NOX: Towards an Operating System for Networks

Natasha Gude Teemu Koponen Justin Pettit Ben Pfaff Martín Casado
Nick McKeown Scott Shenker

This article is an editorial note submitted to CCR. It has NOT been peer reviewed. Authors take full responsibility for this article's technical content.
Comments can be posted through CCR Online.

Categories and Subject Descriptors:
C.2.6 [Computer Communication Networks]: Internetworking
C.2.1 [Computer Communication Networks]: Network Architecture and Design

General Terms: Design, Experimentation, Performance
Keywords: architecture, management, network, security

1 Introduction

As anyone who has operated a large network can attest, enterprise networks are difficult to manage. That they have remained so despite significant commercial and academic efforts suggests the need for a different network management paradigm. Here we turn to operating systems as an instructive example in taming management complexity.

In the early days of computing, programs were written in machine languages that had no common abstractions for the underlying physical resources, making programs hard to write, port, reason about, and debug. Modern operating systems facilitate program development by providing controlled access to high-level abstractions for resources (*e.g.*, memory, storage, communication) and information (*e.g.*, files, directories). These abstractions enable programs to carry out complicated tasks safely and efficiently on a wide variety of computing hardware.

In contrast, networks are managed through low-level configuration of individual components. Moreover, these configurations often depend on the underlying network; for example, blocking a user's access with an ACL entry requires knowing the user's current IP address. More complicated tasks require more extensive network knowledge; forcing guest users' port 80 traffic to traverse an HTTP proxy requires knowing the current network topology and the location of each guest. In this way, an enterprise network resembles a computer without an operating system, with network-dependent component configuration playing the role of hardware-dependent machine-language programming.

What we clearly need is an "operating system" for networks, one that would provide a uniform and centralized programmatic interface to the entire network.[1] Analogous to the read and write access to various resources provided by computer operating systems, a network operating system would provide the ability to *observe* and *control* a network.

A network operating system would not manage the network itself; it would merely provide a programmatic interface. *Applications* implemented on top of the network operating system would perform the actual management tasks.[2] The programmatic interface should be general enough to support a broad spectrum of network management applications.

Such a network operating system represents two major conceptual departures from the status quo. First, the network operating system would present programs with a *centralized* programming model[3]; programs are written as if the entire network were present on a single machine (*i.e.*, one would use Dijkstra to compute shortest paths, not Bellman-Ford). This requires (as in [8, 14, 3] and elsewhere) centralizing network state. Second, programs would be written in terms of high-level abstractions (*e.g.*, user and host names), not low-level configuration parameters (*e.g.*, IP and MAC addresses). This allows management directives to be enforced independent of the underlying network topology, but it requires that the network operating system carefully maintain the bindings (*i.e.*, mappings) between these abstractions and the low-level configurations.

Thus, a network operating system allows management applications to be written as centralized programs over high-level names as opposed to the distributed algorithms over low-level addresses we are forced to use today. While clearly a desirable goal, achieving this transformation from distributed algorithms to centralized programming presents significant technical challenges, and the question we pose here is: *Can one build a network operating system at significant scale?* We argue for an affirmative answer to this question via proof-by-example; herein we describe a network operating system called NOX that achieves the goals outlined above.

Given the space limitations, we only give a cursory description of NOX, starting with an overview (Section 2), followed by a sketch of NOX's programmatic interface (Section 3) and a discussion of a few NOX-based management applications (Section 4). We discuss related work in Section 5, but before going further we want to emphasize NOX's intellectual indebtedness to the 4D project [8, 14, 3] and to the SANE [7] and Ethane [6] designs. NOX is also similar in spirit, but complementary in emphasis, to the Maestro system [4] which was developed in parallel.

---

[1] In the past, the term *network operating system* referred to operating systems that incorporated networking (*e.g.*, Novell NetWare), but this usage is now obsolete. We are resurrecting the term to denote systems that provide an execution environment for programmatic control over the full network.

[2] In the rest of this paper, the term applications will refer exclusively to management programs running on a network operating system.

[3] By *centralized* we allude to a shared memory programming model. However, as we discuss in Section 3, different memory locations may have different access overheads.

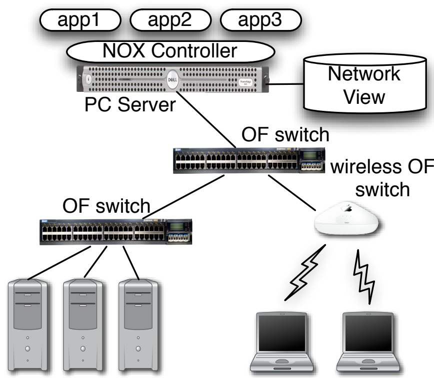

Figure 1: Components of a NOX-based network: OpenFlow (OF) switches, a server running a NOX controller process and a database containing the network view.

2 NOX Overview

We now give an overview of NOX by discussing its constituent components, observation and control granularity, switch abstraction, basic operation, scaling, status and public release.

Components Figure 1 shows the primary components of a NOX-based network: a set of switches and one or more network-attached servers. The NOX software (and the management applications that run on NOX) run on these servers. The NOX software can be thought of as involving several different *controller* processes (typically one on each network-attached server) and a single *network view* (this is kept in a database running on one of the servers).[4] The network view contains the results of NOX's network observations; applications use this state to make management decisions. For NOX to control network traffic, it must be able to affect the behavior of network switches; for this purpose we have chosen to use switches that support the OpenFlow (OF) switch abstraction [1, 12], which we describe later in this section.

Granularity An early and important design issue was the granularity at which NOX would provide observation and control. Choosing the granularity involves trading off scalability against flexibility, and both are crucial for managing large enterprise networks with diverse requirements. For observation, NOX's network view includes the switch-level topology; the locations of users, hosts, middleboxes, and other network elements; and the services (*e.g.*, HTTP or NFS) being offered. The view includes all bindings between names and addresses, but does *not* include the current state of network traffic. This choice of observation granularity provides adequate information for many network management tasks and changes slowly enough that it can be scalably maintained in large networks.

The question of control granularity was more vexing. A centralized per-packet control interface would clearly be infeasible to implement across any sizable network. At the other extreme, operating at the granularity of prefix-based routing tables would not allow sufficient control, since all packets between two hosts would have to follow the same path. For NOX we chose an intermediate granularity: *flows* (similar in spirit to [13]). That is, once control is exerted on some packet, immediately following packets with the same header are treated in the same way. With this flow-based granularity, we were able to build a system that can scale to large networks while still providing flexible control.

Switch Abstraction Management applications control network traffic by passing instructions to switches. These switch instructions should be independent of the particular switch hardware, and should support the flow-level control granularity described above. To meet these requirements, NOX has adopted the OpenFlow switch abstraction (see [1, 12] for details). In OpenFlow, switches are represented by flow tables with entries of the form:[5]

$$\langle header : counters, actions \rangle$$

For each packet matching the specified header, the counters are updated and the appropriate actions taken. If a packet matches multiple flow entries, the entry with the highest priority is chosen. An entry's header fields can contain values or ANYs, providing a TCAM-like match to flows. The basic set of OpenFlow actions are: forward as default (*i.e.*, forward as if NOX were not present), forward out specified interface, deny, forward to a controller process, and modify various packet header fields (*e.g.*, VLAN tags, source and destination IP address and port). Additional actions may later be added to the OpenFlow specification.

Operation When an incoming packet matches a flow entry at a switch, the switch updates the appropriate counters and applies the corresponding actions. If the packet does not match a flow entry, it is forwarded to a controller process.[6] These unmatching packets often are the first packet of a flow (hereafter, flow-initiations); however, the controller processes may choose to receive all packets from certain protocols (*e.g.*, DNS) and thus will never insert a flow entry for them. NOX applications use these flow-initiations and other forwarded traffic to *(i)* construct the network view (observation) and *(ii)* determine whether to forward traffic, and, if so, along which route (control).

As an example of *(i)*, we have built applications that use DNS, DHCP, LLDP, and flow-initiations to construct the network view, including both the network topology and the set of name-address bindings. We have also built applications that intercept authentication traffic to perform user and host authentications (using for example 802.1x). As an example of *(ii)*, we have developed access-control and routing applications that determine if a flow should be allowed, compute an appropriate L2 route, install flow entries in all the switches along the path, and then return the packet to the originating switch (which then forwards it along the designated path).

Scaling Our confidence in the scalability of NOX follows from considering the spectrum of timescales and consistency

[4]For resilience, this database can be replicated, but these replicas must be kept consistent (using traditional replicated database techniques).

[5]It is important to distinguish between the levels of abstraction provided by OpenFlow and NOX. NOX provides network-wide abstractions, much like operating systems provide system-wide abstractions. OpenFlow provides an abstraction for a particular network component, and is thus more analogous to a device driver.

[6]Typically, only the first 200 bytes of the first packet (including the header) are forwarded to the controller, but the controller may adjust this, or request additional packets be forwarded, if more information is deemed necessary.

requirements. In terms of timescales, NOX processing occurs at three very different rates:

- Packet arrivals: this is on the order of millions of arrivals per second for a 10Gbps link.
- Flow initiations: with NOX's definition of flow (which is typically more persistent than NetFlow's definition), the flow-initiation rate is typically one or more orders of magnitude less than the packet arrival rate.
- Changes in the network view: in our deployment experience this is on the order of tens of events per second for networks of thousands of hosts.

In terms of consistency, the only network state that is global (*i.e.*, must be used consistently across the controller processes) is the network view; this consistency requirement arises because applications use data from the network view to make control decisions, and those decisions should be the same no matter to which controller instance the flow has been sent. In contrast, since neither packet state nor flow state are part of the network view, they can be kept in local storage (*i.e.*, packet state in switches, and flow state in controller instances).

Thus, for the categories of events that occur on rapid timescales, NOX can use parallelism; packet arrivals are handled by individual switches without global per-packet coordination, and flow-initiations are handled by individual controller instances without global per-flow coordination. Flows can be sent to any controller instance, so the capacity of the system can be increased by adding more servers running controller processes.

The one global data structure, the network view, changes slowly enough that it can be maintained centrally (or, for resilience, it can be kept consistently on a small set of replicas) for even very large networks.

To give some rough numbers, a single controller process running on a generic PC can currently handle 100,000 flow initiations per second, more than sufficient for the large campus networks we've measured in previous work [6].

Implementation Status We have been developing NOX for over a year and have been running it in our internal network of roughly 30 hosts for over 6 months. It is our only means of network connectivity. NOX runs in user-space on the network servers. Applications are written in either Python or C++ and are loaded dynamically. The core infrastructure and speed-critical functions of NOX are implemented in C++ (currently about 32,000 lines). The network view is a set of indexed hashtables, with extensions for distributed access with local caching to aid scaling across multiple controller instances.

Public Release Our brief presentation of NOX is only the first step in our "proof" that one can build a network operating system. NOX is freely available at http://www.noxrepo.org (GPL license), and we invite the community to build additional applications on NOX. The community's experience will provide the definitive answer to the question of whether NOX provides a useful abstraction for network management.

3 Programmatic Interface

NOX's programmatic interface is conceptually quite simple, revolving around events, a namespace, and the network view.

Events Enterprise networks are not static: flows arrive and depart, users come and go, and links go up and down. To cope with these change events, NOX applications use a set of handlers that are registered to execute when a particular event happens. Event handlers are executed in order of their priority (which is specified during handler registration). A handler's return value indicates to NOX whether to stop execution of this event, or to continue by passing the event to the next registered handler.

Some events are generated directly by OpenFlow messages, such as *switch join*, *switch leave*, *packet received*, and *switch statistics received*. Other events are generated by NOX applications as a result of processing these low-level events and/or other application-generated events. For example, NOX includes applications that will authenticate a user by redirecting raw HTTP traffic (a *packet received* event) to a captive web portal. Once the user is authenticated, the NOX application generates a *user authenticated* event which can be used by other applications.

We have implemented applications that reconstruct the switch and host level topology, discover network services, authenticate users and hosts, enforce network policy, and detect network scanning (to name a few). Each of thes services is coupled with an associated event that can be leveraged by other applications.

Network View and Namespace NOX includes a number of "base" applications that construct the network view and maintain a high-level namespace that can be used by other applications. These applications handle user and host authentication, and infer host names by monitoring DNS. The inclusion of high-level names and their bindings in the network view allows any application to convert a high-level name into low-level addresses (or vice versa), allowing applications to be written in a topology independent manner. To perform such conversions, high-level declarations can be "compiled" against the network view to produce low-level lookup functions that are enforced per-packet. These functions are recompiled on each change to the network view. In Section4.2 we describe how this is used in practice.

Because the network view must be consistent and made available to all NOX controller instances, writing to it incurs some expense. Thus, NOX applications should only write to it when a *change* is detected in the network, and not for every received packet. This is similar to the access model provided by NUMA memory architectures. Also like NUMA, the worst result of a poorly written application is performance degradation, not incorrect function.

Control Management applications exert network control through OpenFlow. The OpenFlow switch abstraction allows applications to insert entries, delete entries, or read counters from entries in the flow table. Through its ability to modify these flow table entries, a management application has complete control of L2 routing, packet header manipulation, and ACLs. With the definition of additional switch actions, applications could also control common per-packet processing primitives such as encryption and rate-limiting.

The OpenFlow abstraction is intended to be general, so it can only require features common to most switches. Therefore we do not expect that the actions standardized by the OpenFlow Switch Consortium will include custom per-packet processing (*e.g.*, deep packet inspection). However, NOX applications can direct traffic through specialized middleboxes,

so NOX-managed networks can still take advantage of the latest per-packet processing technology.

Higher-Level Services NOX includes a set of "system libraries" to provide efficient implementations of functions common to many network applications. These include a routing module, fast packet classification, standard network services (such as DHCP and DNS), and a policy-based network filtering module.

Interface and Runtime Limitations Our goal with NOX is to build a practical platform for writing centralized network applications that can scale to large networks. So far, we have focused on scalability and functionality, and have yet to address a number of practical considerations that would improve the safety and isolation of NOX applications. For example, we assume that there is coordination between application writers and do not try to protect against malicious or faulty applications – a bad application can drop an event, overwrite random memory, or hang the system with an infinite loop. We feel providing inter-application coordination and isolation is a rich area for exploration in which the Maestro project has already made progress [4].

4 Example Applications

We now describe a few examples of NOX applications. To illustrate NOX's programming model, we start with two oversimplified examples. To give an idea of NOX's power, we then describe how we used NOX to re-implement Ethane [6], an identity-based access-control system. To convey NOX's flexibility, we end with two ongoing projects aiming for novel network functionality.

4.1 Two Simple Examples

User-based VLAN Tagging Figure 2 contains a simplistic NOX application that sets up VLAN tagging rules on user authentication based on a predefined user-to-VLAN mapping. NOX is responsible for detecting all flow-initiations, attributing the flow to the correct user, host and ingress access point, and dispatching the event to the application. This would provide attribution in logging and diagnostics; it could also, with minor modifications to the routing module, support traffic isolation.

Simplistic Scan Detection The application in Figure 3 attempts to detect scanning hosts by counting the number of unique L2 and L3 destinations a host tries to contact that have not authenticated. NOX has access to traffic across the network, and it can leverage the network view which tracks all authenticated hosts on the network. Contrast this simple implementation to that in [11], where scan detection required a traffic choke-point and heuristics to guess whether an IP address is active.

4.2 Ethane

We recently built a system called Ethane that provides network-wide access-control using a centralized declaration of policy over high-level principals (*i.e.*, entities in the network view namespace) [6]. Because we have implemented Ethane both with and without NOX, Ethane is an instructive example of how NOX simplifies management application development: our stand-alone Ethane implementation required over 45,000 lines of C++, while our implementation within NOX required a few thousand lines of Python.

```
# On user authentication, statically setup VLAN tagging
# rules at the user's first hop switch
def setup_user_vlan(dp, user, port, host):
    vlanid = user_to_vlan_function(user)
    # For packets from the user, add a VLAN tag
    attr_out[IN_PORT] = port
    attr_out[DL_SRC] = nox.reverse_resolve(host).mac
    action_out = [(nox.OUTPUT, (0, nox.FLOOD)),
            (nox.ADD_VLAN, (vlanid))]
    install_datapath_flow(dp, attr_out, action_out)
    # For packets to the user with the VLAN tag, remove it
    attr_in[DL_DST] = nox.reverse_resolve(host).mac
    attr_in[DL_VLAN] = vlanid
    action_in = [(nox.OUTPUT, (0, nox.FLOOD)),
            (nox.DEL_VLAN)]
    install_datapath_flow(dp, attr_in, action_in)
nox.register_for_user_authentication(setup_user_vlan)
```

Figure 2: An example NOX application written in Python that statically sets VLAN tagging rules on user authentication. A complete application would also add VLAN removal rules at all end-point switches.

```
scans = defaultdict(dict)
def check_for_scans(dp, inport, packet):
    dstid = nox.resolve_host_dest(packet)
    if dstid == None:
        scans[packet.l2.srcaddr][packet.l2.dstaddr] = 1
        if packet.l3 != None:
            scans[packet.l2.srcaddr][packet.l3.dstaddr] = 1
        if len(scans[packet.l2.srcaddr].keys()) > THRESHOLD:
            print nox.resolve_user_source_name(packet)
            print nox.resolve_host_source_name(packet)
# To be called on all packet-in events
nox.register_for_packet_in(check_for_scans)
```

Figure 3: A simplistic NOX application written in Python that attempts to detect scanning hosts by tracking the number of unique, unknown L2 and L3 destinations attempted by a single host.

Ethane has two requirements that make it difficult to implement using traditional network management techniques: *(i)* it requires knowledge of the principals on the network (*e.g.*, users, nodes[7]), and *(ii)* it requires control over routing at the granularity of a flow's 7-tuple (source user, source host, first-hop switch, protocol, destination user, destination host, last-hop switch).[8] Both of these are natively supported by NOX: the application has access to the source and destination principals associated with each event, and the routing module supports route computation with constraints.

Implementing the basic Ethane functionality within NOX involves first checking each flow directly against the declared policy and then passing the resulting contraints to NOX's routing module. The only subtle aspect of the implementation is how to efficiently check flows against the policy, since a linear scan of the policy declaration file for each flow does not scale as the complexity of the policy increases. To improve performance in the average case, we dynamically construct efficient lookup trees from the policy declarations.

[7]We use the term node to refer to hosts, switches, and other network elements.

[8]This requirement arises from policies that impose routing constraints on a particular class of flows, such as requiring them to traverse specified middleboxes.

4.3 Other Applications

The following are two examples of areas where NOX is being employed to explore new functionality.

Power Management There has been significant recent interest in managing networks to save power [9, 2]. The two techniques most commonly discussed are *(i)* reducing the speed of underutilized links, or turning them off altogether, and *(ii)* providing proxies to intercept network *chatter* (only allowing necessary packets to reach hosts would make wake-on-LAN more effective). NOX's global view of the network and the routes currently in use facilitates the former technique, while NOX's interposition on all flow-initiations facilitates the latter. Thus, NOX is an ideal platform for implementing these techniques, and there is ongoing work [10] pursuing this opportunity.[9]

Home Networking Calvert *et al.*, in [5], trace many of the difficulties in managing home networks to the end-to-end nature of the Internet architecture, and propose that a more centralized network design be used in homes. NOX could centralize the observation and control functions of the home network, while preserving the decentralized nature of the datapath (thereby avoiding potential bottlenecks). NOX would also provide a natural platform on which one could build management tools to handle the many higher-layer configuration issues (directing machines to local printers, etc.) that bedevil home networks. There are two ongoing efforts exploring NOX's use in the home.

5 Related Work and Open Issues

The idea of giving control mechanisms a global view of the network was first developed in the context of the 4D project (see [8, 14, 3]). Providing this view required a new networking paradigm based on simple switches enslaved to a logically centralized decision element that oversees the full network. This centralized paradigm is more flexible, since new functionality can be programmed at the decision element rather than requiring a new distributed algorithm, but raises the specter of a single point of failure. However, adequate resilience can be achieved by applying standard replication techniques to the decision element. Note that these replication techniques are completely decoupled from the network control algorithms, so they do not impede application innovation.

The goal of 4D systems is to control forwarding (*e.g.*, FIBs in routers), and thus their network view only includes the network infrastructure (*e.g.*, links, switches/routers). The SANE [7] and Ethane [6] projects provided a broader class of functionality by including a namespace for users/nodes in their network view and keeping track of the bindings between these names and the low-level MAC and IP addresses. SANE and Ethane also capture flow-initiation events, to exercise control at a finer granularity (per-flow control rather than FIB-based control).

NOX extends the SANE/Ethane work in two dimensions. First, it attempts to scale this centralized paradigm to very large systems. This scaling is made possible by the differing timescales discussed above. The second extension is allowing general programmatic control of the network. The SANE/Ethane systems were designed around a single application: identity-based access control. NOX aims to provide a general programming interface that makes it *easier* to support current management tasks and *possible* to provide more advanced management functionality. We have described a few example applications in this paper, but only experience will reveal how generally useful this interface is. By making NOX freely available, we hope that the community will provide valuable feedback on NOX's utility.

A related project of particular note is Maestro [4] (developed in parallel to NOX), which is also billed as a "network operating system". In general, operating systems can be seen as revolving around two basic purposes: *(i)* providing applications with a higher level of abstraction so they need not deal with low-level details, and *(ii)* controlling the interactions between applications. NOX focuses on the first, while Maestro focuses on the second, "orchestratin" the control decisions made by various management applications. We think these approaches could be combined, and we hope to soon explore this possibility.

Given that industry has substantially more experience with practical enterprise management and security than academia, we would be remiss in not mentioning commercial solutions. Many commercial enterprise security products, such as firewalls, intrusion detection and protection systems, network mappers, and proxies, are network *appliances* in that a particular functionality is provided by an element (or several elements) placed in the network. This appliance approach is easy to deploy, but leads to a fragmented architecture in which the different appliances, and their functionality, are completely decoupled. Moreover, none of these appliances provide the flexibility of a general programming environment for network observation and control.

One area where these commercial products are far more advanced than NOX is their ability to deal with packet payloads. Many commercial solutions use deep-packet-inspection, proxies, and/or data-logging, while NOX generally only inspects the first few hundred bytes in a flow. However, NOX-based management applications can incorporate these payload processing technologies by directing flows through the appropriate middleboxes. Thus, one shouldn't view NOX as a replacement for current network management techniques, but instead as a framework that can coordinate and manage these ever-advancing technologies.

[9] We are indebted to Brandon Heller for pointing out NOX's potential role in power management.

6 References

[1] OpenFlow Switch Consortium. http://www.openflowswitch.org/.

[2] M. Allman, K. Christensen, B. Nordman, and V. Paxson. Enabling an Energy-Efficient Future Internet Through Selectively Connected End Systems. In *HotNets-VI*, 2007.

[3] M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, and J. van der Merwe. Design and implementation of a routing control platform. In *NSDI '05*, 2005.

[4] Z. Cai, F. Dinu, J. Zheng, A. L. Cox, and T. S. E. Ng. Maestro: A Clean-Slate System for Orchestrating Network Control Components. under submission, 2008.

[5] K. L. Calvert, W. K. Edwards, and R. E. Grinter. Moving Toward the Middle: The Case Against the End-to-End Argument in Home Networking. In *HotNets-VI*, 2007.

[6] M. Casado, M. J. Freedman, J. Pettit, J. Luo,

N. McKeown, and S. Shenker. Ethane: taking control of the enterprise. In *SIGCOMM '07*, 2007.

[7] M. Casado, T. Garfinkel, M. Freedman, A. Akella, D. Boneh, N. McKeown, and S. Shenker. SANE: A Protection Architecture for Enterprise Networks. In *Usenix Security Symposium*, 2006.

[8] 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. In *ACM SIGCOMM Computer Communication Review*, 2005.

[9] C. Gunaratne, K. Christensen, S. Suen, and B. Nordman. Reducing the Energy Consumption of Ethernet with an Adaptive Link Rate (ALR). *IEEE Transactions on Computers*, forthcoming.

[10] B. Heller and N. McKeown. A comprehensive power management architecture. Work in progress, 2008.

[11] J. Jung, V. Paxson, A. Berger, and H. Balakrishnan. Fast portscan detection using sequential hypothesis testing. In *IEEE Symposium on Security and Privacy*, 2004.

[12] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, 2008.

[13] P. Newman, G. Minshall, and T. L. Lyon. IP switching - ATM under IP. *IEEE/ACM Trans. Netw.*, 6(2):117–129, 1998.

[14] J. Rexford, A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, G. Xie, J. Zhan, and H. Zhang. Network-Wide Decision Making: Toward A Wafer-Thin Control Plane. In *HotNets III*, 2004.