installs a path that explicitly routes the flow through the necessary middleboxes. Not all middleboxes are transparent, and some may modify the packet header, often in a non-deterministic fashion. These transformations can be predicted, as shown in [8], and NOX can utilize such mechanisms.

Network Visibility: In addition, operators need mechanisms to monitor and troubleshoot their network. NOX and OpenFlow provide the basis for an integrated monitoring architecture. OpenFlow switches maintain statistics on a per flow, per port, and per switch basis, and these can be polled by NOX and made available to the operator. Trouble spots can be identified using more detailed queries. For each ToR switch, NOX identifies entries that correspond to flows originating from a host connected to that switch. From the flow entry destination field, NOX also identifies the last hop switch for the path and examines that flow entry to ensure that the number of processed packets reported by the two entries differs by only a small number (such as could be due to the difference in sampling times). If there is a significant discrepancy, NOX then begins at the source, and traces the path of the flow (using the output port specified in the flow entry) to determine the switch(es) at which the discrepancy arose. Using this technique, NOX can quickly identify the location of packet losses.

Finer-grained control: We demonstrated in Section 3 that NOX can reproduce the routing and load-balancing functionality of VL2 and PortLand, with roughly equivalent state requirements. While desirable, small forwarding tables only provide operators with coarse-grained control. NOX can support finer granularities of control, all the way to the extreme case of controlling every flow independently, but at the cost of requiring larger forwarding tables. To quantify this state-control tradeoff, we examine how much state NOX would require to support two different granularities of control.

First, we allow Aggregation and Core switches to maintain a forwarding entry for each destination in the network, eliminating the need to enforce a single routing

	Server Only			20 VMs per server		
Physical Nodes	6000	20000	100000	6000	20000	100000
Entries at ToR	120	120	120	2400	2400	2400
Entries at Agg.	840	1680	5040	16800	33600	100800
Entries at Core	500	834	1389	10000	16667	27778

Figure 3: Average # of flow entries installed when per-destination entries are installed at all switches.

	Server Only			20 VMs per server		
Physical Nodes	6000	20000	100000	6000	20000	100000
Entries at ToR	200	200	200	4000	4000	4000
Entries at Agg.	1200	2400	7200	24000	48000	144000
Entries at Core	5000	8400	13900	100000	168000	278000

Figure 4: Average # of flow entries installed when per-source/destination entries are installed at all switches.

scheme for all destinations within a rack. To keep the state manageable, NOX enforces that all flows to a given destination are routed through the same core switch. Figure 3 details the number of forwarding table entries needed to support this control granularity, assuming that flow destinations are unique and randomly distributed.

The Server-only scenarios only require moderate state, but the Aggregation and Core switches need significantly more state in the large virtualized scenarios (though even the largest state requirement can be handled with a TCAM). Note that such numbers provide a worst-case analysis in many respects. The data from [5] indicated 10 concurrent flows per physical server; we have extrapolated this to be 10 concurrent flows per VM instance, leading to 200 concurrent flows per server, with no assumption of flow locality. Communicating VMs often share a common Aggregation switch, rack, or even physical server, which would reduce the size of the forwarding tables on Aggregation and Core switches.

Second, we consider the state requirements for controlling traffic at the source-destination granularity. Our estimates (Figure 4) assume that all source-destination pairs are unique, and thus correspond to the worst-case scenario of per-flow state. The number of entries is manageable in smaller networks but becomes quite high in the largest networks (again, though, even the largest state requirements are close to what a TCAM could handle).

Deciding which granularity to use will depend on the size and nature of the datacenter. In addition, a hybrid of any of these schemes could be used. For example, Valiant Load Balancing using per-ToR state can be used to provide scalability, but individual flows that have special needs, such as QoS or middlebox traversal, can have per-flow granularity entries installed.

6 Discussion

The technical details of this paper are all about NOX, but our point is far broader. NOX is merely one instantiation of a larger movement (started by the 4D project) towards software-based network management systems that combine sophisticated control planes with simple data planes. The goal of these management systems is to provide network control that is flexible enough to meet a wide range of needs. Moreover, in this vision the network management capabilities appropriate for specific settings are embodied in reusable software modules, allowing the community to build up a growing set of networking capabilities.

Demonstrating that NOX can address a variety of datacenter requirements is only a tiny step forward towards this broad vision. Our next step is to implement the designs discussed here and analyze them more thoroughly in practice.

7 References

[1] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *SIGCOMM*, 2008.

[2] Z. Cai, F. Dinu, J. Zheng, A. L. Cox, and T. E. Ng. Maestro: A new architecture for realizing and managing networked controls. In *LISA*, 2007.

[3] Cisco. Data center: Load balancing data center services.

[4] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang. A clean slate 4d approach to network control and management. *Computer Communication Review*, 2005.

[5] A. Greenberg, 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 *Sigcomm*, 2009.

[6] A. Greenberg, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. Towards a next generation data center architecture: scalability and commoditization. In *PRESTO*, 2008.

[7] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. Nox: Towards and operating system for networks. In *Computer Communication Review*, 2008.

[8] D. Joseph, A. Tavakoli, and I. Stoica. A policy-aware switching layer for data centers. In *Sigcomm*, 2008.

[9] R. N. 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 *Sigcomm*, 2009.

[10] Nicira Networks. private communication, 2009.

[11] OpenFlow Switch Consortium. Openflow switch specification.