Abstract of "Policy Delegation and Migration for Software-Defined Networks" by Andrew DeBock Ferguson, Ph.D., Brown University, April 2014.

In today's networks, non-administrative users have little interaction with a network's control-plane. Such users can send probe traffic to develop inferences about the network's present state, yet they cannot directly contact the control-plane for answers because of security or privacy concerns. In addition to reading the control-plane's state, modern applications have increasing need to write configuration state as well. These applications, running in home, campus, and datacenter networks, know what they need from the network, yet cannot convey such intentions to the control-plane.

This dissertation introduces participatory networking, a novel platform for delegating read and write authority from a network's administrators to end users, or applications and devices acting on their behalf. Users can then work with the network, rather than around it, to achieve better performance, security, or predictable behavior. Our platform's design addresses the two key challenges: how to safely decompose control and visibility of the network, and how to resolve conflicts between untrusted users and across requests, while maintaining baseline levels of fairness and security.

We present a prototype implementation of participatory networking, structured as an API and controller for OpenFlow-based software-defined networks (SDNs). We call our controller PANE, and demonstrate its usefulness by experiments with four real applications (Ekiga, SSHGuard, ZooKeeper, and Hadoop), and its practicality through microbenchmarks. Furthermore, we develop a mechanical proof for a key portion of PANE, the first for an SDN controller.

Unfortunately, network administrators interested in using SDN controllers such as PANE to manage the network face the herculean challenge of migrating existing policy to the new platform. To lessen this challenge, this dissertation introduces Exodus, the first tool for directly translating existing network configurations in languages such as Cisco IOS and Linux iptables to SDN controller software. These controllers are written in Flowlog, a novel, rule-based, tierless language for SDNs we significantly enhance for Exodus.

Automatic migration of existing configurations into SDN controllers has exposed several limitations in both today's languages for SDN programming, and OpenFlow itself. This dissertation explores these limits, and provides guidance on SDN migration and necessary switch features.

Abstract of "Policy Delegation and Migration for Software-Defined Networks" by Andrew DeBock Ferguson, Ph.D., Brown University, April 2014.

In today's networks, non-administrative users have little interaction with a network's control-plane. Such users can send probe traffic to develop inferences about the network's present state, yet they cannot directly contact the control-plane for answers because of security or privacy concerns. In addition to reading the control-plane's state, modern applications have increasing need to write configuration state as well. These applications, running in home, campus, and datacenter networks, know what they need from the network, yet cannot convey such intentions to the control-plane.

This dissertation introduces participatory networking, a novel platform for delegating read and write authority from a network's administrators to end users, or applications and devices acting on their behalf. Users can then work with the network, rather than around it, to achieve better performance, security, or predictable behavior. Our platform's design addresses the two key challenges: how to safely decompose control and visibility of the network, and how to resolve conflicts between untrusted users and across requests, while maintaining baseline levels of fairness and security.

We present a prototype implementation of participatory networking, structured as an API and controller for OpenFlow-based software-defined networks (SDNs). We call our controller PANE, and demonstrate its usefulness by experiments with four real applications (Ekiga, SSHGuard, ZooKeeper, and Hadoop), and its practicality through microbenchmarks. Furthermore, we develop a mechanical proof for a key portion of PANE, the first for an SDN controller.

Unfortunately, network administrators interested in using SDN controllers such as PANE to manage the network face the herculean challenge of migrating existing policy to the new platform. To lessen this challenge, this dissertation introduces Exodus, the first tool for directly translating existing network configurations in languages such as Cisco IOS and Linux iptables to SDN controller software. These controllers are written in Flowlog, a novel, rule-based, tierless language for SDNs we significantly enhance for Exodus.

Automatic migration of existing configurations into SDN controllers has exposed several limitations in both today's languages for SDN programming, and OpenFlow itself. This dissertation explores these limits, and provides guidance on SDN migration and necessary switch features.

Policy Delegation and Migration for Software-Defined Networks

by

Andrew DeBock Ferguson

B. S. E., Princeton University, 2008

Sc. M., Brown University, 2011

A dissertation submitted in partial fulfillment of the

requirements for the Degree of Doctor of Philosophy

in the Department of Computer Science at Brown University

Providence, Rhode Island

April 2014

This dissertation by Andrew DeBock Ferguson is accepted in its present form by the Department of Computer Science as satisfying the dissertation requirement for the degree of Doctor of Philosophy.

Date ______________ ______________________________
Rodrigo Fonseca, Director

Recommended to the Graduate Council

Date ______________ ______________________________
Shriram Krishnamurthi, Reader
Brown University

Date ______________ ______________________________
Jennifer Rexford, Reader
Princeton University

Approved by the Graduate Council

Date ______________ ______________________________
Peter M. Weber
Dean of the Graduate School

Acknowledgements

"The pleasure we found in working together made us exceptionally patient;
it is much easier to strive for perfection when you are never bored."
Daniel Khaneman

This dissertation is the result of five years of wonderful collaboration. The development and implementation of the ideas between these pages simply would not have been possible without the hard work, deep discussions, and shared excitement of all my co-authors: Rodrigo Fonseca, Arjun Guha, Betsy Hilliard, Shriram Krishnamurthi, Chen Liang, Tim Nelson, Jordan Place, and Michael Scheer. I owe them all a tremendous debt.

The success of these projects is also due to the technical and administrative staffs of the Brown Computer Science department, particularly Lauren Clarke, Jeff Coady, Mark Dieterich, Kathy Kirman, Dawn Reed, and Max Salvas. With our interest in experimental infrastrucutre, I suspect systems researchers pose a unique and difficult challenge for their departments' technical staff; nonetheless, Jeff and Max were always ready to satisfy my creative requests, and Mark at least feigned understanding when I brought down the department routers in the middle of the night. I truly appreciate all of their help.

A very special and important thank you goes to my labmates: Chen Liang, Jon Mace, Marcelo Martins, Jeff Rasley, Matheus Santos, Da Yu, and Ray Zhou. Thanks to them and the life they brought to the lab, it was okay to consider the systems lab "home" during the long stretches needed to realize these projects. Thanks, you guys.

Shriram Krishnamurthi and his CSCI 1730 (Programming Languages) course changed my life. I am surely not the first person to whom this has happened, and I certainly won't be the last. By introducing me to the power of type systems, functional programming, logic programming, and many other topics, I finally had the tools to effect the changes I knew needed to be made in today's networks. I have leaned heavily on his students

over the years: Spiros Eliopoulos, Arjun Guha, Ben Lerner, Tim Nelson, Joe Politz, Justin Pombrio, and Hannah Quay-de la Vallee. For three years, I have always sat within 15 feet of at least a few of them, and I am going to miss terribly the luxury of easy access to their wisdom.

I have so much to thank Jennifer Rexford for. As an undergraduate, Jen's introduction to computer networks was the only class in which I couldn't help but start the homework just as soon as it was released; she had made everything just too interesting! Later, she would write my letter of recommendation to grad school, despite a) my last minute request, and b) being trapped in the Bankok airport due to the December 2009 Thai protests. Since then, the instances of Jen's help to my development as a researcher have only continued to pile-up.

Critical amongst Jen's contributions was the early introduction to Frenetic. Learning from, talking with, debating with, drinking with, and collaborating with the wider Frenetic family – Carolyn Anderson, Marco Canini, Spiros Eliopoulos, Nate Foster, Mike Freedman, Arjun Guha, Rob Harrison, Nanxi Kang, Naga Katta, Chris Monsanto, Srinivas Narayana, Mark Reitblatt, Josh Reich, Jennifer Rexford, Cole Schlesinger, Alec Story, Dave Walker, and those to whom I apologize for forgetting – was truly instrumental to the success of these projects. I couldn't have done this without their contributions: in papers, in conversation, and on Github.

I also want to thank Peter Bodik and Srikanth Kandula for an incredibly maturing internship at Microsoft Research. While the paper we produced on guaranteeing job latency in data parallel clusters does not fit within this dissertation, the experience taught me the level of work required for these projects, and firmly connected to reality what I had read about Big Data and warehouse-scale computing.

Finally, I must thank my advisor, Rodrigo Fonseca, who, from the first day, gave me the freedom, support, and encouragement to pursue my intellectual curiosities. Sometimes, this lead to failure ... but a few times, it worked out, eventually bringing you, dear reader, this dissertation. From the first time we wrote a paper together, I knew for certain I was lucky to have Rodrigo as an advisor. I still am.

Contents

List of Figures

Chapter 1

Introduction

Thesis Statement Modern applications can benefit from read and write interaction with a network's control-plane, yet such interaction is currently unsupported due to security and fairness concerns. We design a practical and feasible platform for software-defined networks (SDNs) that addresses these challenges through policy delegation, and demonstrate its benefits. Furthermore, we can assist administrators migrating to these new networks by directly translating their existing network configurations to SDN controller software.

1.1 The Need for Delegated Network Management

Today's applications, whether running in datacenters, enterprise, campus, or home networks have an increasingly difficult relationship with the network. Networks are the shared fabric interconnecting users, applications, and devices, and fluctuating, unpredictable network performance and reliability create challenges and uncertainty for network administrators, application developers, and frustrated end-users alike. As a result, software developers, researchers, and administrators expend considerable effort to work *around* the network rather than work *with* the network: video conferencing applications constantly probe network bandwidth [11], overlay networks are used to re-route traffic [11], network paths are reactively reconfigured based on inferences [1], and humans are required to throttle heavy network loads in response to planned or unplanned shifts in traffic matrices. Using humans for network control, however, is no panacea, having been responsible for significant recent outages at both Github [35] and Amazon [2].

At a minimum, packet networks forward data, collect traffic statistics, and divide traffic based on addresses or other header fields. Increasingly, modern networks also provide additional services, often implemented via middleboxes, such as firewalling, compression, encryption, threat-detection, acceleration, and caching. Yet, all of these features are, for the most part, invisible to the applications passing traffic through them, or only available via rudimentary interfaces such as DSCP header bits.

With greater visibility into and control of the network's state, a conferencing application could request bandwidth for a video call, and learn via the network that, while only a guaranteed audio call is available now, it could reserve a video call in one hour. An intrusion detection script on a user machine could request that the *network* filter traffic from a specific source. An important RPC service could protect latency-sensitive flows from competing background traffic. Or, a MapReduce-style application could request bandwidth guarantees, or maximally disjoint paths, to improve performance of its shuffle phase.

Such examples suggest that an API should exist between the network's control-plane and its users, applications, and end-hosts. These principals need both *read* access, to learn the network's present and expected future conditions, and *write* access, to make independent configuration changes for their own benefit, and provide helpful knowledge, such as future traffic demands, directly to the network's control-plane.

In this thesis, we introduce the concept of *participatory networking*, in which the network provides a configuration API to its users, applications, and end-hosts, and present the design, implementation, and evaluation of the first practical participatory networking controller for an OpenFlow-enabled software-defined network.

In the absence of security guarantees and limited authorities, participatory networks would be places of anarchy. To be usable, such networks must provide isolation for their independent principals, preventing malicious users from hogging bandwidth, dropping packets, or worse. While isolation could be provided through network virtualization, we believe that is not always the right abstraction, as it hides the fundamentally shared aspect of networks.

By contrast, participatory networks reveal the conflicts between their principals, and directly expose the control-plane's view of the network. Principals can use this greater awareness to make better-informed decisions about how to use or reconfigure the network.

1.2 Software-Defined Networking

The recent development of Software-Defined Networks (SDNs) offers a platform to realize the vision of participatory networks [37, 62]. SDNs separate the logic that controls the network from its physical devices, allowing configuration *programs* to operate on a high-level, global, and consistent view of the network. This change has brought significant advances to datacenter and enterprise networks, such as high-level specification of access control [66] and QoS [53], safe experimentation [79], in-network load-balancing [93], and seamless VM migration [27].

In a traditional network, the control-plane, which determines how to transmit packets, is distributed across the network's switches. Each switch independently runs distributed algorithms (*e.g.*, Bellman-Ford) to determine the packet forwarding policy. By contrast, an SDN has a logically centralized controller, running on a commodity server platform, which calculates a forwarding policy (*e.g.*, with Dijkstra's algorithm). The controller implements this policy by programming the switches' data-planes, which process traffic at line-rate.

This programming is accomplished via a protocol such as OpenFlow [62], which provides a common abstraction of a switch's data-plane. The principal component of this abstraction is a sequence of prioritized "Match-Action" pairs, where matches list a set of packet header fields, and actions specify handling of the matching packets: transmit via a port, place in a queue, drop, etc.

1.3 Migrating to Software-Defined Networks

Software-defined networks also offer to simplify the management of enterprise networks, a notoriously challenging problem [8, 17, 18, 51, 85, 101]. SDNs ease the evolvability of the network by centralizing configuration and management, and enable the use of modern programming languages and verification techniques.

However, migrating from an existing, working network environment to an SDN presents a formidable hurdle [45]. Enterprises and administrators are familiar with, and quite likely depend upon, the specific behavior of existing configurations, and any SDN replacement will need to begin with identical behavior.

Unfortunately, these networks can be large and complex [51, 58, 103], with network behavior defined by myriad policies, usually specified for each individual device, in a variety of languages that configure distributed programs embedded in these devices. The scale and complexity of the aggregate behavior of these rules means the process of creating a controller program for an equivalent SDN is non-trivial. For example, Purdue's network was reported to have over 15,000 hosts, 200 routers, 1,300 switches, and 182 VLANs in 2011 [85]. Stanford's backbone, whose publicly available configuration we use in our evaluation (Chap. 9) [103], has two large border routers connected through 10 switches to 14 Operational Zone routers. In aggregate, the configuration comprises 757,000 forwarding entries and 1,500 ACL rules.

Today, there are a few options for incrementally migrating a physical network to an SDN [17, 20, 58], which we review later in the dissertation. However, a common deficiency of these migration paths is the need to rewrite network policies and configurations from scratch, a tall order for busy network operators [6]. For a risk-averse network administrator, this may be reason enough not to migrate.

This dissertation directly addresses the problem of migrating distributed network configurations to equivalent SDN controller programs, and presents Exodus, a system we developed for performing this conversion. The controllers Exodus generates are written in Flowlog [68], a language for SDN programming we designed out of our experience building the PANE controller.

The development of PANE, Flowlog, and Exodus has exposed a number of deficiencies in the current state of SDN development. We close with a discussion of these deficiencies relative to traditional networking (Chap. 10), and within the modern software-stack itself (Chap. 11).

1.4 Summary of Contributions

In summary, this thesis makes the following contributions:

1. We introduce the concept of participatory networking, an interface for Software-Defined Networks which allows administrators to safely delegate control of the network to end users and their applications [31].

2. We present a rich conflict-resolution formalism – hierarchical flow tables (HFTs) – which allows conflicting policies to coexist in the network. HFTs introduce hierarchical merge composition [29], which complements existing parallel [33] and sequential [64] composition.

3. We demonstrate, through a participatory networking prototype, four instances in which end-user applications benefit from read and write interaction with a network's control-plane [30].

4. We present and evaluate Exodus, the first framework for automatic migration of existing network configurations to equivalent Software-Defined Networks.

Chapter 2

Participatory Networking

An API for Application Control of SDNs

This chapter introduces the concept of *participatory networking*, in which the network provides a configuration API to its users, applications, and end-hosts, and offers an overview of the first participatory networking controller for a software-defined network. Later chapters will describe this design in more detail, and present its evaluation.

Our prototype OpenFlow-based controller for participatory networks is called PANE. The PANE controller implements a capability system – it delegates read and write authority, with optional restrictions, from the network's administrators to the users, or applications and hosts acting on their behalf. The controller is logically centralized, has a global view of the network, and implements the principals' high-level intents by changing the configuration of the actual network devices. In addition, we implement and evaluate the examples described above by augmenting four real applications (Chap. 6).

PANE's user-facing API serves as the next layer on the current SDN stack. The abstractions provided by software-defined networks allow us to reason formally about PANE's design, and ensure the network continues to provide baseline levels of fairness and security, even as principals dynamically invoke their capabilities.

Our design addresses the two key challenges of participatory networks: how to safely decompose control and visibility of the network, and how to resolve conflicts between participants and across requests. PANE's solutions to these challenges were developed both through formal reasoning, and by porting real-world applications to solve existing use cases. We start with an overview of our solution in this chapter, followed by in-depth discussions of each challenge in Chap. 4.

Many approaches to achieving some of these goals have been previously proposed including active networking [86], IntServ networking [13], and distributed reservation protocols such as RSVP [14] and NSIS [60]. We discuss their relation to participatory networking in Sec. 6.3. PANE does not introduce any new functionalities into the network. Instead, it exposes existing functionalities and provides a single platform for reasoning about their use. We argue that this approach provides several advantages: a single target for application developers, a unified management framework for network administrators, and, most importantly, the ability to reason across all network resources.

PANE is designed for networks within a single administrative domain including corporate WANs, datacenters, campus or enterprise networks, and home networks. In such networks, there is, logically, a single owner from which authority can be delegated. PANE's design does not rely on changes to the end-hosts, such as the use of a particular hypervisor, making it suitable for networks with user-owned or managed devices.

2.1 The PANE Controller

We first present an overview of PANE, including the model of interaction, the types of messages, and the kinds of network resources one can request. We discuss the challenges involved in exposing network control to multiple principals, and the solutions we propose. We then discuss additional considerations that influenced PANE's design, which we detail in the following chapters (3-5).

PANE allows principals to gain controlled visibility into the network and to safely influence network operations. *Principals* in PANE are end users, or, most commonly, applications and devices running on their behalf. We assume some infrastructure in the network for authentication, such as 802.1x associated with an existing user database. After authentication, principals interact with the PANE controller using a simple text-based protocol.

Principals can issue three types of messages to read and write network state: *requests* (Sec. 3.1), *queries* (Sec. 3.2), and *hints* (Sec. 3.3). Requests are for resources (*e.g.*, bandwidth or access control), with an action to be taken by the controller. Queries read some component of network state (*e.g.*, traffic between hosts, or available bandwidth). Hints inform PANE about current or future traffic characteristics; the controller may choose to use hints to improve service. Our initial design implements a first come-first serve service model, where the controller handles messages in a serialized fashion.

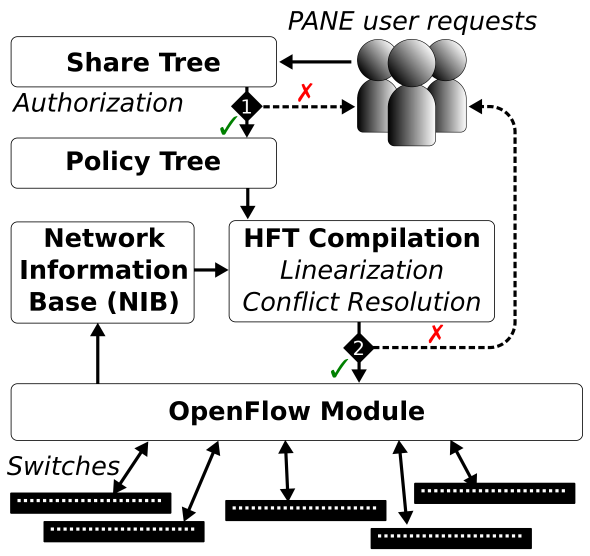


Figure 2.1: The PANE system and request processing

Each message refers to some subset of the network's flows; we call these *flowgroups*. For example, a message may request to deny all traffic from a particular host, or to rate-limit a single flow, or query the bandwidth used by some set of flows. Without restrictions, a malicious or negligent principal can adversely affect the network – a key novelty of PANE is its method to safely allow multiple principals to affect the network, without ceding administrative privileges.

As simple examples, a firewall program in a user's machine may issue a request for denying all traffic from external IP addresses to itself, to be implemented at the edges of the network; a distributed RPC library may issue a hint for the network informing that all flows between two ports will be small flows; or a backup application may query the network about the current buffer utilizations along a path.

PANE's design addresses two key challenges. First, it provides a flexible mechanism that gives multiple principals control over a fine-grained portion of network resources. Second, it resolves the inevitable conflicts that arise between principals' requests, while allowing oversubscription.

Limiting Authority PANE uses *shares* to limit the authority of principals. A share states *who* (which principals) can say *what* (which messages) about *which* flows in the network. This statement is represented, respectively, by a share's three components: its *principals*, *privileges*, and *flowgroup*. Figure 4.1(a) shows an example share. Principals of a share have two implicit privileges. A principal can delegate its privileges to another principal, much like passing an object capability. In addition, principals can create sub-shares of shares to which they have access. Shares are thus organized in a global *share tree*. The share tree enforces two key invariants: a sub-share's

```
1 root: NewShare aliceBW for (user=Alice) [reserve <= 10Mb] on rootShare.
2 root: Grant aliceBW to Alice.
3 Alice: reserve(user=Alice,dstPort=80) = 8Mb on aliceBW from +20min to +30min.
4 root: NewShare bobAC for (dstHost=10.0.0.2) [deny = True] on rootShare.
5 root: Grant bobAC to Bob.
6 Bob: deny(dstHost=10.0.0.2, srcHost=10.0.0.3) on bobAC from now to +5min.
7 Bob: deny(dstHost=10.0.0.4, srcHost=10.0.0.3) on bobAC.
```

Figure 2.2: Sample interaction between three principals and PANE.

flowgroup must be a subset of its parent's flowgroup, and a sub-share's privileges cannot be more permissive than its parent share's privileges.

Figure 2.2 traces an example interaction in which the root user creates a new share, `aliceBW` restricted to Alice's traffic (Line 1). This share carries the privilege to reserve up to 10 Mbps of guaranteed minimum bandwidth, and is a sub-share of the root share. In Line 2, the root user grants Alice access to this share. Alice then uses this share to reserve bandwidth for her HTTP (port 80) flows for 10 minutes, starting 20 minutes in the future (Line 3). In the next line, the root user creates a share for traffic destined to Bob's computer (`10.0.0.2`) with the privilege to deny traffic, and subsequently grants Bob access to this share (Line 5). In Line 6, Bob successfully uses this share to block access from a host `10.0.0.3` for five minutes. However, his attempt to block traffic from that host to `10.0.0.4` is rejected as the specified flow group is not a subset of the share's flowgroup (`dstHost=10.0.0.2`).

Resolving Conflicts The share tree constrains the policies that can be realized in the network, but does not itself cause any policy to be implemented in the network. Instead, accepted requests and realized hints determine network policy. We call such accepted requests and realized hints *policy atoms* – units of the overall network policy. Policy atoms are arranged in the same hierarchy as the share tree, forming a *policy tree*. A policy tree is a declarative data structure that represents the desired global policy for the network. PANE materializes this policy in the network by installing rules in the switches that implement an equivalent policy (Chap. 5).

Policy atoms thus exist in the context of a share, and are bound by the shares' privileges and flowgroup. However, policy atoms may conflict. For example, one policy atom may deny all HTTP flows, while another allows HTTP flows. These atoms may even exist on different shares. The PANE share tree is flexible: it supports oversubscription, and allows several shares to express policies for overlapping flowgroups. A key novelty of PANE is a principled and intuitive *conflict-resolution* algorithm for hierarchical policies.

We develop Hierarchical Flow Tables (HFTs) to materialize PANE's policy tree. HFTs provide a model for resolving conflicts in a hierarchy of policies, and a formally-verified compiler from such hierarchies to flow tables suitable for OpenFlow switches. In particular, HFTs use *conflict resolution operators* within and between each node in the hierarchy to flexibly resolve conflicts. We describe the design of PANE's operators, and the semantics and implementation of HFTs in Sec. 4.2.

Request Processing Having summarized PANE's key ideas, we now describe at a high level the processing of a single request, as depicted in Figure 2.1. When an authenticated principal sends the controller a message, perhaps requesting a resource for a flowgroup in a particular share, PANE first checks that the request is admissible per the share's flowgroup and privileges – Check 1 in the figure.

If this first check passes, PANE then checks to see if it is compatible with the state of the network – Check 2. This check involves all accepted requests (*i.e.*, policy atoms) in the policy tree, and the physical capabilities of the network. For example, a bandwidth reservation requires a circuit between two endpoints with sufficient bandwidth and switch queues. This check requires compiling the current policy tree, augmented with the request itself. If this check passes, the request is incorporated into the tree, and the controller can install the policy onto the network. This process also has a variation which only partially fulfills requests; Sec. 4.2.4 describes both variations in more detail.

A final key feature, which we detail in subsequent chapters, is that PANE allows principals to request resources for future intervals. To support this, PANE maintains a time-indexed sequence of policy trees. The above checks may thus be made against future, planned network state as appropriate.

Chapter 3

Interacting with PANE

We now expand upon the three message types introduced in the overview: requests, queries, and hints. Table 3.1 has a concise specification of these messages, and their relation to other key concepts in PANE.

3.1 Requests

A *request* affects the state of the network for some interval of time. By default, requests take effect immediately and do not expire; specific start and end times may optionally be provided. Verifying if a request can be granted may require walking the tree's hierarchy, depending on the type of request. This design allows resources to be oversubscribed; overallocation is prevented when requests are granted, and not when shares are created.

Participatory networks may support requests for a variety of network resources and services, which we detail next.

Access Control The simplest type of network service exposed by PANE is access control – the ability to allow and deny traffic, using the **Allow** and **Deny** requests. Like all requests, they specify a flowgroup describing the affected traffic, and the share which the principal is using to invoke the privilege. Each access control privilege is optionally constrained by a specified number of seconds, n. To exceed this limit, principals must periodically renew requests. Shares lacking the ability to allow or deny traffic have $n = 0$. When creating a sub-share, a principal cannot exceed these constraints. For example, if a share carries the privilege to **Deny** traffic for up to 300 seconds, a sub-share cannot be created with the privilege to **Deny** traffic for up to 301 seconds; similarly, a sub-share could not be created with the privilege to **Allow** traffic.

The handling of a given packet is ultimately decided by the composition of every matching access control

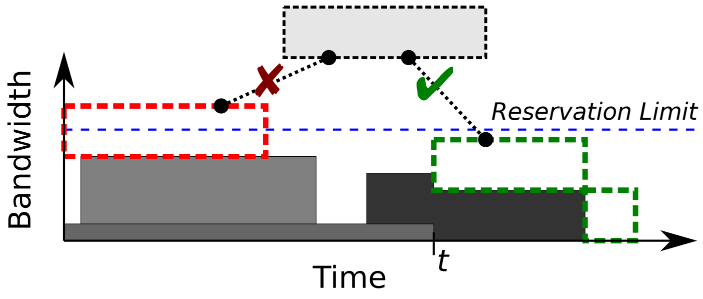


Figure 3.1: Example user request for reserved bandwidth; PANE determines that it cannot be fulfilled until time t.

request. This composition makes use of the share tree's hierarchy to resolve conflicts – for example, an access control request made on a child share overrides those in parent shares. We defer further discussion of PANE's general approach to conflict resolution until Sec. 4.2.

With each request, the principal can specify a fulfillment mode, either *strict* or *partial*. These are provided for atomicity and convenience. In strict mode, PANE rejects a request if it would be (partially) overridden by any previous request. For example, if a user wants to allow connections to TCP ports 1000-2000, but there exists a request in a sub-share that denies port 1024, PANE rejects the request, explaining why. In partial mode, PANE implements the request, and informs the user that it was only partially satisfied; in the same example, PANE would inform the user that it has allowed ports 1000-1023, and 1025-2000.

These modes exist for two reasons: first, to avoid race conditions in request allocations, and second, to avoid complicated, fine-grained specifications that depend on PANE's current state. We defer a more complete discussion of the strict and partial fulfillment modes until Sec. 4.2.4.

Guaranteed Minimum Bandwidth PANE also provides a **Reserve** privilege which provides guaranteed minimum bandwidth (GMB) between two hosts. Shares which contain the privilege to reserve bandwidth are limited by a modified token bucket: it has the usual attributes of fill rate F, capacity C, and maximum drain rate M, and an additional minimum drain rate m. This lower bound prevents reservations with very low drain rates that could last indefinitely. A simple reservation with maximum bandwidth B is a special case with $F = M = B; C = m = 0$. GMB reservations are ultimately implemented by PANE's runtime as a sequence of forwarding actions and switch queues, as we describe in Chap. 5. Requests which cannot be implemented are rejected.

Figure 3.1 shows a simple example in which a principal has requested an immediate bandwidth reservation. PANE determines that granting the request will exceed the share's available bandwidth. The principal then examines the share's schedule of available bandwidth and sends a new request for a reservation to start at *t*; PANE accepts the request and later implements it.

When creating sub-shares of shares with GMB privileges, the sub-share's token bucket must "fit inside" the parent's token bucket; parents cannot provide more tokens to their children than they receive. However, a share's tokens can be over-subscribed by its sub-shares. When a request is granted, it draws tokens from all of its parent shares, up to the root of the tree, thus preventing over-allocation.

Path Control A third request type directs flows through or around middleboxes using **Waypoint** and **Avoid**. For example, a university's network administrators can route students' traffic through a packet shaper during business hours, and security researchers can avoid intrusion detection systems for traffic to be collected by honeypots. Shares contain sets of IP addresses listing the middleboxes which they can route through or avoid, and, as with flowgroups, sub-shares may only contain subsets of their parents' sets. PANE implements **Waypoint** and **Avoid** by installing flow-specific forwarding rules along a path determined by fixing or deleting nodes as appropriate when routing over the network graph (Sec. 5.1). Requests to create unrealizable paths are rejected.

Rate-limits PANE supports rate-limit requests which result in matching traffic being routed through ports with established rate-limiters, as available in current switches. While basic, such requests can be used to mitigate DOS attacks or enforce traffic contracts between tenants in a shared datacenter. PANE's global view of the network enables it to make best use of the switches' features and place rate-limiters close to the traffic's source, as we describe in Sec. 5.1. Like PANE's bandwidth reservations, rate-limits are currently restricted to circuits; a network with distributed rate-limiters, such as those proposed by Raghavan, *et al.* [73], could support more general limits. Their integration is left as future work.

3.2 Queries

PANE also supports messages to query the state of the network. These queries may be for general information about the network, such as the type of a link (*e.g.*, copper or optical), the set of hosts located downstream of a particular port, or other properties. Each share may contain a list of properties which it is privileged to read. This list is similar to a "view" on a database; when sub-shares are created, this view may be further occluded. While these restrictions provide basic privacy protection when exposing the network's state, they are not complete. For example, if a switch has three links, and a principal has the privilege to read the sending and receiving rates on two of the links, but not the third, it can infer the rate on the third link. We leave a more complete development of privacy protections as future work.

The current OpenFlow specifications and design make a number of properties available which principals in PANE may query including: the number (or list) of hosts behind a particular port, port-specific diagnostic data such as the number of packets dropped, the number of CRC errors, etc., the physical and topological location of switches, and the access medium of links. In the future, we would like to support additional details we believe would benefit applications such as the current signal-to-noise ratio or broadcasting power of wireless access points.

PANE also supports a "network weather service" which provides coarse information about current traffic conditions. For example, statistics about the total amount of traffic over a core link are available, but not statistics about individual flows. By integrating PANE with a project like Frenetic [33], we expect it could support the ability to query the traffic statistics of individual flows, as such queries require a more robust OpenFlow runtime than our current implementation.

Applications can issue queries to the PANE controller to improve the user experience. For example, Hadoop could use the weather service to place reducers away from currently-congested parts of the network, and streaming video players can determine that a wireless access point is attached to a cellular modem or similarly constrained backhaul as Shieh, *et al.* proposed [80].

3.3 Hints

The final type of message in PANE is a hint. Hints are used to provide the network with information which may improve the application's or network's performance, without creating an additional requirement. Providing hints across abstraction boundaries is a natural feature in other systems. For example, if a three-tier web application makes a request to the database layer, and hints to the caching layer that it won't make the same request again in the near future, the cache may choose to skip storing the result.

Three hints which are useful for networked applications include: the size (in bytes) of a flow, a desired flow-completion deadline, and the predictability of future traffic. PANE can use flow size information to spread large flows across multiple paths of equal-cost, as in Mahout [24] or Hedera [1]. Deadlines can be communicated to supporting routers such as those proposed in D3 [94]. And hints about traffic predictability can be used by traffic optimizers such MicroTE [10].

PANE *may* use hints to derive and install policy atoms which affect related traffic, although it gives no guarantee or notification to the user. For example, a hint that a flow is short may generate a policy atom to increase that flow's priority. We call such hints *realized*, and their corresponding policy atoms are tagged as merely hints (cf. Table 3.1).

The integration of hints, which can benefit non-PANE systems, as in the examples above, is deliberate. PANE provides a network administrator with the framework to delegate privileges, divide resources, and account for their usage; the ability to issue hints is a privilege, particularly those which affect limited resources. The framework provided by PANE makes it more feasible to implement hints in an untrusted environment, where malicious principals may issue false or excessive hints in an attempt to gain an advantage.

Finally, in the absence of transactional-style requests (*e.g.*, a request for "resource A *or* resource B"), PANE's hints are a more flexible way to provide information to the network than via requests. In this use, hints share a similar role to PANE's partial fulfillment mode for requests (Sec. 4.2.4).

Share	S	$\in \{P\} \times \{F\} \times \{Priv\}$	A share gives principals some privileges to affect a set of flows.
Principal	P	$::= (\textbf{user}, \textit{host}, \textbf{app})$	A triple consisting of an *application*, running on a *host* by a *user*.
Flow	F	$::= \langle \textbf{srcIP}{=}n_1, \textbf{dstIP}{=}n_2,$	A set of packets with shared properties: source and destination IP address,
		$\textbf{proto}{=}n_3, \textbf{srcPort}{=}n_4, \textbf{dstPort}{=}n_5 \rangle$	transport protocol, and source and destination transport ports.
Privilege	$Priv$	$::= \textbf{CanDeny}\ n \mid \textbf{CanAllow}\ n$	The privileges to allow or deny traffic for up to n seconds (optional).
		$\mid \textbf{CanReserve}\ n \mid \textbf{CanRateLimit}\ n$	The privileges to reserve bandwidth or set rate-limits, up to n MB.
		$\mid \textbf{CanWaypoint}\ \{IP\} \mid \textbf{CanAvoid}\ \{IP\}$	The privileges to direct traffic through or around particular IP addresses.
Message	Msg	$::= P : \{F\} : S \rightarrow (Req\ Tspec \mid Hint\ Tspec \mid Query)$	A message from a principal with a request, hint, or query using a share.
Time Spec	$Tspec$	$::= \textbf{from}\ t_1\ \textbf{until}\ t_2$	An optional specification from time t_1 until t_2.
Request	Req	$::= \textbf{Allow} \mid \textbf{Deny}$	Request to allow/deny traffic.
		$\mid \textbf{Reserve}\ n \mid \textbf{RateLimit}\ n$	Request to reserve n MB or rate-limit to n MB.
		$\mid \textbf{Waypoint}\ IP \mid \textbf{Avoid}\ IP$	Waypoint/avoid traffic through a middlebox with the given IP address.
Query	$Query$	$::= \textbf{TrafficBetween}\ srcIP\ dstIP \mid \ldots$	Query the total traffic between two hosts.
Hint	$Hint$	$::= \textbf{Duration}\ t \mid \ldots$	Hint that the flow's duration is t.
Policy Atom	$Atom$	$::= P : \{F\} \rightarrow Req\ Tspec$	A requested modification of network state.
		$\mid\ \textbf{Hint}\ P : \{F\} \rightarrow Req\ Tspec$	A realized hint; it may be removed if it conflicts with a future request.

Table 3.1: Main concepts in PANE

Chapter 4

The Two Challenges

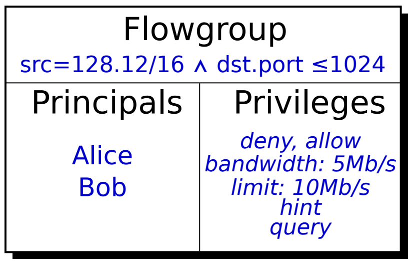
Participatory networking faces two key challenges to its realization. The first is how to safely decompose control and visibility of the network, and the second is how to resolve conflicts between participants and across requests. We now explain how PANE's design overcomes these challenges.

4.1 Privilege Delegation

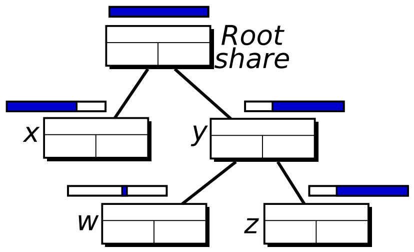
This section presents the semantics of shares and how principals' messages are authorized in more detail. The PANE controller maintains two key data structures. First, the *share tree* determines the privileges that principals have to read or write network state. The tree-structure allows principals to create new shares and delegate authority to each other. The share tree itself does not affect the state of the network. Instead, the second key data-structure, the *policy tree*, holds *policy atoms* that can affect the network. PANE maintains the invariant that all policy atoms in the policy tree are properly authorized by the share tree at all times.

A share-tree is an n-ary tree of *shares*, where a share gives a set of *principals* some *privileges* to affect a set of *flows* in the network. We elaborate on these terms below.

Principals A PANE principal is a triple consisting of an application running on a host by a user. A principal may be (**Skype**, *192.168.1.7*, **Alice**) or (**Hadoop**, *10.20.20.20*, **Bob**) for example. Shares in PANE are held by principal-sets. We abbreviate singleton sets to their principal. We also use wildcards to denote large sets. *e.g.*, (**Alice**, $\star$, $\star$) is the set of all principals with Alice as the user, and ($\star$, $\star$, **Hadoop**) is the set of all principals with Hadoop as the application. We write ($\star$, $\star$, $\star$) to denote the set of all principals.



(a)



(b)

Figure 4.1: (a) A PANE share. (b) A share hierarchy. The rectangle above each share represents a flowgroup according to one dimension (*e.g.*, source IP). Sub-shares are defined on a subset of their parent's flowgroup, and may not have more permissive privileges than their parent.

Principals send messages to the PANE controller to request resources and query the state of the network. For example, the principal (**Skype**, *192.168.1.7*, **Alice**) may request low-latency service between the Skype call's source and destination, and the principal (**Hadoop**, *10.20.20.20*, **Bob**) may request guaranteed bandwidth between the three machines in an HDFS write pipeline, as we implement in Sec. 6.1.

In a deployed system, PANE could use 802.1x to authenticate the user portion of a principal against an existing user database such as Active Directory or LDAP. In an ideal environment, the application and host portions could be attested to by a TPM module and application signatures on the end host [82]. For now, our prototype only considers the user portion of a principal.

The three-part principal design allows users and network administrators to fully understand the provenance of each request. For example, in a cluster of Hadoop machines, requests by different Application Masters are identifiable back to the specific machine they were made from. Similarly, users can differentiate between requests from distinct applications on the same machine.

Flows A flow is a set of related packets on which requests are made. For example,

$$\langle \textbf{srcIP}{=}w, \textbf{dstIP}{=}x, \textbf{proto}{=}\mathit{TCP}, \textbf{srcPort}{=}y, \textbf{dstPort}{=}z \rangle$$

is a flowgroup that denotes a TCP connection from $w : y$ to $x : z$. A PANE share allows principals to affect a set of flows, which we denote with wildcards when possible. For example, the following flowgroup denotes all HTTP requests:

$$\langle \textbf{srcIP}{=}\star, \textbf{dstIP}{=}\star, \textbf{proto}{=}\mathit{TCP}, \textbf{srcPort}{=}\star, \textbf{dstPort}{=}80 \rangle$$

whereas the following denotes HTTP requests and responses:

$$\langle \textbf{srcIP}{=}\star, \textbf{dstIP}{=}\star, \textbf{proto}{=}\mathit{TCP}, \textbf{srcPort}{=}\star, \textbf{dstPort}{=}80\rangle \cup$$
$$\langle \textbf{srcIP}{=}\star, \textbf{dstIP}{=}\star, \textbf{proto}{=}\mathit{TCP}, \textbf{srcPort}{=}80, \textbf{dstPort}{=}\star\rangle$$

A key invariant of the share tree is that if share S_1 is a sub-share of share S_2, then S_1's flowgroup is a subset of S_2's flowgroup. Therefore, sub-shares allow principals to implement fine-grained delegation of control.

Privileges Privileges in PANE define the messages principals may send using the share. Each message type, as described in the previous chapter, has a corresponding privilege. For example, **CanAllow** n and **CanDeny** n permit admission-control policies to be requested for n seconds, and **CanWaypoint** $\{IP\}$ indicates that principals can route traffic through an IP address in the given set.

4.2 Conflict Resolution

Conflicts arise naturally in a participatory network, as PANE is designed to allow multiple, distributed principals to author the network configuration. For example, one principal may issue a request to deny traffic to TCP port 80, while another may request such traffic be allowed. This section discusses how PANE handles conflicts between overlapping requests through the introduction of Hierarchical Flow Tables (HFTs).

Two requests overlap when the intersection of their respective flowgroups is not empty, *i.e.*, there are some flows that match both. As described in Chap. 2, principals make requests in the context of a share, and accepted requests become policy atoms residing in this share. Policy atoms, then, inherit from the share tree a natural hierarchical relationship, which we call the *policy tree*. The network's effective policy is a function of the set of all policy atoms, their position in the tree, and the semantics of conflict resolution between overlapping policy atoms.

We now develop a detailed semantics for HFTs (Sec. 4.2.1), describe a compiler which translates HFTs for use in OpenFlow-based SDNs (Sec. 4.2.2), detail PANE's choice of conflict-resolution operators (Sec. 4.2.3), the fulfillment of *strict* and *partial* requests, (Sec. 4.2.4), and finally, analyze the complexity of the HFT compiler (Sec. 4.2.5).

$$\begin{array}{llcl}
 & H & = & \text{header names and ingress ports} \\
\text{patterns} & V & = & \mathit{const} \mid \mathit{prefix} \mid \star \\
\text{matches} & M & = & \varnothing \mid \langle \overrightarrow{H, V} \rangle \\
\text{actions} & A & = & \textbf{Allow} \mid \textbf{Deny} \mid \textbf{Reserve}(n) \mid \textbf{RateLimit}(n) \mid \mathbf{0} \\
\text{conflict-resolution operators} & (+) & = & A \to A \to A \\
\text{policy atoms} & P & = & M \times A \\
\text{policy tree nodes} & D & = & (+_D) \times 2^P \\
\text{policy trees} & T & = & (+_P) \times (+_S) \times D \times 2^T \\
\text{packets} & K & = & \langle \overrightarrow{H, \mathit{const}} \rangle
\end{array}$$

$\boxed{cmb : D \times K \to A}$

$$\begin{array}{c}
cmb((+, \{\cdots(M_i, A_i)\cdots\}), K) = A'_1 + \cdots + A'_k + \mathbf{0} \\
\text{where} \quad \{A'_1, \cdots, A'_k\} \quad = \quad \{A_i | M_i \cap K \neq \varnothing\}
\end{array}$$

$\boxed{eval : T \times K \to A}$

$$\begin{array}{l}
eval((+_P, +_S, D, \{T_1, \cdots, T_n\}), K) = cmb(D, K) +_P A_1 \\
\begin{array}{lrcl}
\text{where} & A_1 & = & eval(T_1, K) +_S A_2 \\
 & A_2 & = & eval(T_2, K) +_S A_3 \\
 & \cdots & & \\
 & A_n & = & eval(T_n, K) +_S \mathbf{0}
\end{array}
\end{array}$$

Figure 4.2: Semantics of HFT[2]

4.2.1 Semantics of HFT

A Hierarchical Flow Table allows several principals to author a tree of policies, and specify custom conflict-resolution operators at each node in the tree. In this section, we define the semantics of a policy tree as the final action it produces on an individual packet, after it has consolidated actions from all policies in the tree.[1] In Sec. 4.2.2, we compile these policy trees to run efficiently on hardware.

Figure 4.2 defines packets (K), policy trees (T), actions (A), and a function *eval* that matches packets against policy trees and returns an action. For our purposes, packets are a vector of header names and values; we do not match on packets' contents. For concreteness, we depict the actions we have implemented in our prototype (Chap. 5): admission control, reserving a guaranteed minimum bandwidth (GMB), rate-limiting bandwidth, and $\mathbf{0}$, a special "don't care" action.

A policy tree is a tree of policy nodes (D), which contain sets of policy atoms (P). An atom is a match rule and action pair, (M, A). When a packet matches a policy atom, $M \cap K \neq \varnothing$, the atom produces its action. The interesting cases occur when a packet matches several policy atoms with conflicting actions. In these cases, we

[1]This semantic model, where the central controller conceptually sees all packets, is inspired by Frenetic [33].

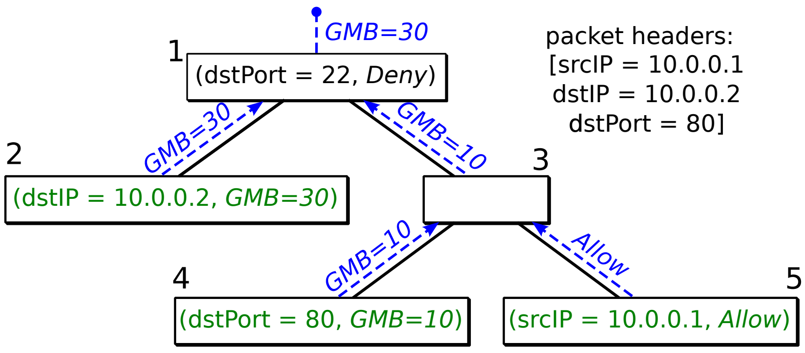


Figure 4.3: Evaluation of a single packet

resolve conflicts with the conflict-resolution operators (+) attached throughout the policy tree.

Policy trees have different types of conflict-resolution operators at several points in the tree (*i.e.*, $+_D$, $+_P$, $+_S$ in Figure 4.2). These multiple types allow an HFT to resolve different types of conflicts using independent logic. For example, conflicts between parent and child nodes may be resolved differently than conflicts between a single node's internal policy atoms. Therefore, the choice of conflict-resolution operators is a key policy decision. Our prototype network (Chap. 5) provides two default operators; developing and evaluating additional operators, such as operators to support priorities across requests, is left as future work.

The function *cmb* matches a packet with an individual policy tree node. If a packet matches several policy atoms, *cmb* uses the node's internal conflict-resolution operator, $+_D$, to combine their actions. The compiler requires $+_D$ to be associative and have $\mathbf{0}$ as its identity.[3]

The function *eval* matches a packet with a policy tree by applying *cmb* to the policy tree node at the root, and recursively applying *eval* to its children. A policy tree has conflict-resolution operators $+_P$ and $+_S$, which respectively allow it to resolve parent-child and inter-sibling conflicts differently. In particular, $+_P$ does not have to be commutative – it is always used with the parent's action on the left and the child's action on the right. This lets us express intuitive conflict resolutions such as "child overrides parent."

Example: Figure 4.3 depicts a simple policy tree and illustrates how *eval* produces an action, given the tree and indicated packet. Each node contains its policy atoms, and atoms which match the packet are colored green. The *eval* function recursively produces an action from each sub-tree; these actions are the labels on each node's outgoing edge.

In this example, the policy atoms at each leaf match the packet and produce an action. Node 3 receives

[2] $2^{M \times A}$ is the set of all subsets of pairs drawn from M and A.

[3] That is, we require $a + (b + \mathbf{0}) = (a + b) + \mathbf{0} = a + b$.

conflicting actions from its children, which it resolves with its inter-sibling conflict-resolution operator: $\mathbf{Reserve}(10) +_S \mathbf{Allow} = \mathbf{Reserve}(10)$. Node 3 has no policy atoms itself, so it produces the $\mathbf{0}$ action. Since $\mathbf{0}$ is the identity of all conflict-resolution operators, $\mathbf{0} +_P \mathbf{Reserve}(10) = \mathbf{Reserve}(10)$ is the resulting action from this sub-tree.

Finally, Node 1 computes the aggregate action of its children: $\mathbf{Reserve}(30) +_S \mathbf{Reserve}(10) = \mathbf{Reserve}(\max(30, 10))$. Since Node 1's policy atoms do not match the packet, the final action is $\mathbf{0} +_P \mathbf{Reserve}(30) = \mathbf{Reserve}(30)$.

4.2.2 Compiling Policies

The preceding section assumes that a central function, *eval*, observes and directs all packets in the network. Although *eval* specifies the meaning of policy trees, this is not a practical implementation. We now describe how to compile HFT's policy trees to run on commodity switches, which support simpler, linear flow tables, to produce a practical implementation.

Our compiler works in two stages. First, we translate policy trees to *network flow tables*, which have a basic, linear matching semantics. Second, we use network flow tables to configure a distributed network of switches, translating high-level actions such as $\mathbf{GMB}(n)$ to low-level operations on switches (Sec. 5.1).

A network flow table (N) is a sequence of paired match rules and actions. The *scan* function, defined in Figure 4.4, matches packets against network flow tables and returns the action associated with the first matching rule. If no rules match the packet, then *scan* returns $\mathbf{0}$.[4]

The matching semantics of network flow tables correspond to the matching semantics of switch flow tables exposed by OpenFlow. When a packet matches a switch flow table, only one rule's action applies. If a packet matches multiple rules, the switch selects the one with the highest priority. A rule's index in a network flow table corresponds to a switch flow table priority, with index 0 as the highest priority. Since all rules have distinct indices, a naive correspondence would give all rules distinct priorities. A more compact one, which we use, maps a sequence of non-overlapping network flow table rules to a single priority in a switch flow table.

The lin_T function is our compiler from policy trees to network flow tables. It uses lin_D as a helper to compile policy tree nodes. The lin_D function translates policy atoms to singleton network flow tables, and combines them with $union(+, N, N')$. *Union* builds a network flow table that matches packets in either N or N'. Moreover, when a packet matches both N and N', *union* computes the intersection using the $+$ conflict-resolution operator to combine actions.

[4]The *scan* function is derived from NetCore [63].

$$\text{Network Flow Tables} \quad N \quad = \quad \langle \overrightarrow{M, A} \rangle$$

$\boxed{scan : N \times K \to A}$

$$\frac{M_1 \cap K = \varnothing \cdots M_{j-1} \cap K = \varnothing \qquad M_j \cap K \neq \varnothing}{scan(\langle (M_1, A_1) \cdots (M_n, A_n) \rangle, K) = A_j}$$

$$\frac{M_1 \cap K = \varnothing \cdots M_n \cap K = \varnothing}{scan(\langle (M_1, A_1) \cdots (M_n, A_n) \rangle, K) = \mathbf{0}}$$

$\boxed{lin_D : D \to N}$

$$\begin{aligned} & lin_D\left(+_D, \{M_1, A_1, \cdots, M_j, A_j\}\right) = N_1 \\ & \text{where} \quad N_1 = union(+_D, \langle M_1, A_1 \rangle, N_2) \\ & \qquad\qquad \cdots \\ & \qquad\qquad N_j = union(+_D, \langle M_j, A_j \rangle, \langle\rangle) \end{aligned}$$

$\boxed{lin_T : T \to N}$

$$\begin{aligned} & lin_T\left(+_P, +_S, D, \{T_1 \cdots T_k\}\right) = union(+_P, lin_D(D), N_1) \\ & \text{where} \quad N_1 = union(+_S, lin_T(T_1), N_2') \\ & \qquad\qquad \cdots \\ & \qquad\qquad N_k = union(+_S, lin_T(T_k), \langle\rangle) \end{aligned}$$

$\boxed{union, inter : (+) \times N \times N \to N}$

$$\begin{aligned} & union((+), N_1, N_2) = inter((+), N_1, N_2) N_1 N_2 \\ & inter((+), \langle \cdots (M_i, A_i) \cdots \rangle, \langle \cdots (M_j', A_j') \cdots \rangle) = \\ & \qquad \langle \cdots (M_i \cap M_j', A_i + A_j')) \cdots \rangle \end{aligned}$$

Figure 4.4: Network Flow Tables

Similarly, lin_T recursively builds network flow tables for its subtrees, and calls lin_D on its root node. It applies *union* to combine the results, using $+_S$ and $+_P$ where appropriate.

The functions in Figure 4.4, lin_T, lin_D, *union*, and *inter* require the conflict-resolution operators to satisfy the following properties.

Definition 1 (Well-formed) *T is well-formed if:*

- *All conflict-resolution operators are associative, and*
- ***0** is the identity of all conflict-resolution operators.*

Proving the compiler correct requires the following key lemma, which states that all conflict-resolution operators distribute over *scan*.

Lemma 1 *For all +, N_1, and N_2, where **0** is the identity of +, $scan(union(+, N_1, N_2)) = scan(N_1) + scan(N_2)$.*

With this, we prove the compiler correct.

$$\boxed{+_P : A \times A \to A}$$

$$\begin{array}{llll}
\textbf{Deny} & +_P & \textbf{Allow} & = \textbf{Allow} \\
\textbf{Allow} & +_P & \textbf{Allow} & = \textbf{Allow} \\
A_P & +_P & \textbf{Deny} & = \textbf{Deny} \\
\textbf{Deny} & +_P & \textbf{Reserve}(n) & = \textbf{Reserve}(n) \\
\textbf{Reserve}(m) & +_P & \textbf{Reserve}(n) & = \textbf{Reserve}(\max(m,n)) \\
\textbf{Reserve}(m) & +_P & \textbf{Allow} & = \textbf{Reserve}(m) \\
\textbf{Allow} & +_P & \textbf{Reserve}(m) & = \textbf{Reserve}(n) \\
\textbf{Deny} & +_P & \textbf{Ratelimit}(n) & = \textbf{Ratelimit}(n) \\
\textbf{Ratelimit}(m) & +_P & \textbf{Ratelimit}(n) & = \textbf{Ratelimit}(\min(m,n)) \\
\textbf{Ratelimit}(m) & +_P & \textbf{Allow} & = \textbf{Ratelimit}(m) \\
\textbf{Allow} & +_P & \textbf{Ratelimit}(m) & = \textbf{Ratelimit}(n)
\end{array}$$

$$\boxed{+_S : A \times A \to A}$$

$$\begin{array}{llll}
\textbf{Deny} & +_S & A_2 & = \textbf{Deny} \\
\textbf{Reserve}(m) & +_S & \textbf{Reserve}(n) & = \textbf{Reserve}(\max(m,n)) \\
\textbf{Reserve}(m) & +_S & \textbf{Allow} & = \textbf{Reserve}(m) \\
\textbf{Ratelimit}(m) & +_S & \textbf{Ratelimit}(n) & = \textbf{Ratelimit}(\min(m,n)) \\
\textbf{Ratelimit}(m) & +_S & \textbf{Allow} & = \textbf{Ratelimit}(m)
\end{array}$$

The $+_S$ operator is commutative. We only show representative cases.

Figure 4.5: PANE's conflict-resolution operators

Theorem 1 (Soundness) *For all well-formed policy trees, T and packets, P, $eval(T, P) = scan(lin_T(T), P)$.*

We mechanize all our definitions and proofs using the Coq proof assistant [23].[5] ■

4.2.3 Conflict-resolution Operators in PANE

As discussed previously, HFTs resolve conflicts through the use of conflict resolution operators. These operators take two conflicting requests as input, and return a single resolved request. For example, a packet which matches policy atoms from **Reserve**(10) and **Reserve**(30) may be resolved to the higher guaranteed bandwidth, **Reserve**(30), as occurs at Node 1 in Figure 4.3.

The HFT design allows for complex conflict-resolution operators, and could support different operators at each node in the tree. However, for PANE we chose simple conflict-resolution operators in the interest of user and administrator understanding. Figure 4.5 specifies PANE's conflict-resolution operators. PANE's parent-child operator ($+_P$) specifies a "child overrides parent" policy for admission control. PANE's $+_S$ and $+_D$ operators are identical, and specify a "**Deny** overrides **Allow** policy" between siblings.

These operators' simple design is heavily influenced by PANE's first come-first serve approach to granting requests – for example, the operators do not consider the principal who made the request; each request is treated equally within its hierarchical context. However, by taking advantage of this design flexibility, operators which

[5]The complete proof is available with PANE's source code: `http://github.com/brownsys/pane`

resolve conflicts by using priorities could be introduced. Because such an approach would lead to previously accepted requests being preempted, the PANE controller would need to maintain a connection to each principal to provide preemption notifications. Avoiding this complexity is an additional benefit of PANE's current, simple approach.

Finally, it is important to note that PANE's conflict-resolution operators may drop previously realized hints to fulfill *strict* requests, detailed next, as needed.

4.2.4 Strict vs Partial Fulfillment

We now return to PANE's *strict* and *partial* modes of fulfillment, first introduced with the **Allow** and **Deny** privileges. In each mode, a request is first authenticated against the share tree, then, as shown in Figure 2.1, PANE verifies the resulting policy tree can be compiled to a valid network configuration. After this verification, the two modes differ.

In strict mode, PANE ensures that a request's specified action is the same as the action returned by HFT's *eval* function for all packets in the request's flowgroup – that is, no conflict resolution operator has changed the resulting action for any matching packets. More formally, when a request with match rule M and action A is added to a policy tree, yielding tree T, $\forall$ packets $K \in \{K | M \cap K \neq \varnothing\}, eval(T, K) = A$. If this condition does not hold, the request is rejected. In partial mode, the request is not subject to this check, and may even be relaxed – for example, a request for 30 Mbps of guaranteed bandwidth on a share with only 20 Mbps available will be relaxed to a request for 20 Mbps.

These modes are useful for three reasons. First, strict mode provides the principal with a guarantee that the request will be implemented in the network as specified. This is a limited form of change-impact analysis: *was the impact of my change on the network's configuration what I expected? If not, cancel the request.* We will expand PANE's ability to provide change-impact analysis in future work.

Second, partial mode improves support for concurrent requests, as at least a relaxed form of a partial request will succeed. Without this, a principal faces the risk of repeatedly crafting strict requests based on the network state at time t_0, only to have the request arrive at time $t_2 > t_0$ and conflict with a request accepted at time t_1, where $t_2 > t_1 > t_0$.

Finally, partial mode's ability to relax a request is a useful convenience. For example, if a principal has permissions which affect dozens of specific TCP ports in the range 1000-2000, yet not all of them, partial requests can be made for that range, and the requests would be relaxed to just the specific ports, freeing the principal from needing to specify the particular ports on each request.

Partial reservations, such as the 20 Mbps received of the 30 Mbps requested in the example above, are particularly useful as applications can use them to provide upper-bounds for transfer time. Although the faster reservation may have been preferred, the slower one still provides predictability to the end-user (and in either scenario, the actual bandwidth received by the transfer may be even higher than the guaranteed minimum). Such a use case is different from that for bandwidth hints; with hints, the principal does not know how the information will be used, if at all.

4.2.5 Compiler Complexity

To realize a policy tree in OpenFlow hardware, we have to compile it to flow tables for each switch. A direct implementation of the HFT algorithm produces flow tables of size $O(2^n)$, where n is the size of the policy tree. With two changes, we can greatly reduce the complexity: the modified algorithm yields flow tables of size $O(n^2)$ in $O(n^2)$ time. This section is an overview of our results.

OpenFlow flow tables are simple linear sequences of patterns and actions. A flow can match several, overlapping policy atoms in a policy tree and trigger conflict-resolution that combines their policies. However, in an OpenFlow flow table, a flow will only trigger the action of the highest-priority matching pattern.

For example, suppose the policy tree has two atoms with the following flowgroups:

$$\langle \textbf{srcIP}{=}X, \textbf{dstIP}{=}Y, \textbf{proto}{=}\texttt{tcp}, \textbf{srcPort}{=}\star, \textbf{dstPort}{=}\star \rangle$$
$$\langle \textbf{srcIP}{=}\star, \textbf{dstIP}{=}\star, \textbf{proto}{=}\texttt{tcp}, \textbf{srcPort}{=}\star, \textbf{dstPort}{=}\texttt{80} \rangle$$

Suppose flows that match the first flowgroup – all flows from X to Y – are waypointed through some switch, and that flows that match the second flowgroup – all HTTP requests – are given some bandwidth reservation. These two flowgroups overlap, thus a flow may be (1) waypointed with a reservation, (2) only waypointed, (3) only given a reservation, or (4) not be affected by the policy.

An OpenFlow flow table that realizes the above two-atom policy tree must have entries for all four cases. In general, such an approach generates all possible combinations given trees of size n — *i.e.* flow tables of size $O(2^n)$.

We make two changes to prune the generated flow table: (1) we remove all rules that generate empty patterns and (2) we remove all rules whose patterns are fully shadowed by higher-priority rules. The earlier algorithm is recursive, and we prune after each recursive call. It is obvious that this simple pruning does not affect the semantics of flow tables. However, a surprising result is that it dramatically improves the complexity of the algorithm.

The intuition behind our proof is that for sufficiently large policy trees, the intersections are guaranteed to produce duplicate and empty patterns that get pruned. To see this, note OpenFlow patterns have a bit-vector that determines which fields are wildcards. Suppose two patterns have identical wildcard bits and we calculate their intersection:

First, if the two patterns are identical, then so is their intersection. Of these three identical patterns, two get pruned. Second, if the two patterns are distinct, since their wildcards are identical, they exactly match some field differently. Thus, their intersection is empty and pruned.

If patterns have h header fields, there are only 2^h unique wildcard bit-vectors. Therefore, if a policy tree has more than 2^h policy atoms, it is assured that some intersections create empty or duplicate patterns that are pruned, thus thinning the number of generated rules as new policy atoms are considered.

Our full complexity analysis shows that when the number of policy atoms, n, is larger than 2^h, then the compilation algorithm runs in $O(n^2)$ time and produces a flow table of size $O(n^2)$. OpenFlow 1.0 patterns are 12-tuple, and our current policies only use 5 header fields. Therefore, on policies with more than 2^5 policy atoms, the algorithm is quadratic.

Updating Flow Tables

It is not enough for PANE to generate flow tables quickly. It must also propagate switch updates quickly, as the time required to update the network affects the effective duration of requests. The OpenFlow protocol only allows switches to be updated one rule at a time. A naive strategy is to first delete all old rules, and then install new rules. In PANE, we implement a faster strategy: the controller state stores the rules deployed on each switch; to install new rules, it calculates a "diff" between the new and old rules. These diffs are typically small, since rule-table updates occur when a subset of policy atoms are realized or unrealized.

Chapter 5

The PANE Controller

The complete PANE system integrates the previously described components into a fully-functioning SDN controller, as depicted in Fig. 2.1. It manages simultaneous connections with the network's principals and its switches. In this role, it is responsible for implementing both our participatory networking API, as well as the details of computing default forwarding routes, transmitting OpenFlow messages, and reacting to network changes such as switches joining and links failing. To accomplish these tasks, the PANE controller maintains three data structures: the share tree, a sequence of policy trees, and a *network information base* (NIB), described below.

We have developed a prototype PANE controller using Haskell and the Nettle library for OpenFlow [89]. Although we chose OpenFlow as our substrate for implementing PANE, participatory networking's design does not depend on OpenFlow. PANE could be implemented using other mechanisms to control the network, such as 4D [37], MPLS, or a collection of middleboxes.

A prototype release is available on Github, and we provide a virtual machine for Mininet-based evaluation on our website.[1] The release also includes a Java library which implements an object-oriented interface to PANE's text API.

The PANE controller is an entirely event-driven multicore program. The three primary event types are incoming PANE API messages, incoming OpenFlow messages, and timer events triggered by the start or finish of previously accepted requests or realizable hints.

API messages always specify a share on which they are operating. When a message arrives, the PANE

[1]`http://pane.cs.brown.edu`.

controller first uses the share tree to determine whether it is authorized, and then, for requests, whether it is feasible by consulting the policy trees, as described in the previous chapters.

When requests start and expire, the PANE controller compiles the new policy tree to a set of switch flow tables, translating high-level actions to low-level operations on individual switches in the network. For example, a **Reserve**(n) action becomes a circuit of switch queues and forwarding rules that direct packets to those queues. As we will describe next, PANE's runtime uses its NIB and a default forwarding algorithm to realize this and other actions. Our implementation constructs a spanning tree and implements MAC learning as its forwarding algorithm.

When possible, PANE uses the slicing extensions to OpenFlow 1.0 to create and configure queues, and out-of-band commands when necessary. While OpenFlow allows us to set expiry timeouts on flow table entries, our controller must explicitly delete queues when reservations expire.

As Reitblatt, et al. [74] articulate, it is a challenge to update switch configurations without introducing inconsistent, intermediate stages. Our implementation does not presently address this issue; we anticipate confronting this problem in the future.

5.1 PANE's Network Information Base

A network information base (NIB) is a database of network elements – hosts, switches, ports, queues, and links – and their capabilities (*e.g.*, rate-limiters or per-port output queues on a switch). The runtime uses the NIB to translate logical actions to a physical configuration, determine a spanning tree for default packet forwarding, and to hold switch information such as manufacturer, version, and its ports' speeds, configurations, and statistics.

For example, PANE's runtime implements a bandwidth reservation, $(M, \textbf{Reserve}(n))$, by querying the NIB for the shortest path with available queues between the corresponding hosts. Along this path, PANE creates queues which guarantee bandwidth n, and flow table rules to direct packets matching M to those queues. We chose this greedy approach to reserving bandwidth for simplicity, and leave the implementation of alternatives as future work.

PANE also uses the NIB to install **Deny** rules as close as possible to the traffic source. If the source is outside our network, this is the network's gateway switch. If the source is inside the network, packets are dropped at the closest switch(es) with available rule space.

The NIB we implement is inspired by Onix [55]. It uses a simple discovery protocol to find links between switches, and information from our forwarding algorithm, such as ARP requests, to discover the locations of hosts.

5.2 Additional Features

The PANE runtime supports several additional features beyond the requests, hints, and queries previously described. Principals are able to query PANE to determine their available capabilities, examine the schedule of bandwidth availability, create sub-shares, and grant privileges to other principals. PANE's API also provides commands to determine which existing requests and shares can affect a specified flowgroup; this is particularly useful for debugging the network, such as to determine why certain traffic is being denied.

Beyond the API, the PANE controller also includes an administrative interface which displays the current state and configuration of the network, real-time information about the controller's performance such as memory and CPU usage, and allows the dynamic adjustment of logging verbosity.

5.3 Fault Tolerance and Resilience

While the principals in PANE are authenticated, they do not need to be trusted as privileges are restricted by the system's semantics. We recognize, however, that it may be possible to exploit combinations of privileges in an untoward fashion, and leave such prevention as future work.

Our prototype implementation of PANE is currently defenseless against principals which issue excessive requests; we leave such protection against denial-of-service as future work, and expect PANE's requirement for authenticated principals to enable such protections.

The PANE controller must consider two types of failures. The first is failure of network elements, such as switches or links, and the second is failure of the controller itself.

When a switch or link fails, or when a link's configuration changes, the PANE runtime must recompile the policy tree to new individual switch flow tables, as previously used paths may no longer be available or acceptable. Because the underlying network has changed, this recompilation step is not guaranteed to succeed. If this happens, we defer to PANE's first come-first serve service model, greedily replaying requests to build a new policy tree which does compile; implementing this simply requires annotating the current policy tree's policy atoms with the order in which they were created. Developing a more sophisticated approach to re-constructing a feasible policy tree, perhaps taking advantage of priorities, or with the goal of maximizing the number of restored requests, remains as future work.

To handle failure of the controller, we can keep a database-like persistent redo log of accepted requests, periodically compacted by removing those which have expired. Upon recovery, the PANE controller could restore its state from this log. In production settings, we expect the PANE controller to be deployed on multiple servers with shared, distributed state. Switches would maintain connections to each of the controllers as newer OpenFlow specifications support. We leave the design and analysis of both options as future work. Because network principals use PANE in an opt-in fashion to receive predictable performance, a complete runtime failure would simply return the network to its current state of providing best-effort performance only.

Chapter 6

Evaluation of Participatory Networking

We evaluate our PANE prototype with the Mininet platform for emulating SDNs [57], and with real networks. Our primary testbed includes two Pronto 3290 switches running the Indigo firmware,[1] and several software OpenFlow switches (both Open vSwitch and the reference user-mode switch) running on Linux Intel-compatible hardware, and on the TP-Link WR-1043ND wireless router. Wired connections are 1 Gbps and wireless runs over 802.11n. Clients on the network include dedicated Linux servers, and fluctuating numbers of personal laptops and phones. In addition to the participatory networking API, the network also provides standard services such as DHCP, DNS, and NAT.

Members of our group have been using the testbed since February 2012 to manage our traffic, and during this time, it has been our primary source of network connectivity. The testbed is compatible with unmodified consumer electronic devices, which can easily interact with a PANE controller running at a well-known location.[2]

In the following chapters, we examine two aspects of our prototype. First, we consider four case studies of real applications that use the PANE API to improve end-user experience (Sec. 6.1). Second, we evaluate the practicality of implementing the PANE API in current OpenFlow-enabled networks, considering questions such as the latency of processing requests, and the number of rules created by networked applications (Sec. 6.2).

[1] `http://indigo.openflowhub.org/`

[2] The PANE controller could also be specified using a DHCP vendor-specific or site-specific option.

6.1 Application Usage

We ported four real applications to use the PANE API: Ekiga, SSHGuard, ZooKeeper, and Hadoop. We now describe how intentions of an application developer or user can be translated to our API, and the effects of using PANE on the network and the application. Our PANE-enabled versions of these applications are all publicly available on Github.[3]

6.1.1 Ekiga

Ekiga is an open source video conferencing application. We modified Ekiga to ask the user for the anticipated duration of video calls, and use a **Reserve** message to request guaranteed bandwidth from the network between the caller's host and either the network gateway or the recipient's host, for the appropriate time. If such a reservation is not available, Ekiga retrieves the schedule of available bandwidth from PANE and calculates the earliest time at which a video call or, alternatively, an audio call, can be made with guaranteed quality. It then presents these options to the user, along with a third option for placing a "best effort" call right away.

Realizable reservations cause the PANE controller to create guaranteed bandwidth queues along the path of the circuit, and install forwarding rules for Ekiga's traffic. Measurements of Skype use on a campus network with more than 7000 hosts show that making reservations with PANE for VoIP applications is quite feasible. Skype calls peaked at 75 per hour, with 80% of calls lasting for fewer than 30 minutes [11]. This frequency is well within current OpenFlow switches' capabilities, as we measure in Sec. 6.2.

6.1.2 SSHGuard

SSHGuard is a popular tool to detect brute-force attacks via log monitoring and install local firewall rules (*e.g.*, via `iptables`) in response. We modified SSHGuard to use PANE as a firewall backend to block nefarious traffic entering the network. In particular, this means such traffic no longer traverses the targeted host's access link.

For example, if Alice is running SSHGuard on her host and it detects a Linux syslog entry such as:

```
sshd[2197]: Invalid user Eve from 10.0.0.3
```

[3]`http://github.com/brownsys`.

SSHGuard will block Eve's traffic for the next five minutes using PANE's **Deny** request. The PANE controller then places an OpenFlow rule to drop packets to Alice's host coming from Eve's at a switch close to Eve's host.

Although this is a basic example, it illustrates PANE's ability to expose in-network functionality (namely, dropping packets) to end-user applications. Besides off-loading work from the end-host's network stack, this approach also protects any innocent traffic which might have suffered due to sharing a network link with a denial-of-service (DoS) attack.

To demonstrate this benefit, we generated a UDP-based DoS attack within our testbed network. We started an `iperf` TCP transfer between two wireless clients, measured initially at 24 Mbps. We then launched the attack from a Linux server two switch-hops away from the wireless clients. During the attack, which was directed at one of the clients, the performance of the `iperf` transfer dropped to 5 Mbps, rising to only 8 Mbps after the victim installed a local firewall rule. By using PANE to block the attack, the transfer's full bandwidth returned.

6.1.3 ZooKeeper

ZooKeeper [44] is a coordination service for distributed systems used by Twitter, Netflix, and Yahoo!, among others, and is a key component of HBase. Like other coordination services such as Paxos [56], ZooKeeper provides consistent, available, and shared state using a quorum of replicated servers (the *ensemble*). For resiliency in the face of network failures, ZooKeeper servers may be distributed throughout a datacenter, and thus quorum messages may be negatively affected by heavy traffic on shared links. Because ZooKeeper's role is to provide coordination for other services, such negative effects are undesirable.

To protect ZooKeeper's messages from heavy traffic on shared links, we modified ZooKeeper to make bandwidth reservations using PANE. Upon startup, each member of the ensemble made a reservation for 10 Mbps of guaranteed minimum bandwidth for messages with other ZooKeeper servers. Additionally, we modified our ZooKeeper client to make a similar reservation with each server it connected to.

We installed ZooKeeper on an ensemble of five servers, and developed a benchmarking client which we ran on a sixth. The client connected a thread to each server and maximized the throughput of synchronous ZooKeeper operations in our ensemble. To remove the effect of disk latency, the ZooKeeper servers used RAM disks for storage. At no time during these experiments were the CPUs of the client, switches, or servers fully loaded. Like our modified applications, this benchmarking tool is also available on Github.

Figure 6.1 shows the latency of ZooKeeper DELETE requests during the experiment. In the "Pre" line, ZooKeeper alone is running in the network and no reservations were made using PANE. In the "Post" line, we used `iperf` to generate bi-directional TCP flows over each of the six links directly connected to a host.

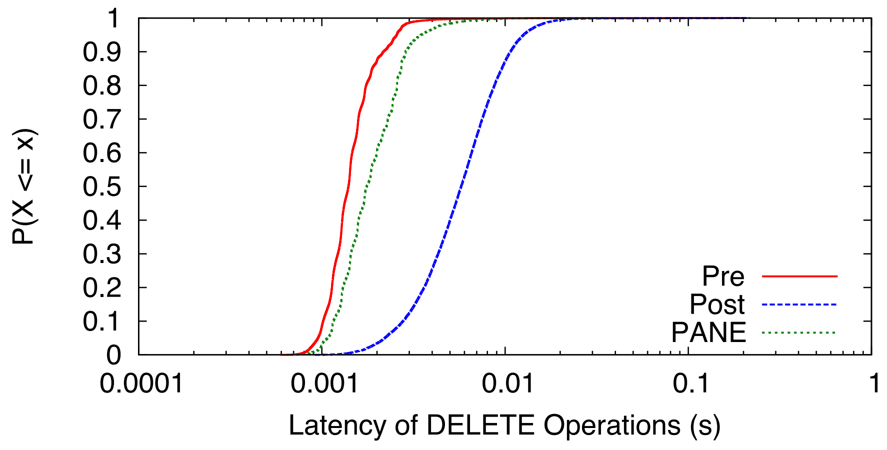
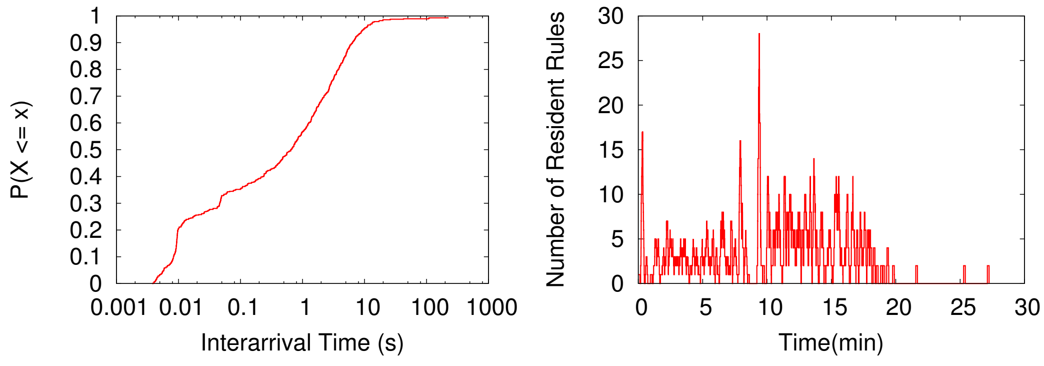


Figure 6.1: Latency of ZooKeeper DELETE requests.



(a)

(b)

Figure 6.2: Effect of Hadoop on PANE and network.

This traffic totaled 3.3 Gbps, which we found to be the maximum Open vSwitch could sustain in our setup. As shown in the figure, this competing traffic dramatically reduced ZooKeeper's performance – average latency quadrupled from 1.55ms to 6.46ms (we obtained similar results with a non-OpenFlow switch). Finally, the "PANE" line shows the return to high performance when ZooKeeper reserved bandwidth using PANE.

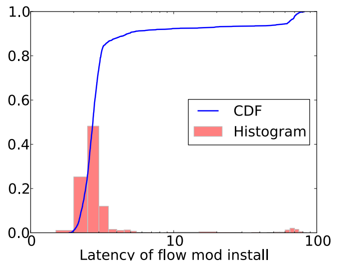
We found similar results for other ZooKeeper write operations such as creating keys, writing to unique keys, and writing to the same key. Read operations do not require a quorum's participation, and thus are less affected by competing background traffic.

6.1.4 Hadoop

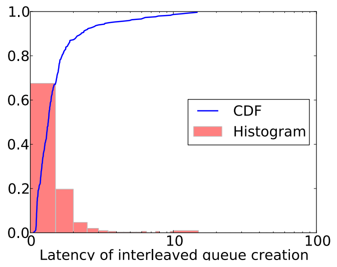
In our final case study of PANE's application performance benefits, we augmented a Hadoop 2.0.3 pre-release with support for our API. Hadoop is an open source implementation of the MapReduce [26] data-processing framework. In Hadoop, large files are divided across multiple nodes in the network, and computations consist of two phases: a map, and a reduce. During the map phase, a function is evaluated in parallel on independent file pieces. During the reduce, a second function proceeds in parallel on the collected outputs of the map phrase; the data transfer from the mappers to the reducers is known as the shuffle. During the shuffle, every reduce node initiates a transfer with every map node, making it particularly network-intensive for some jobs, such as sorts or joins. Finally, the output of the reduce phase is written back to the distributed filesystem (HDFS).

By using PANE, our version of Hadoop is able to reserve guaranteed bandwidth for its operations. The first set of reservations occurs during the shuffle – each reducer reserves bandwidth for transferring data from the mappers. The second set of reservations reserves bandwidth when writing the final output back to HDFS. These few reservations protect the majority of network transfers that occur during the lifetime of a Hadoop job. Our version of Hadoop also makes reservations when a map task needs to read its input across the network; however, such transfers are typically less common thanks to "delay scheduling" [102]. Therefore, in a typical job, the total number of reservations is on the order of $M \times R + R \times 2$ where M and R are, respectively, the number of nodes with map and reduce tasks. The number of reservations is not precisely described by this formula as we do not make reservations for node-local transfers, and reducers may contact a mapper node more than once during the shuffle phase. As reducers can either be copying output from mappers in the shuffle, or writing their output to HDFS, the maximum number of reservations per reducer at any time is set by the value of *mapreduce.reduce.shuffle.parallelcopies* in the configuration, which has a default value of five.

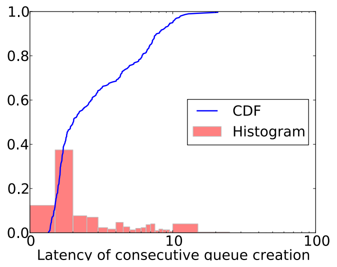
To measure the effect of using PANE to make reservations in Hadoop, we developed a benchmark which executed three 40 GB sort jobs in parallel on a network of 22 machines (20 slaves, plus two masters) connected by a Pronto 3290 switched controlled by PANE. Hadoop currently has the ability to prioritize or weight jobs using the scheduler, but this control does not extend to the network. In our benchmark, the first two jobs were provided with 25% of the cluster's memory resources, and the third, acting as the "high priority" job, was provided with 50%. The benchmark was run in two configurations: in the first, Hadoop made no requests using PANE; in the second, our modified Hadoop requested guaranteed bandwidth for each large flow. These reservations were proportional to the job's memory resources, and lasted for eight seconds, based on Hadoop's 256 MB block size. In our star topology with uniform 1 Gbps links, this translated to 500 Mbps reservations for



(a)



(b)



(c)

Figure 6.3: Latency of switch operations in milliseconds.

each link.

Averaged across three runs, the high priority job's completion time decreased by 19% when its bandwidth was guaranteed. Because it completed more quickly, the lower priority jobs' runtime also decreased, by an average of 9%, since Hadoop's work-conserving scheduler re-allocates freed memory resources to remaining jobs.

While Hadoop was running, we also measured its effect on PANE and the switch's flow table. Figure 6.2(a) is a CDF of the time between Hadoop's reservations. As currently implemented, PANE modifies the switch flow table after each request. This CDF shows that batching requests over a 10 ms window would decrease the number of flow table updates by 20%; a 100 ms window would decrease the updates by 35%. Figure 6.2(b) plots the amount of flow table space used by Hadoop during a single job. On average, Hadoop accounted for an additional 2.5 flow table entries; the maximum number of simultaneous Hadoop rules was 28.

6.2 Implementation Practicality

In addition to examining PANE's use in real applications, we also evaluated the practicality of its implementation in current OpenFlow networks. We found that our Pronto 3290 switches, running the Indigo 2012.09.07 firmware, were capable of supporting 1,919 OpenFlow rules, which took an average of 7.12 ms to install per rule. To measure this, we developed a benchmarking controller which installed wildcard match rules, issuing a barrier request after each `flow_mod` message was sent. We based this controller on Floodlight, and it is available for download from our Github page.

The latency distribution to fully install each `flow_mod` is show in Figure 6.3(a). It has two clusters – for the 92.4% of `flow_mod`'s with latency less than 10.0 ms, the average latency was 2.80 ms; the remaining 7.6% had an average latency of 59.5 ms. For PANE's principals, these much higher tail latencies imply that requests cannot always be implemented within a few milliseconds, and for truly guaranteed traffic handling, requests have to be made at least 100 milliseconds in advance.

We found that our Pronto switches could support seven hardware queues with guaranteed minimum bandwidth on each port, and each queue required an average of 1.73 ms to create or delete, as shown in Figure 6.3(b). However, this average doubles to 3.56 ms if queues are created consecutively on the same port (*i.e.*, P1Q1, P1Q2, P1Q3, ..., P2Q1, etc.), as shown in Figure 6.3(c). This shows that an optimized PANE controller must consider the order in which switch operations are made to provide the best experience for its principals.

Together, these results suggest that an individual switch can support a minimum of about 200 reservations per second. Higher throughput is possible by batching additional requests between OpenFlow barriers. While these switch features are sufficient to support the four applications above, we found that Hadoop's performance benefited from per-flow reservations only when flows transferred more than one megabyte. For smaller flows, the overhead of establishing the reservations outweighed the benefit. In an example word count job, only 24% of flows were greater than 1 MB; however this percentage rises to 73% for an example sort.

6.3 Related Work

Programming the Network PANE allows applications and users to influence network operations, a goal shared by previous research such as active networking [86]. In active networks, principals develop distributed programs that run in the network nodes. By contrast, PANE sidesteps active networks' deployment challenges via its implementation as an SDN controller, their security concerns by providing a much more restricted interface to the network, and their complexity by providing a logically centralized view of the network.

Using Application-Layer Information Many previous works describe specific cases in which information from end-users or applications benefits network configuration, flexibility, or performance; PANE can be a unifying framework for these. For example, Hedera [1] showed that dynamically identifying and placing large flows in a datacenter can improve throughput up to 113%. PANE avoids Hedera's inference of flow size by enabling applications and devices to directly inform the network about flow sizes. Wang, *et al.* [92] propose *application-aware networking*, and argue that distributed applications such as Hadoop can benefit from communicating their preferences to the network control-plane, as we show in §6.1. ident++ [65] proposes an architecture in which an OpenFlow controller reactively queries the endpoints of a new flow to determine whether it should be admitted. TVA is a network architecture in which end-hosts authorize the receipt of packet flows via capabilities in order to prevent DoS-attacks [97]. By contrast, PANE allows administrators to delegate the privilege to install restricted network-wide firewall rules, and users can do so either proactively or reactively (cf. §6.1.2).

UPnP [88] allows applications to control a network gateway, such as to add a port-forwarding entry to a NAT table, but is restricted to this setting.

Darwin [21] introduced a method for applications to request use of a network's resources, including computation and storage capabilities in network processors. Like PANE, Darwin accounts for resource use hierarchically. However, it does not support over-subscription, lacks support for access control and path management, and requires routers with support for active networks. Yap, *et al.* have also advocated for an explicit communication channel between applications and software-defined networks, in what they called *software-friendly networks* [98]. This earlier work, however, only supports requests made by a single, trusted application. By contrast, PANE's approach to delegation, accounting, and conflict-resolution allow multiple applications to safely communicate with an SDN controller.

Network QoS and Reservations Providing a predictable network experience is not a new goal, and there is a vast body of protocols and literature on this topic. PANE relies heavily on existing mechanisms, such as reservations and prioritized queue management [53, 84], while adding user-level management and resource arbitration. PANE also goes beyond QoS, integrating hints and guarantees about access control and path selection. To date, we have focused on mechanisms exposed by OpenFlow switches; we expect other mechanisms for network QoS such as those proposed for rate-limiting TCP streams in a distributed fashion [73] could be integrated as well. PANE's primary contribution is a unified framework for exposing these mechanisms while delegating authority and accounting for resource use within a single network.

Like PANE, protocols such as RSVP [14] and NSIS [60] provide applications with a way to reserve network resources on the network. PANE, however, is designed for single administrative domains, which permits centralized control for policy decisions and accounting, and sidesteps many of their deployment difficulties. PANE provides applications with control over the configuration of network paths, which RSVP and NSIS do not, and goes beyond reservations with its hints, queries, and access control requests, which can be made instantly or for a future time. Finally, RSVP limits aggregation support to multicast sessions, unlike PANE's support for flow groups.

Kim, et al. [53] describe an OpenFlow controller which automatically configures QoS along flow paths using application-described requirements and a database of network state. PANE's runtime performs a similar function for the **Reserve** action, and also supports additional actions.

Recent works in datacenter networks, such as Oktopus [5] and CloudNaaS [9], offer a predictable experience to tenants willing to fully describe their needs as a virtual network, only admitting those tenants and networks whose needs can be met through careful placement. This approach is complementary to PANE's, which allows principals to request resources from an existing network without requiring complete specification.

Software-Defined Networking PANE is part of a line of research into centralized network management including Onix [55], Tesseract [96], and CoolAid [22]. CoolAid provides high-level requests and intentions about the network's configuration to its operators; PANE extends this functionality to regular users and applications with the necessary delegation and accounting, and implements them in an SDN. PANE builds upon the control-plane abstractions proposed by Onix and Tesseract for, respectively, OpenFlow and 4D [37] networks.

Recent developments in making SDN practical (*e.g.*,[38, 62, 89]) greatly improve the deployability of PANE. Resonance [66] delegates access control to an automated monitoring system, using OpenFlow to enforce policy decisions. Resonance could be adapted to use PANE as the mechanism for taking action on the network, or could be composed with PANE using a library such as Frenetic [33].

Expressing policies in a hierarchy is a natural and common way to represent delegation of authority and support distributed authorship. Cinder [77], for example, uses a hierarchy of *taps* to provide isolation, delegation, and division of the right to consume a mobile device's energy. PANE uses HFTs [29] as a natural way to express, store, and manipulate these policies directly, and still enable an efficient, equivalent linear representation of the policy.

FlowVisor [79] divides a single network into multiple slices independently controlled by separate OpenFlow controllers. FlowVisor supports delegation – a controller can re-slice its slice of the network. Each of these controllers sends and receives primitive OpenFlow messages. In contrast, PANE allows policy authors to state high-level, declarative policies with flexible conflict resolution.

Networking and Declarative Languages PANE's design is inspired by projects such as the Margrave tool for firewall analysis [67] and the Router Configuration Checker [28], which apply declarative languages to network configuration. Both use a high-level language to detect configuration mistakes in network policies by checking against predefined constraints. PANE, however, directly integrates such logic into the network controller.

FML [43] is a Datalog-inspired language for writing policies that also supports distributed authorship. The actions in PANE are inspired by FML, which it extends by involving end-users, adding queries and hints, and introducing a time dimension to action requests. In an FML policy, conflicts are resolved by a fixed scheme – deny overrides waypoints, and waypoints override allow. By contrast, PANE offers more flexible conflict resolution operators. For example, within a single HFT policy tree, one policy node may specify "allow overrides deny," while another specifies "deny overrides allow."

FML also allows policies to be prioritized in a linear sequence (a *policy cascade*). PANE can also express a prioritized sequence of policies, in addition to more general hierarchies. For example, PANE uses an inverted "child overrides parent" conflict-resolution scheme (Sec. 4.2.3) by default, and the author of an individual policy node can adopt a more restrictive "parent overrides child" scheme. FML does not support both "child overrides parent" and "parent overrides child" schemes simultaneously.

The eXtensible Access Control Markup Language (XACML) provides four combiner functions to resolve conflicts between subpolicies [36]. These functions are designed for access control decisions and assume an ordering over the subpolicies. By contrast, HFTs support user-supplied operators designed for several actions and consider all children equal.

Chapter 7

Exodus

Toward Automatic Migration of Enterprise Network Policies to SDNs

This chapter considers the problem of migrating distributed network configurations to equivalent SDN controller programs, and introduces Exodus, a system we developed for performing this conversion. In subsequent chapters, we will see how Exodus works by following a detailed example, evaluate its capabilities, and reflect on the lessons it offers regarding SDN migration.

Exodus consumes configurations written in languages such as Cisco IOS and Linux iptables, and generates programs written in Flowlog, a research language for SDN programming [68]. Flowlog provides a compiler and run-time system for controlling OpenFlow switches; atop this, Exodus generates both the controller software and the switch configurations needed to execute these programs. Two of Flowlog's advantages are that it provides rich verification tools (which are not the focus of this dissertation), and enables Exodus to generate code we believe is easy to later evolve and maintain (Chap. 9). We will discuss the choice of Flowlog in more detail in Sec. 7.2.

By bootstrapping an SDN controller program with equivalent behavior to the original network (in full or in part), Exodus's approach lets network operators iteratively obtain the benefits of centralization. For example, Exodus can convert the configuration of just a single router, replacing one piece of the network, or it can convert the network's complete set of configurations. The resulting SDN controller can then be used to control OpenFlow-enabled switches in the production network, or it can be used to evaluate the new configuration in a laboratory or emulation environment.

This automated (and quick!, see Chap. 9) process gives the network operators time to become comfortable with their new SDN, before upgrading more critical components of the network. (For instance, Panopticon [58] shows how to achieve many benefits of SDN in a cost-aware manner, by only selecting a few routers at a time for conversion to an SDN; however, Panopticon does not explain what software to actually *run* on the controller, the gap that Exodus fills. We discuss alternate approaches to SDN migration, such as hybrid mode switches, in Sec. 10.4.) Only after the existing policy is successfully migrated may it make sense to begin reaping additional benefits which are exclusive to SDNs, such as treating the network as a single "big switch."

In keeping with the heterogeneity of modern networks, Exodus is not limited to IOS. Because it uses a rich intermediate language based on first-order logic, it can handle other configuration languages as well, limited only by parsing and translation into this logic; we have built a translator for Linux iptables as well, and support features similar to those in Juniper Network's JunOS. Because the intermediate language is similar enough to Flowlog, we present only the latter, which is also the concrete output of Exodus. Also, to make the dissertation more broadly accessible, we present the compiler through examples rather than formal rules.

The rest of this exposition proceeds as follows: we start by exploring an input language to our compiler, Cisco IOS, and introduce a running example program (Sec. 7.1). Next, we describe our chosen target language, Flowlog, and justify our choice (Sec. 7.2). This is followed by a description of our system (Chap. 8), which we evaluate on the configurations of a large campus network (Chap. 9). Conducting this research has identified several interesting weaknesses in current technology, which we discuss in Chap. 10. We then explain how we can extend our prototype to cover more features (Sec. 10.3), discuss related work (Sec. 10.4), and conclude.

7.1 Background: Cisco IOS

We begin by introducing the source language, IOS. IOS is expressive: it provides not just a "routing" language or a "firewall" language, but also a rather wide array of features that we discuss below. This set of features ensures that our compilation task is a non-trivial one. (Later in this section, we discuss configuration languages other than IOS.)

As a running example for the exposition, we present a small enterprise's configuration, consisting of a pair of networking devices connected as shown in Fig. 7.1. The figure represents two devices, `int` and `ext`, that sandwich a DMZ. The enterprise places publicly visible servers, such as the Web server, in this DMZ. External traffic is allowed to connect to specified servers and ports in the DMZ, but traffic cannot penetrate further to enter the corporate LAN. In turn, limited traffic from the corporate LAN is allowed to egress to the external

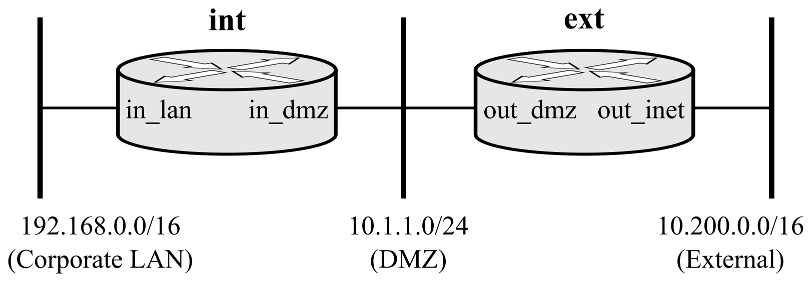


Figure 7.1: Topology for Example Network

network. Finally, the external network is allowed certain carefully delineated connections with the corporate LAN, as we discuss below.

Listing 7.1 shows a small IOS configuration for a single device above. (We will use excerpts from this listing throughout this dissertation.) Line 1 defines the name of the router to be `int` (short for "internal"). Lines 3-5 define an interface (or "port") called `in_dmz`, assign it an IP address and subnet (`10.1.1.1/24`), and indicate that the router should apply NAT functionality to traffic arriving at this interface (which is an *outside*, or public-facing, interface). Lines 7-10 define the internal interface `in_lan`, and assign an ingress filter (via **`access-group`**) as well as an IP address and subnet (`192.168.1.1/16`). NAT is enabled here as well, but as an *inside* interface.

Lines 12-15 define the access-list (or ACL) `102`, which is used to filter traffic arriving at `in_lan`. It says that: **`ip`** packets arriving from `192.168.4.1/24` should be denied access to the `10.1.1.3` host, `tcp` packets from any source destined for `10.1.1.3:25` are permitted, as are `tcp` packets to port `80` on any host. Other packets are denied. IOS resolves conflicts between these rules using the standard first-applicable semantics: A packet from the forbidden subnet will be denied, even if it is destined for `10.1.1.3:25`.

Line 17 configures NAT for the router. In this case, all outgoing packets will be rewritten to have source address `10.1.1.1` (**`interface`** `in_dmz`). The **`overload`** keyword enables multiple internal devices to map to the single external address, multiplexing via the traditional Layer-4 attributes. While IOS also allows a pool of external addresses to be used, that feature is beyond the scope of this example. Access-list `1` dictates which packets NAT should apply to. Line 18 defines access-list `1`, which matches all packets from the subnet on `in_lan`. Finally, line 20 introduces a default route. If a packet is destined for an address outside the two interfaces' subnets, `10.1.1.2` will be its next-hop router.

Listing 7.2 provides the configuration for the other device, `ext`. The IOS features it uses are mostly identical to those in Listing 7.1. This router shares the DMZ subnet (`10.1.1.0/24`) with the `int` router. Its other interface exits the example network via subnet `10.200.0.0/16`. The one major difference is that this device uses a *reflexive* access-list.

```
hostname int

interface in_dmz
ip address 10.1.1.1 255.255.255.0
ip nat outside

interface in_lan
ip access-group 102 in
ip address 192.168.1.1 255.255.0.0
ip nat inside

access-list 102 deny ip 192.168.4.1 0.0.0.255 host 10.1.1.3
access-list 102 permit tcp any host 10.1.1.3 eq 25
access-list 102 permit tcp any any eq 80
access-list 102 deny any

ip nat inside source list 1 interface in_dmz overload
access-list 1 permit 192.168.1.1 0.0.255.255

ip route 0.0.0.0 0.0.0.0 10.1.1.2
```

Listing 7.1: Example IOS Configuration (1)

```
hostname ext

interface out_dmz
ip access-group 103 in
ip address 10.1.1.2 255.255.255.0

interface out_inet
ip access-group 104 in
ip address 10.200.1.1 255.255.0.0

ip access-list extended 103
  deny ip any host 10.200.200.200
  deny tcp any any eq 23
  permit tcp host 10.1.1.1 any eq 80 reflect returnflow
```

```
15   permit tcp host 10.1.1.1 any eq 22 reflect returnflow
16   deny any
17
18 ip access-list extended 104
19   deny 10.200.200.200
20   permit tcp any host 10.1.1.3 eq 25
21   permit tcp any host 10.1.1.4 eq 80
22   evaluate returnflow
23   deny any
```

Listing 7.2: Example IOS Configuration (2)

Reflexive access lists are used to dynamically open temporary firewall holes to permit return packets for approved outgoing flows. The rules on lines 14-15 allow outgoing web and SSH traffic from the NAT address, with the additional stipulation `reflect returnflow`, which creates a table (and gives it the name `returnflow`) of ongoing flows approved by this rule. Line 22 evaluates this table, permitting the return traffic. Note the lack of a **`permit`** `tcp any host 10.1.1.1` rule to permit the NAT's return traffic; such a rule is both overly broad and unnecessary due to the reflexive ACL configuration.

Other Configuration Languages So far, we have discussed only the IOS language. Our parser also handles a significant subset of the Linux iptables language, which has similar functionality but different syntax. For instance, consider the following configuration (which mirrors the permit rule on line 13 of Listing 7.1):

```
iptables -A INPUT -i int_lan
         -d 10.1.1.3 -dport 25
         -p tcp -j ACCEPT
```

Exodus has an internal, logic-based intermediate language that serves as a target of all these compilers. Because this intermediate language is agnostic to surface syntax, we believe Exodus can also work well with other popular (and in some cases, competing) configuration languages, such as Juniper's JunOS. Nevertheless, for simplicity, the rest of this dissertation is written in terms of IOS.

Exodus Compiler Coverage: Challenges and Scope There are numerous challenges involved in converting such configurations to SDN. First, the text to be converted describes multiple small programs: packet filtering, NAT, static routing tables, and local-subnet routing. Each of these must be faithfully translated by the compiler. Second, the semantics of these programs resides in the firmware that interprets them; uttering **`ip nat`** `inside` does not describe how to implement the NAT, but merely configures it according to the available settings. Third, the configuration is tailored to a single network appliance; every such device on the network has a separate configuration that describes its local behavior. A complete translation to SDN must integrate all such devices into a single, unified system.

Finally, the IOS language is used in a variety of different network appliances, and exposes many additional features (e.g., deep packet inspection, dynamic routing, multicast groups, DoS protection, VPNs, and more) beyond those already presented. Moreover, even basic features of IOS can exhibit a sometimes baffling array of variant syntax. For instance, the `access-list`s of Listing 7.1 could have been written differently, as an `ip access-list`, using similar but not identical syntax as seen in Listing 7.2.

As an initial step, we have focused on an essential set of features rather than attempting to convert all of IOS. To date, our IOS-to-SDN compiler supports:

1. interface and subnet configurations;
2. standard and most extended IOS ACLs, including reflexive access-lists;
3. static routing tables, and policy-based static routing; and
4. ACL-based "overload" NAT.

It also accepts multiple IOS configurations to produce a single SDN program.

Overload NAT (an IOS term for port address translation, where a single public IP address is multiplexed by Layer-4 attributes) was chosen to demonstrate the viability of our compilation technique; the same techniques could be used to support other varieties, such as static network address translation, or a NAT employing a pool of IP addresses. We defer further discussion of how to extend our compiler to support additional IOS features until Sec. 10.3.

7.2 Choosing a Target Language

Since our goal is to translate a collection of IOS policies into an SDN controller program, we must select a target controller platform. There are many options: C++ code for the NOX platform [38], Python code for POX [71], or Java atop Floodlight [32]. There are also numerous research languages available [34, 48, 64, 68, 89–91]. Our final choice was motivated by the following needs:

Output in a High-Level Language After the policies have been turned into SDN programs, they will not remain static: instead, they will need to be modified as the network and its requirements evolve. Therefore, it is important to produce programs that are not so low-level that they end up being regarded as "write-only". Instead, we must target a meaningfully high-level language, and produce relatively readable output in that language. This requirement makes the highly expressive research languages [34, 48, 54, 68, 91] an attractive choice.

One advantage to producing high-level code is that it can also then be subjected to program analysis and verification activities, which will better support evolution of the controller software. There is a wide and growing range of such tools, though the most powerful, *sound* tools seem to be built for research languages [39, 68].

Support for Controller State As we have already seen, policies refer extensively to state. The resulting controller programs must be able to support NAT, reflexive access lists, and other features that dynamically affect how packets are handled. Thus, the target SDN platform must support stateful controller programs, and allow the modification of network behavior based on controller state.

While platforms like NOX obviously support these, some research languages do not [63, 64, 91]. This requirement also bars us from using the simplest of targets – OpenFlow switch rules – as they are stateless and cannot themselves implement these dynamic policies [62]. High-level languages with built-in state include Maple's algorithmic policies [91], and the rule-based languages Nlog [54], Flog [48] and Flowlog [68].

Support for Proactive Compilation Recent research has shown how to proactively compile high-level SDN programs to OpenFlow flow tables without programmer intervention [63]. Selecting such a platform removes the need for our compiled SDN programs to micro-manage flow-table updates on switches, and allows us to focus on the core goals of this project.

Availability as Open Source Finally, as developing our IOS compiler required us to extend the target language in numerous ways (Sec. 10.4), having an open source implementation was a necessity. This requirement eliminated several candidate languages (e.g., both Maple and Nlog are closed-source).

This collection of requirements induced us to choose Flowlog as our target language. By doing so, our compiler can generate high-level rule-based code, and exploit an existing proactive compiler (which in turn relies on the Frenetic project's NetCore language [63]) to OpenFlow. In addition, Flowlog has powerful verification tools to support the evolution of the generated controller program.

However, we do not claim that the choice of Flowlog is canonical. Though we have presented arguments in favor of it, the choice of target language is also a matter of taste: some might prefer Java or C++ generated for Floodlight or NOX to programs in a rule-based language. Nevertheless, Flowlog does provide a stateful, proactively-compiled language which therefore serves as an excellent target for compilation; therefore, *we can view Flowlog the language as merely an API for its proactive compiler*. The ideas of this dissertation apply just as well to other compilation targets, though the engineering decisions are likely to be rather different (e.g., if one were generating C++ code that needed to proactively manage OpenFlow rules).

7.3 Flowlog

Flowlog Example Since we will use Flowlog to show the output of compilation in the remainder of the dissertation, we begin with an illustrative example, which we explain line-by-line:

```
1 TABLE seen(ipaddr);
2 ON ip_packet(pkt):
3   DO forward(new) WHERE
4     new.locPt != pkt.locPt;
5   INSERT (pkt.nwSrc) INTO seen WHERE
6     pkt.nwDst IN 10.0.0.0/16
7     AND NOT seen(pkt.nwSrc);
```

Line 1 declares a one-column table of IP addresses. As tables are a common representation of network data, used for routing tables, ARP caches, NAT tables, and more, the Flowlog runtime maintains state in the form of a database.

The remainder of the program comprises two *rules*, both of which are triggered by any IP packet arrival (line 2) on the network. The first rule (lines 3-4) implements a basic "flood" forwarding policy; the `pkt.locPt` term represents the incoming packet's arrival port, and the `new.locPt` term represents the packet's egress port. If multiple valid egress ports exist, the packet will be sent out of all of them. The second rule (lines 5-7) inserts the packet's source IP address into the table if the packet is destined for the `10.0.0.0/16` subnet, and the address is not already stored in the table.

Flowlog Runtime The Flowlog runtime is implemented in OCaml, and is built atop the NetCore and OCaml-OpenFlow packages. Packet arrivals, switch connections, and other OpenFlow events are passed via NetCore to Flowlog, where they trigger appropriate rules. For instance, IP packets would trigger the two rules in the example above. Correspondingly, changes in Flowlog tables are propagated by its runtime back to NetCore, which in turn updates OpenFlow rules on the relevant switches.

Although the program's text makes it appear that every packet is seen by the program, Flowlog's proactive compiler is in fact far more efficient. The switch rules it produces ensure that the only packets that reach the controller are ones that will change its internal state (in this example, packets with as-yet-unseen source addresses).

When we execute the above Flowlog program, it compiles into the following initial NetCore policy, which is then distributed to the switches by the NetCore runtime:

```
1 (filter dlTyp = ip; all) +
2 (filter dlTyp = ip &&
3         dstIP = 10.0.0.0/16; fwd(OFPP_CONTROLLER))
```

Line 1 specifies that all **`ip`** packets be flooded. Line 2 selects the packets of interest and sends them to the controller. If `10.0.0.1` pings `10.0.0.2`, the controller receives both the initial echo request and reply packets. It adds both IP addresses to its state table, and issues a new policy that ensures it does not see packets from those hosts again:

```
(filter dlTyp = ip; all) +
(filter (dlTyp = ip &&
         dstIP = 10.0.0.0/16) &&
         !(srcIP = 10.0.0.1 ||
            srcIP = 10.0.0.2);
 fwd(OFPP_CONTROLLER))
```

Chapter 8

From IOS to SDN

Now we present our core technical contribution. Our goal is the workflow of Figure 8.1. For instance, given the examples from Listing 7.1 and Listing 7.2 stored as files `natfw.txt` and `outerfw.txt` respectively, the user runs:

```
exodus natfw.txt outerfw.txt
```

This produces an SDN system that mirrors the network behavior represented by these IOS configurations. Given this two-device configuration, Exodus produces an SDN system for two OpenFlow switches wired in the same way, each holding six tables. It also produces Flowlog code that uses these OpenFlow tables, together with internal state relations, to reproduce the behavior of the original devices.

We present this process in three steps. First (Sec. 8.1), we describe the flow tables, and explain how we map them to current OpenFlow hardware. Second (Sec. 8.2), we describe the compiler from IOS to Flowlog that generates the controller software. Finally (Sec. 8.3), we discuss deploying and running the resulting system using the complete Exodus tool suite, which is publicly available on Github.

8.1 Network Configuration

Exodus needs to reflect the semantics of IOS and the requirements for IP routers (RFC 1812 [4]). To do so, Exodus creates six logical OpenFlow tables per router in the original configuration: two for access-control, two for routing, and one each for Layer-2 rewriting and NAT, as shown in Fig. 8.2(a). These tables cannot actually implement these features; that requires support from the controller. Rather, they implement the corresponding stage of the processing pipeline, dropping, rewriting, or forwarding packets as dictated by the controller.

The sequential composition of tables in Fig. 8.2(a) maps to OpenFlow 1.1+'s pipeline of multiple tables, and

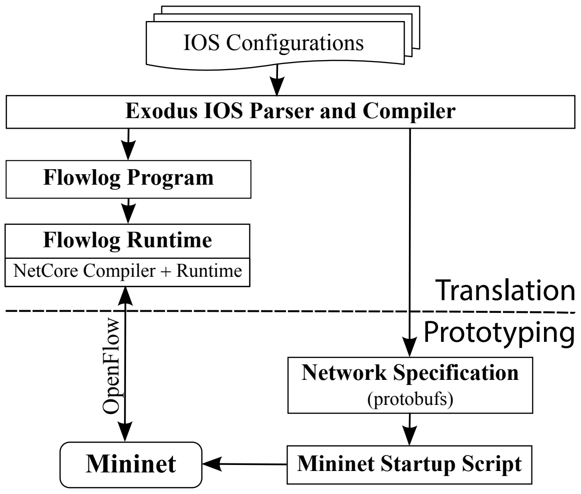


Figure 8.1: Exodus Workflow

echoes the hardware pipelines of traditional, non-OpenFlow routers. Packets first enter from the subnets on the left, where an inbound ACL is applied before forwarding them to the second stage, which implements the routing table. The routing table also determines if a packet needs to be translated. If so, if goes through the NAT table, and then through a second round of routing. The fifth stage sets the destination MAC address. A final access check is performed in the outbound ACL table, before the packet reaches the intended subnet.

In OpenFlow 1.0, which we use due to its mature support, sequential composition is well-known to create large numbers of rules due to the necessary cross-products. To keep the number of rules in check, Exodus *physically* performs the composition by wiring four single-table switches in series (see Fig. 8.2(b)). Our current pipeline is designed to minimize the number of switches; we "fold" the tables in a symmetric arrangement around the NAT, and packets flow in both directions. In the inbound direction, the second table, L2 Rewrite, is just a pass-through.

Our design prepares Exodus for transition to newer versions of OpenFlow with support for multiple tables. It also makes clear the features needed by each flow table, allowing one to program a single switch with the protocol-independent packet processors proposed in [12]: the ACL tables may match any field, but only need to drop or forward packets; the routing table forwards packets based on masked IP addresses; the NAT table performs Layer-3 rewriting and exact matching; and the Layer-2 rewriting has many "narrow" entries, matching on IP addresses and setting the corresponding MAC addresses of all connected hosts.

Of the four types of flow tables used by Exodus, three – ACL, routing, and NAT – are managed by code generated from the IOS configuration (as described in Sec. 8.2). The fourth, Layer-2 rewriting, is independent

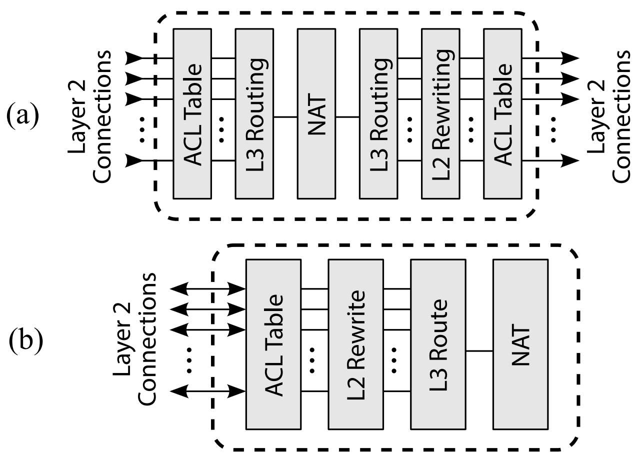


Figure 8.2: Logical flow tables in an Exodus router implementation (a), and the implementation with physical switches in OpenFlow 1.0 (b).

of IOS, and simply sets the destination MAC address on outgoing, routed traffic. To populate this, we developed an ARP Proxy, which allows the router to respond to ARP requests for its interfaces and known hosts, and to issue queries on the attached subnets.

8.2 Code Generation

The heart of Exodus is a compiler that converts IOS into Flowlog that runs on the controller. We present the compiler by describing its treatment of each supported feature. Separately, we have modified Flowlog's proactive compiler so actions generated by each of these features are mapped to the OpenFlow switch tables corresponding to that feature.

Static and Policy-Based Layer-3 Routing

The core purpose of a router is Layer-3 routing. The Exodus compiler translates IOS routing tables into Flowlog code that manages the OpenFlow tables responsible for routing. For instance, consider the default route given on line 20 of Listing 7.1. Each such rule is compiled into a Flowlog fragment that defines a next-hop IP address for incoming packets. In this case, only a "default" route needs to be specified for all packets arriving at the `int` router:

```
routerAlias("int", pkt.locSw)
  AND (10.1.1.2 = nexthop)
```

`routerAlias` is an empty Flowlog relation defined by Exodus and populated by code generated by the compiler. It maps between the string-based names in IOS and the numeric identifiers in OpenFlow. The fragment assigns a value (`10.1.1.2`) to the next-hop for all packets arriving at the `int` router; the value is then used by the modified Flowlog compiler to populate the appropriate OpenFlow table.

Exodus's output also assigns next-hops based on IOS's "policy routing" feature, which assigns next-hops to packets based on access-lists. While our main example does not use this feature, an interface might be configured as:

```
1 hostname example
2 interface eth0
3 ip policy route-map internet
```

Line 3 specifies that the policy named `internet` should be applied to traffic on this interface. The `internet` policy must be defined in the configuration, as in the following example (from [67]):

```
1 access-list 10 permit 10.232.0.0 0.0.3.255
2
3 route-map internet permit 10
4 match ip address 10
5 set ip next-hop 10.232.0.15
```

Line 1 of the policy defines a set of packets to apply a next-hop address to. Lines 3-5 specify that address – in this case, `10.232.0.15`. As in the static-routing case, Exodus creates a corresponding Flowlog fragment that assigns the appropriate next-hop for matching packets:

```
1 routerAlias("example", pkt.locSw)
2   AND portAlias("example", "eth0",
3     pkt.locPt)
4   AND pkt.nwSrc IN 10.232.0.0/22
5   AND 10.232.0.15 = nexthop
```

Line 1 filters on the packet's switch location. Lines 2-3 match the interface the packet arrived from; `portAlias` is another Flowlog relation that maps between the string-based interface names in IOS and OpenFlow's numeric port identifiers. Line 4 applies the access-list, making certain that only packets from `10.232.0.0/22` receive the next-hop address given by line 5. As in the static-routing case, the next-hop value is then used by the modified Flowlog compiler to produce OpenFlow switch tables.

Packet Filtering

We now turn to the OpenFlow tables for ingress and egress ACLs. Consider the rules on lines 12-13 of Listing 7.1. This ACL can be trivially represented as OpenFlow rules of essentially the same form, using the "drop" action for the **`deny`** rule, and forwarding otherwise. As Flowlog currently uses negation in place of an explicit "drop", the compiler embeds **`deny`** rules as negative conditions; e.g. lines 3-4 of:

```
1 ON tcp_packet(pkt):
2   DO forward(new) WHERE
3     NOT (pkt.nwSrc IN 192.168.4.1/24
4       AND pkt.nwDst = 10.1.1.3)
5     AND pkt.nwDst = 10.1.1.3
6     AND pkt.tpDst = 25
```

As IOS allows an interface to have separate ingress and egress filters, Exodus produces separate Flowlog fragments for each. This does not add any notable complexity to the output, and is in keeping with the router pipeline described in Sec. 8.1.

Reflexive ACLs

Whereas ordinary ACLs required no controller state, reflexive access-lists do. A rule tagged with `reflect`, like those seen on lines 14-15 of Listing 7.2, matches traffic like a normal ACL rule, but also causes the device to remember the traffic's source and destination for later use. A corresponding **`evaluate`** rule, like the one on line 22 of the same configuration, uses that memory to permit return traffic.

While static ACLs can explicitly permit return traffic, dynamically adding new such rules requires the SDN controller. Thus, in an SDN context, reflexive access-lists must be managed by the controller, as we describe next, rather than performed entirely by the switch.

Stage 1: "Reflect": Remembering Outgoing Traffic To hold the state on the controller, Exodus uses a relation called `reflexiveACL`, which contains a row for each hole to be opened in the firewall. This state relation resides on the *controller*, and Flowlog's runtime will use it to automatically keep the ACL tables on the switches up to date. To declare the new state relation, the compiler generates:

```
TABLE reflexiveACL(string, ipaddr,
  tpport, nwproto, ipaddr, tpport);
```

Column 1 of this relation stores the identifier (in this case, `returnflow`) used for the traffic of interest. Columns 2-6 store the standard flow-identification information. For every matching `reflect` in the IOS ACL, the compiler creates a corresponding Flowlog **`INSERT`** rule. For instance, the rule on line 14 compiles into:

```
1  ON tcp_packet(pkt):
2    INSERT ("returnflow", pkt.nwSrc,
3      pkt.tpSrc, pkt.nwProto, pkt.nwDst,
4      pkt.tpDst) INTO reflexiveACL
5    WHERE pkt.nwSrc=10.1.1.1
6    AND pkt.tpDst=80
7    AND aclAlias("ext-out_dmz-acl",
8      pkt.locSw, pkt.locPt, ANY)
9    AND NOT reflexiveACL("returnflow",
10     pkt.nwSrc, pkt.tpSrc, pkt.nwProto,
11     pkt.nwDst, pkt.tpDst);
```

Lines 5-6 ensure that the ACL rule actually applies to the packet in question, and lines 7-8 enforce that the check applies only to the proper router and interface names.

Stage 2: "Evaluate": Permitting Return Traffic For every **`evaluate`** clause in the original configuration, the compiler asserts a corresponding ACL rule in Flowlog. These rules resemble those of Line 5, and also refer to the `reflexiveACL` state table:

```
1    aclAlias("ext-out_inet-acl", pkt.locSw,
2      pkt.locPt, new.locPt)
3    AND reflexiveACL("returnflow",
4      pkt.nwDst, pkt.tpDst, pkt.nwProto,
5      pkt.nwSrc, pkt.tpSrc)
6    AND (22 = pkt.tpSrc)
7    AND (10.1.1.1 = pkt.nwDst) AND
8    NOT (10.200.200.200 = pkt.nwsrc)
```

This example encodes the **`evaluate`** rule on line 22 of Listing 7.2. The `aclAlias` reference limits the rule to the proper router and interface. The reference to the `reflexiveACL` table on lines 3-5 has a reversed ordering from the above **`INSERT`** rule; this is because the insertion rule triggered on outgoing traffic, while this rule filters incoming traffic for the same flow. The final line prevents the rule from applying if the higher priority deny rule (line 19 of Listing 7.2) would.

Network Address Translation

NAT presents two challenges. First, like reflexive access-lists, NAT in SDNs requires some amount of controller management; Exodus must produce Flowlog code that governs the dynamic nature of NAT. Second, NAT *changes* the headers of packets as they are forwarded. While header-mutation is a primitive action in OpenFlow, the exact values used depend on the NAT's overall state.

As mentioned in Sec. 7.1, our compiler currently supports only one of the many types of NAT available in IOS: dynamic port-address translation using a single public IP. This is sufficient to show the feasibility of our approach; static NAT is simpler as it requires no controller state, and pool NAT only requires an additional table of available public IPs.

The compiler produces Flowlog code that, on controller start, populates state relations describing how NAT should be performed. For example, these values for Listing 7.1:

```
1 ON startup(e):
2   INSERT (0x100000000001, 192.168.0.0, 16)
3     INTO needs_nat;
4   INSERT (0x400000000001,1,1,10.1.1.1)
5     INTO natconfig;
6   INSERT (10.1.1.1, 0x6, 10000)
7     INTO seqpt;
8   INSERT (10.1.1.1, 0x11, 10000)
9     INTO seqpt;
```

Line 1 states that the insertions happen at controller startup. Lines 2-3 record that new flows from the private IP `192.168.0.0/16` must be subject to NAT on a given switch. Lines 4-5 configure a related switch to use the given external IP address for this translation. Lines 6-9 set up initial port values for NAT (port `10000` for both TCP and UDP).

Exodus also includes a library, a fragment of which is shown below, that contains rules that utilize these tables. These describe, for instance, how to handle new TCP traffic from the internal subnet:

```
1 ON tcp_packet(pkt) WHERE
2   NOT ptassign(0x6, pkt.nwSrc, pkt.tpSrc,
3     ANY, ANY)
4   AND natconfig(pkt.locSw, pkt.locPt,
5     publicLocPt, publicIP)
6   AND NOT natconfig(pkt.locSw, ANY,
7     ANY, pkt.nwDst)
```

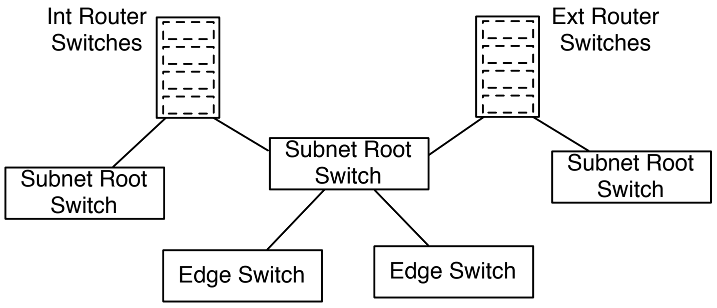


Figure 8.3: Two Exodus routers attached to a shared subnet

```
8  AND seqpt(publicIP, 0x6, x)
9  AND add(x, 1, publicPt):
10 INSERT (0x6, pkt.nwSrc, pkt.tpSrc,
11   publicIP, publicPt) INTO ptassign;
12 DO forward(new) WHERE
13   new.locPt = publicLocPt
14   AND new.nwSrc = publicIP
15   AND new.tpSrc = publicPt
16   TIMEOUT 600;
```

Lines 2-3 dictates that no NAT port has yet been assigned to this flow. Lines 4-5 extracts the public IP address to be used. Lines 6-7 prevents the rule from applying if the packet is being sent to a public IP that is reserved for NAT (i.e., if it is return traffic). Lines 8-9 produce a fresh TCP port by incrementing the one used most recently. Lines 10-11 record the port used and the source of the NAT flow (for translating future packets in the flow). Lines 12-16 send this initial packet on its way; the final `TIMEOUT` line applies a 600-second idle timeout to the forwarding rules, which allows the program to reclaim expired NAT ports for later use.

Although the tables and the data here are different, this process is similar to how the compiler handles reflexive access-lists. While space restrictions prevent us from giving the entire NAT program, the rules produced for return traffic, UDP traffic, etc. are similar.

8.3 Prototyping the Network

Exodus produces more than just a Flowlog program: it also produces a description of the network on which it runs, including the switches, their wiring, and configuration, which allows us to have a running prototype of the network in an emulation environment such as Mininet [57]. The specification can also serve as a blueprint for a physical network that implements the same policies as the original network.

These configuration values are loaded into the Flowlog program on startup, which then interacts with the corresponding tables in the physical or emulated network. The compiler outputs the complete description in a custom Google Protocol Buffers format [72]. We have written a Python script that loads this described network into Mininet for experiments.

In addition to the switches for each router, this script also creates sample Layer-2 networks for each unique subnet, as shown in Fig. 8.3. These sample subnets include a per-subnet "root switch" to which routers on that subnet are attached, as well as generic "edge switches" and end-hosts (not shown). Forwarding within the subnets is provided by a standard MAC Learning application (also in Flowlog), although any form of Layer-2 connectivity will suffice. Finally, the script also launches SSH and web servers on each host to support interactive testing of the network's complete ACL, NAT, and routing configuration.

Chapter 9

Evaluation of Exodus

Exodus is an experimental tool for converting traditional network configurations into an equivalent SDN controller program. To evaluate it, we must consider three important questions: feasibility, utility, and correctness.

9.1 Feasibility

The feasibility of Exodus is a function of both its own methods, and the OpenFlow technology on which it runs. Beyond examples such as those in Sec. 7.1, we have tested Exodus's features and scalability with the router configurations of the publicly available Stanford network configuration [103], a large campus network supporting more than 30,000 users. We find that Exodus is capable of quickly producing equivalent policies that fit within the bounds of existing OpenFlow hardware. However, the limits of existing hardware restrict the number of simultaneous end-hosts that can be supported.

To test features, we ran Exodus over the Sec. 7.1 examples, launched the resulting network in Mininet, and manually exercised the policies in the configuration to verify their compliance. For example, we were able to successfully connect via SSH to hosts in the 10.200.0.0/16 subnet from hosts attached to edge switches in the corporate LAN. This showed the success of our routing, NAT, and reflexive ACL implementations. Standard ACL was similarly confirmed with additional tests.

To test scalability, we ran Exodus on the 16 router configurations of the Stanford network. These routers had between 15 and 84 interfaces each, a total of 1500 ACLs, and did not use IOS's NAT functionality. The conversion required 2.19 seconds (averaged over 10 runs) on a 1.7 GHz Core i7 laptop, and the runtime populated each switch's flow tables within 2-5 seconds (64 flow tables in total).

After startup, we found that the switches implementing IP routing each required only two or three more OpenFlow rules than the number of attached subnets, with the maximum being 86. The sizes of the Layer-2 rewrite tables ranged from 55 to 325 rules, depending on the number of attached subnets. The ACL tables ranged in sizes from 31 to 581 OpenFlow rules (2840 in total). Subsequent evaluation revealed that while the size of each ACL table depended on the number of attached subnets and complexity of individual ACL configurations, this was an area where Flowlog's efficiency could improve on this computationally difficult problem [3]. Nonetheless, all of these tables fit within the limits of existing OpenFlow hardware, which typically support just 1500-3000 rules [76].

As expected, however, the number of OpenFlow rules in the Layer-2 rewrite table (and the NAT table for the small example) scale linearly with the number of end-hosts (number of connections for the NAT). In Exodus's current design, the Layer-2 rewrite table contains one rule per end-host in attached subnets. Similarly, to the NAT table, Exodus adds three rules per new connection; this is one more than optimal, for reasons we discuss below (Chap. 10). Therefore, until OpenFlow hardware matures – either with specifically-designed hardware [12], or by making non-TCAM tables available in existing switches – rate-limiting end-host activities will be extremely important in enterprise networks.

9.2 Utility

We now consider utility: When compiling IOS into an SDN controller, does the compiler leave us with readable, editable code? Can the code be easily augmented with new SDN features, not present in the original network? And, can Exodus help us better understand the network?

The maintainability question is important, since a true migration to SDN must support not only the configuration at the time of migration, but also future edits. Kim *et al.* [51] report more than 2,000 router configuration changes per month at Georgia Tech, and over 8,000 switch configuration changes per month at the University of Wisconsin, Madison. Flowlog has a rule-based syntax and trigger-response abstraction that resembles IOS access-lists; adding a new permission or opening a hole in a firewall only requires new rules be added, and existing rules can be left intact.

Removing existing permissions does require editing the current ACL rules: either by removing a rule entirely, or adding an additional predicate to adjust the permissions. Adjusting policy routing would require a similar change. As Sec. 8.2 shows, however, the code generated by Flowlog has two properties: (1) There is a relatively clear mapping from the original IOS configuration (which is further enhanced by comments inserted by the compiler), making it easy for operators to map their knowledge of the IOS configurations to the Flowlog program. (2) The code generated is fairly high-level and direct, easing subsequent maintenance. Furthermore, operators can be guided in these changes by Flowlog's existing analysis tools. Finally, even if the SDN is eventually reimplemented, the Flowlog version has value as an oracle for systematic testing of the new system against the old.

The remainder of the Exodus-generated configuration is governed by static table entries loaded at program startup. For example, assigning a given interface the IP and subnet `10.1.1.1/16` is performed by a single row in a table. These can be edited even more easily than the program rules.

To further evaluate Exodus's code quality, we sought to augment its output with a novel SDN application, providing features not present in IOS. This application first blocks multicast DNS traffic, a significant consumer of bandwidth in enterprise networks [17]. The application then implements tunnels, on-demand, across the network, for end-users who wish to stream content to registered Apple devices. We were pleased to find this addition easy to accomplish: Exodus, which generates NetCore, can be composed with other NetCore programs either sequentially or in parallel, and can also be composed with other Flowlog programs. This mDNS application required only seven new Flowlog rules and an additional table.

Finally, while running Exodus on the Stanford network configuration, it detected missing references, contradictory statements, and unused IOS fragments present in the original configuration. Its ability to detect such fragments using the first-order logic in its intermediate language makes Exodus's utility during migration apparent, even before the first OpenFlow rules are generated. We will discuss several approaches to using Exodus as part of an SDN migration in Sec. 10.2.

9.3 Compiler Validation

While we did the manual checks of correctness in our small configurations, as described in Sec. 9.1 above, we have left open the question of how to formally validate or prove the correctness of Exodus. Broadly speaking, there are two approaches we might use.

The first is to statically prove the correctness of the compiler (and accompanying run-time system). The logical underpinnings of Flowlog make it easy to put the work on formal foundations, including proofs of compiler correctness. However, such a proof would require a comprehensive semantics for the source languages. Unfortunately, for languages like IOS, there are only partial solutions to this problem [16, 67, 104]. (We are not aware of any formalizations of iptables.)

The other, complementary, approach is to compare the behavior of the resulting systems. That is, we can dynamically check for equivalence between the Exodus-produced OpenFlow tables, and the forwarding behavior of the source input. This approach could be built upon the recently developed Header Space Analysis [50] and related ATPG tool [103]. To do so, the existing HSA implementation would need to be extended to support the range of language features supported by Exodus, and to accept OpenFlow tables as input. We discuss this further in Sec. 10.4.

Chapter 10

Discussion of SDN Migration

Automatically translating one language to another starkly highlights differences between the source and target languages. While developing Exodus, we have encountered concrete deficiencies in OpenFlow, NetCore, and Flowlog, relative to the functionality of Cisco's IOS. In addition, our work with Exodus has provided clarifying examples for some of the architectural and physical tradeoffs between traditional and software-defined networks. We now discuss these two areas.

10.1 Language Limitations

Exodus uses a stack of languages (cf. Chap. 8) to transform Cisco's IOS into an SDN controller, ranging from a specific, year-old research artifact (Flowlog) to a general-purpose, in-production specification (OpenFlow). As such, we will chiefly focus on the issues uncovered in OpenFlow, offer a few lessons for high-level SDN language designers, and only briefly touch upon deficiencies exposed in Flowlog.

10.1.1 OpenFlow Shortcomings

Exodus exposed three shortcomings in OpenFlow which we discuss in detail. Although Exodus was developed against the OpenFlow 1.0 specification, as it offered the most mature implementations, these shortcomings all remain in the proposed OpenFlow 1.4 standard.

Idle Timeout for NAT The first shortcoming relates to "Idle Timeouts" for flow table rules. OpenFlow includes the ability to set a per-rule timeout which is reset whenever the rule matches a packet – for example, a rule with such a timeout may expire if it has not matched any traffic during the previous 60 seconds. The switch may also be instructed to notify the controller when such a rule expires.

Together, these timeouts and notifications allow OpenFlow controllers to implement soft state (for host mobility, caching, or reusing finite resources, such as the ports in a NAT scheme), with the support of the switch hardware. This support from the switch is important, as without it the controller would be forced to poll each switch's counters periodically to determine if any rules should have expired during the previous period. While polling each connected switch to implement Idle Timeouts may waste controller resources, it would be even worse on the switches themselves.

Recent measurements have shown hardware switches are incapable of answering more than four controller queries per second under the best conditions, and that performance decreases as the number of flow table entries increase [25, 76]. Furthermore, because switch CPU bandwidth can be the bottleneck resource, controller queries issued more frequently than once per second can increase the latency of flow table updates by an order of magnitude on some hardware [76].

Unfortunately, even simple policies cannot always be expressed with a single flow table rule. A NAT, for example, must commonly generate two rules, one in each direction, to support a single translated flow. This translated flow is also assigned a finite resource (one of the Layer-4 ports on a public-facing IP address), which should be released when the flow is no longer in use: a perfect use-case for Idle Timeouts, if not for the need for multiple flow table rules.

Hence, an OpenFlow-based NAT controller would prefer the Idle Timeout be triggered only when *both* rules have been idle for the specified period; any other design is simply incorrect. Extending OpenFlow 1.4's FlowMod "bundles," introduced recently to support atomic transactions, to also be a unit over which an Idle Timeout could be set, would solve this problem. Finally, OpenFlow switches should ideally also support a second style of Idle Timeouts which are triggered by TCP packets with the FIN or RST flags. Such timeouts have been supported by an Open vSwitch extension since version 1.5.90, and are configurable in Cisco IOS.

Matching with Ranges The second OpenFlow limitation we encountered with Exodus related to translating IOS rules of the following form, which match Layer-4 port numbers using ranges or inequalities:

```
access-list 101 permit tcp any any range 8080-8180
access-list 101 deny tcp any any gt 134
```

These one-line statements in IOS become many more in OpenFlow due to its more restricted syntax for matching. While OpenFlow 1.0 could only match on Layer-4 port numbers exactly, the situation at least improved with OpenFlow 1.2's introduction of the OpenFlow Extensible Match (OXM).

The OXM format provides a bit mask for each field, although some fields are additionally restricted to only certain bit mask patterns, such as CIDR masks. Even without additional restrictions, OXM's binary design still forces the rule on Line 1 to be expanded into six OXM matches: 8080/12, 8096/11, 8128/11, 8160/12, 8176/14, and 8180/16 (following the CIDR convention). Regardless of underlying implementation (hardware or software), this design creates more rules. These will be seen when displaying the flow table, when debugging control traffic, when calculating flow statistics, or when synchronizing controller state.

The syntax translation from IOS ranges and inequalities to OXM bit masks is mechanical, and an obvious benefit that can be provided by high-level SDN languages. However, their commonality suggests they should be an abstraction provided by a future OpenFlow specification, rather than a feature all high-level languages must reimplement.

Additional ICMP Fields A final OpenFlow restriction encountered by Exodus was a lack of support for matching on the identifier field of ICMP queries, which include basic ICMP Echo Requests. RFC 3022 instructs NATs to remap this field, otherwise ICMP queries cannot be multiplexed between multiple private, source IP addresses, and the same public, destination IP [83].

We encourage the Open Networking Foundation to add support for matching on and rewriting the ICMP Query Identifier field (and updating the preceding checksum) to OpenFlow. Without this support, OpenFlow-based NATs must send all ICMP traffic to the controller, using a match on the existing ICMP type and code fields, for the necessary modifications.

10.1.2 Lessons for SDN Language Designers

High-level languages for SDN programming are an important and active area of research [34, 48, 64, 68, 89–91]. Their goal is to raise the level of abstraction, freeing programmers from mundane details of flow table programming and switch implementations, and to make SDN programs more reusable and analyzable. Abstraction design is an iterative process, and our work with Exodus has led us to identify a few rough spots which we now discuss.

Composing Actions without Matches A common reason for using high-level SDN languages is to simplify the composition of multiple policies. Existing semantics for composition include parallel [34], sequential [64], and hierarchical merge [30]. Prior to this work, none of these approaches could compose an arbitrary header modification without first exactly matching on the field being modified. In other words, a (match, action) pair would be required for *every* observed source MAC address, for example, to update a packet's Layer-2 source during hop-by-hop IP routing.

The reason for this restriction is understandable: setting a header field without first exactly matching violates parallel composition. For example, consider a simple policy which sets the source MAC address on packets and emits them from port 3, which is composed in parallel with a monitoring policy to emit all packets from port 4:

```
(srcMac -> 00:00:00:00:00:01; emit: 3) || emit: 4
```

Because OpenFlow does not have an action to copy packets, this might naively become the following action sequence:

```
mod_dl_src=00:00:00:00:00:01; output:3; output:4
```

which will incorrectly send the modified packet also out port 4, instead of the original, unmodified packet. In this case, correct compilation is still possible with reordering:

```
output:4; mod_dl_src=00:00:00:00:00:01; output:3
```

However, correct compilation is only possible if the policy composes *at most one* such update without match in parallel. Previously, when the original header field was available from the exact match, a compiler could simply "undo" the header rewrite with a second modification before proceeding with the parallel composition.

Since routers must rewrite the source MAC address when forwarding packets from one subnet to another, composing at most one rewrite action without match is necessary for scaling the flow tables effectively in Exodus. Therefore, we have added support for this feature to NetCore, and believe high-level SDN languages which offer parallel composition should include this variant, as we do.

Packet Processing Continuations Existing high-level language controllers such as NetCore, Nettle, and Maple present an abstraction in which a policy function is conceptually evaluated for every packet, a model introduced by Ethane [17]. The semantics of this evaluation, however, are to run to completion – that is, a packet arrives at the controller, the function is evaluated, and new packets are emitted, before the next packet is processed.

While developing Exodus, we have found the need to occasionally *suspend* the execution of this packet-processing function, perform processing on other packets, and later return to the suspended execution. This suspended execution is known as a *continuation*.

As a concrete example, consider the process of rewriting the destination MAC address after a packet has been routed. If a packet arrives, and the router does not know the corresponding MAC address, it must suspend processing, emit an appropriate ARP request, and wait for an asynchronous reply. Only after processing the ARP reply, if any, can it finish processing the previous packet. For this reason, we encourage designers of high-level languages for SDNs to support continuations in their packet-processing model.

Stable Flow Table Output Finally, we urge the developers of high-level languages to strive to compile logically-equivalent policies to syntactically-equivalent flow tables. For the foreseeable future, rule space in hardware OpenFlow tables will remain at a premium, and application developers will find themselves regularly examining the tables to optimize their resources.

In all the languages discussed above, the syntax of generated flow tables can change dramatically when packets are reordered or logically equivalent policies are swapped. While harmless from the packets' perspective, such changes make contemporary SDN programming more difficult. Ideally, automated optimization will improve this situation, and we are encouraged by recent efforts [46, 47]; where optimizations are unavailable, we suggest a canonical ordering be used.

10.1.3 Flowlog Deficiencies

A few deficiencies in Flowlog's design were also revealed by Exodus. Taking a "default drop" position, without exposing it as an explicit action, created additional OpenFlow rules in the ACL tables (Line 5), and a third rule for each connection in the NAT table (Sec. 9.1). In addition, we found that Flowlog is unable to "dequeue" just a single element at a time from a relation; thus, our NAT could not reuse the Layer-4 ports assigned to connections it learned had closed. Finally, providing mathematical operations at compile-time, rather than only via built-in relations such as `add`, would have made our task more pleasant.

10.2 Architectural and Physical Tradeoffs

The output from Exodus is very clearly a hybrid: a centralized SDN controller with explicit mappings to a set of distributed switches. Although the collection of input policies may now be *joined*, they are not truly *unified*; as an initial prototype, Exodus does not output a policy expressed over a single "big switch" abstraction [19, 64, 78]. However, armed with the combined policies translated to a high-level SDN language, we can now consider a *range* of SDN designs, which we discuss below, both for the policy abstractions, and their physical implementations.

Furthermore, the initial step taken by Exodus, generating a set of OpenFlow rules equivalent to an organization's IOS polices, gives an organization insight into the resources required to replace an existing, traditional network with one controlled by OpenFlow. For example, how many flow tables will be needed? How many entries should they support? And, what hardware actions will be required? We previously saw example answers to these questions for the Stanford network in Chap. 9.

As described in Chap. 7, enterprise networks can be quite large. If policies were dispersed across the entire network, it could be very difficult to unify them onto a single abstraction. Fortunately, this is generally not the case, as enforcing rules only at the core of an enterprise network is generally considered a best practice for the following reasons:

1. Lower administrative burden: Making changes across the network is time-consuming and error-prone.
2. Edge-switch limitations: Cheaper edge-switches may simply be incapable of enforcing the desired policy.
3. Immunity to end-host mobility: Placing a rule at the core means it will affect all traffic on the network.

While these three reasons are a benefit during the migration, since there are fewer distributed policies to unify, it may be sensible to reconsider these reasons after migration. For the first, the simplified management experience provided by a centralized control-plane is a primary tenet of SDNs. For the second, a programmatic controller with a global view can make use of any resources which are at the edge; the question of how many resources to place at the edge is a topic of debate [20], and we will return to it below.

As for the third, end-host mobility raises a design point when implementing SDN controllers, one closely aligned with "reactive" versus "proactive" compilation. In the reactive design, proposed in the original Ethane work [17], each new flow (or perhaps each new host) would require the controller to establish flow table rules on the switch. A proactive design, by contrast, eliminates the controller overhead by pre-establishing all necessary rules on the switch, and only reacting to less frequent events such as link failures, load re-balancing, or operator updates.

These two designs represent a continuum, however, as a proactive design prevents packets from reaching the controller by placing restrictions on supported policies to enable compilation, and making assumptions about the traffic, which may create unnecessary flow table rules. In an enterprise network, end-hosts frequently disconnect, migrate, connect, and re-authenticate, which commonly involves the exchange of several messages with local network infrastructure such as DHCP and EAP servers. (Georgia Tech reports an average of 2,360 authentication events per minute on its wireless network [52].) Given this reality, a decision to compile policies in a more reactive fashion may make more sense here than in the datacenter, where unplanned mobility is not an issue, and latency is more important.

We return now to the question of edge switch functionality. An organization using Exodus already has a network, and may consider at least two paths for migrating to an SDN. Along one path, they might leave their existing, policy-less edge switches in place, and upgrade the network core to support OpenFlow. The recent Panopticon work suggests that even a single, upgraded core switch can be very beneficial [58], and the Exodus prototype applies to this scenario. Along the second path, an organization could focus on upgrading edge switches, making them capable of implementing the policy, and following an architecture similar to that proposed by Casado, *et al.* [20]. Under this approach, rules would be pushed from the core to the edge, and Exodus's current design can only offer general guidance about the total size of the network's flow tables, and not a working controller. Although evaluating the risks, costs, and benefits of these approaches is up to each organization, tools like Exodus can help illuminate the decisions.

Finally, although SDNs seek to centralize network configuration, networks will continue to have multiple regions, each under the direction of a distinct set of logically centralized controllers (for example, the sets of Onix controllers at the edge of each datacenter in Google's B4 network [45]). This will happen for many reasons including fault isolation, upgrade testing, scalability, and administrative independence. Compiling distributed router configurations into a centralized control program lets us later refactor the boundaries of these regions. It would appear to be very difficult to do such a refactoring safely without first joining the configurations using a tool like Exodus.

10.3 The Route Ahead

Our goal with this work is to present a working prototype of a system for converting existing network configurations to SDNs. We hope the possibilities raised by Exodus will motivate further development of migration tools. Exodus currently supports the Cisco IOS and Linux iptables features described above, and can be extended to support the following additional features:

VLANs VLANs introduce an abstraction layer to support multiple broadcast domains inside a single Layer-2 fabric. Adding support for VLANs in Exodus requires the straight-forward addition of an additional switch table before the current ACL table (Fig. 8.2). The VLAN table would then demultiplex the attached Layer-2 fabrics into the subnet inputs used by the current design.

Routing Protocols Exodus currently supports static and policy-based routing, storing the routes in a standard (subnet $\mapsto$ next hop) routing table. Distributed routing protocols such as BGP and OSPF can be supported by exposing this table to updates from RouteFlow [75] using Flowlog's existing support for Thrift-serialized external events.

Pooled and Static NAT Cisco IOS supports three forms of NAT: overload, static, and pooled, which correspond with N-1, N-N, and N-M translation of private to public IP addresses. Exodus implements overload NAT with a relation mapping private IP addresses to public Layer-4 ports. Its database semantics make the addition of static and pooled NAT trivial.

VPNs and Tunnels OpenFlow 1.3 introduced support for PBB (MAC-in-MAC) encapsulation, which can be used for tunneling. As with VLANs, additional data-plane layers may be (de-)multiplexed by inserting an additional table. Support for other tunneling and VPN approaches, such as IP-in-IP and GRE, currently require additions to OpenFlow or external configuration.

MPLS Supporting MPLS will likely require the greatest changes to Exodus, as it replaces the current subnet-based forwarding with a label-based approach. Although use of MPLS has been made easier since its introduction in OpenFlow 1.1, support is still missing in high-level SDN languages.

10.4 Related Work

Migrating enterprise networks to networks with centralized control is an important topic in the SDN literature. While early proposals, such as 4D [37] or SANE [18], were understandably "clean-slate" designs, with no upgrade path other than starting from scratch, a subsequent strategy was safe co-existence. Ethane [17] required no host modifications, and allowed its switches to be incrementally deployed alongside regular switches. OpenFlow, from the start, introduced hybrid switches that could operate both with Layer-2/3 control protocols or be managed by a controller, and had the requirement that OF switches would keep OpenFlow traffic isolated from production traffic [62]. Even in the case of incremental upgrades, these strategies are "dual-stack", meaning that the SDN and the traditional network are independent.

A migration approach that is feasible in fully virtualized environments is to run virtual SDN switches in the hypervisors in the edge, and provide network virtualization [20]. As noted in [58], this approach is not feasible in many enterprise and campus networks where the edge terminates in legacy access switches.

Panopticon [58] provides another migration strategy that is more integrated than a dual-stack approach. With strategic switch placement, it can almost match the benefits of a full SDN deployment for any flow that goes through at least one OpenFlow switch. With this, it provides the illusion that the entire network is a single SDN to controller applications.

Our work is related, yet orthogonal, to all of these approaches; all require the configurations and policies for the SDN controller be written afresh. Exodus automatically performs a partial migration of the existing configuration to an equivalent SDN setup.

Another approach to SDN migration is to progressively replace existing routers with functionally equivalent OpenFlow components, and then later benefit from the evolvability of such components. B4 [45], Google's SDN system for wide-area traffic engineering, used such a strategy to replace BGP border routers in their WAN with custom OpenFlow switches. They replaced the BGP logic in the routers with a Quagga BGP node and a proxy application between the two. In doing this, they had to migrate the BGP configuration from the routers to Quagga. RouteFlow [75] allows for a similar strategy. While in a different setting, our approach analogously allows us to *automatically* migrate the configuration of an IOS-based router to a combination of a controller and a set of flowtables on OpenFlow switches.

Benson *et al.* [7] describe an approach for mining high-level reachability policies from existing network configurations. Our approach also extracts policy from existing configurations, but, in contrast, converts the policy to a declarative program that implements the same policy. Our focus is also broader, including NAT, routing, and ACLs.

Capretta, et al. [16] describe and formally verify a conflict-detection algorithm for Cisco IOS firewall configurations. Their formalism encompasses only ACLs, not NAT, routing, or Layer-2 behavior, although their support for idiosyncratic IOS ACL features (such as matching ports greater than a fixed value) is superior to ours. Zhang, et al. [104] use SAT-solvers to perform analysis on existing firewall configurations and to synthesize equivalent, smaller configurations. Their techniques apply to a generalized abstract notion of firewall configuration, and do not take routing, NAT, or modifications to Layer-2 header information into account. Nelson, et al. [67] also compile IOS configurations to a logical formalism for verification and other analysis. Their compiler supports more IOS features than ours, such as static and pool NAT, but their focus is analysis of single packets at Layers-3 and 4, so their compiler does not address issues such as the translation of Layer-2 addresses when packets cross Layer-3 subnets. All these works, however, focus on translation for the purpose of analysis, not to generate code for execution.

As described in Chap. 8, Exodus produces Flowlog [68] programs which compile to NetCore, and then OpenFlow. Implementing the features described in this work required making several enhancements to Flowlog and NetCore. Flowlog did not originally provide access to ARP packet payloads; to create an ARP cache and proxy, we extended Flowlog with a general hierarchy of packet types. To translate ACLs and static routes, which can use address masking, we added support for matching IP address ranges (rather than only individual addresses) to both Flowlog and NetCore. In addition, we added an event type that allows Flowlog programs to react when OpenFlow table entries expire, with corresponding support in NetCore. We also enhanced the Flowlog compiler to support joins over multiple state relations, which were previously forbidden.

Finally, Header Space Analysis [50] and ATPG [103] could be used to verify the correctness of the Exodus-produced configurations, by analyzing both the original IOS configurations and the resulting OpenFlow rules. While this is relevant future work, HSA does not currently support all of the features of IOS that Exodus needs, and does not provide parsers for generic OpenFlow rules.

Chapter 11

Conclusion

This dissertation has presented two novel contributions, and respective prototypes, to software-defined networking. The first, PANE, allows network administrators to safely delegate their authority over the network's policy. The design and configuration of today's networks is already informed by application needs (*e.g.*, networks with full-bisection bandwidth for MapReduce-type frameworks, or deadline-based queuing [5] for interactive web services). PANE provides a way for the network to solicit and react to such needs automatically, dynamically, and at a finer timescale than with human input. To do this, our design overcomes the two challenges of decomposing network control, and resolving conflicts between users' needs.

However, before using a novel SDN controller such as PANE, many administrators will wish to migrate their existing configurations to the new platform. Our second contribution, Exodus, is the first SDN migration tool which directly migrates existing network policies to equivalent SDN controller software and an OpenFlow-based network configuration. Automatic migration allows network operators familiar with their own networks, but not SDN, to quickly explore the benefits of this new approach. By generating code in a high-level, rule-based language, Exodus makes it easy to bootstrap a new network controller which can evolve at the frenetic pace of enterprise network environments [51]. The high-level semantics of the generated program opens the avenue for change-impact analysis, and potential refactoring of the physical configuration of the network, bringing the full benefits of an SDN deployment. No matter the migration strategy eventually employed, Exodus gives network administrators a concrete, working prototype from which to begin discussion and compare solutions.

11.1 Bringing PANE to Flowlog

At present, controllers for software-defined networks are in their infancy, and most controllers are built to accomplish particular objectives: PANE, the challenge of policy delegation, and Exodus, the task of policy migration. As we consider the further growth and maturation of software-defined networking, we begin with the thought exercise of implementing PANE in Flowlog, the language in which Exodus controllers are generated.

Flowlog's initial design was heavily influenced by our experience building PANE. The PANE controller is fundamentally event-driven, as new network configurations are driven by the dynamic requests arriving from end users. It also stores a lot of state about the network: host locations, bandwidth availability, access control lists, switch and link statuses, timelines of future requests, and more. The central nature of both events and state made PANE a poor match for the existing SDN languages.

Because Flowlog was designed around events and state, implementing PANE in Flowlog might appear straightforward. Several components of the PANE runtime such as host discovery, topology discovery, spanning-tree construction, and forwarding decisions, could be replaced with well-built NIB and forwarding programs in the current Flowlog. Furthermore, much of the PANE API is exposed to its users assuming a "one big switch" abstraction of the network, which aligns with that of the NetCore policies Flowlog generates.

However, we find that Flowlog currently falls short in two key areas: composition, and routing. In Flowlog's present design, all event handlers are triggered at once – it is not possible to first evaluate an incoming packet with, for example, an access-control module (or PANE), and then second with a forwarding module. While the physical composition employed by Exodus circumvents this problem, PANE requires logical composition as its policies will not necessarily yield end-to-end routes for all packets.

Indeed, this example is directly related to Flowlog's second key shortcoming – lack of abstractions for routing – as PANE policies may, for example, simply declare that some packets should be waypointed through a particular switch before reaching their destination. Such a policy, like any which introduces topological constraints, would be challenging to implement in Flowlog, which only provides either a hop-by-hop or a "one big switch" approach to network programming.

11.2 Limitations of the "One Big Switch" Abstraction

PANE is not the only SDN application which cannot always be implemented in a "one big switch" view of the network. Other examples include:

- **Guaranteeing latency** – A flow's end-to-end latency is a function of many variables, including the choice of path. As such, an SDN application designed to manage flow latencies will require details of the network topology. For example, in a Clos datacenter network, latency is a function of the number of switches (and therefore links) a packet traverses.
- **Provisioning circuits** – Many scientific WANs are designed to provision optical circuits on demand, either for bulk data dissemination (*e.g.*, in ESNet) or to isolate experimental traffic (*e.g.*, in GENI). Such applications are a particular form of wide-area traffic engineering, and obviously break the "one big switch" abstraction.
- **Scheduling maintenance** – A controller application designed to seamlessly migrate traffic ahead of scheduled maintenance necessarily requires information about the network's physical infrastructure (*e.g.*, the bandwidth available on each link, which links will be taken offline by switch maintenance, which links share the same optical cable, etc.).

From these examples, we see that the "one big switch" abstraction works best under two assumptions: first, that the network is uniform, and second, that applications are topology-independent. When networks are not uniform (for example, a WAN with many different link costs, or a network with middleboxes along particular paths), or control applications make topology-dependent decisions (for example, to manage latency, or schedule maintenance) then the controller must expose topology information through a NIB, and should provide abstractions for path programming and routing.

11.3 Lessons from Building SDN Controllers

Realizing that SDN controllers should provide abstractions for path programming as well as for taking a "one big switch" view is not the only lesson we can draw from the development of PANE and Exodus. Upon reflection, the following list of thirteen features useful in almost every SDN controller emerges:

1. **Tables (state)** – Every network implementation contains many tables, which may be used for routing, forwarding, address translation, ARP caching, link-state, access control, and more.
2. **Events** – Networks are very dynamic, and control applications must respond to many types of events including switch or link up, switch or link down, host discovery, and external events.

3. **Policy composition** – Supporting policy composition using sequential, parallel, prioritized, and hierarchical merge strategies allows for the controller to be written in a modular fashion.

4. **Path-based reasoning** – Traffic engineering applications, and those discussed in the previous section, all use the network topology to make decisions and may introduce constraints on a packet's path through the network.

5. **Virtualization** – Virtualizing the network for SDN applications consists of two components:

    - **Indirection** – Writing a policy on an abstract view of the network (*e.g.*, "one big switch", or a particular slice [40])
    - **Isolation** – Policies written on disparate abstract views of the network should not interfere [79].

6. **Scalability** – The SDN platform should scale with the network and its demands, through federation [87], hierarchy [61, 99], and scalable state storage.

7. **Redundancy** – Redundant, fault-tolerant controllers are a must for any SDN platform in production.

8. **Verification** – Verifying control programs, as we do in Flowlog, should help to eliminate bugs in these critical infrastructure components.

9. **Debugging** – Useful SDN platforms will provide debugging functionality such as capturing packets, and replaying forwarding decisions [41, 95].

10. **Resource management** – Many network resources, such as per-switch flow tables, and per-port rate-limiters, are constrained. SDN control platforms will need to track the use of these resources, and are the right place to intelligently optimize their use [25, 46, 47, 100]. Ideally, such platforms could offer the illusion of infinite rule space on the switches [49], just as regular applications see virtual memory.

11. **Consistency** – Many SDN applications require some form of consistency in both the data-plane and the control-plane. Developing and implementing notions of consistency for SDN is ongoing [15, 59, 69, 74].

12. **Queries** – Measuring network state by querying the data-plane is a key component of many network applications [1, 10, 42, 45]. SDN control platforms will need intelligence to answer queries from multiple applications without overwhelming limited network hardware [76].

13. **Security** – Finally, like an OS kernel, the underlying SDN platform is responsible for enforcing network security guarantees irrespective of the applications in use [70, 81].

Today, most SDN controllers excel at only a handful of features on the list above, and none – to our knowledge – support them all; PANE and Exodus are but stepping-stones in the evolution of software-defined networks.

"The world only spins forward ... The Great Work Begins."

Tony Kushner, *Angels in America*

Bibliography

[1] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2010. Cited on [1, 13, 37, 76]

[2] `https://aws.amazon.com/message/65648/`. Cited on [1]

[3] David A. Applegate, Gruia Calinescu, David S. Johnson, Howard Karloff, Katrina Ligett, and Jia Wang. Compressing Rectilinear Pictures and Minimizing Access Control Lists. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2007. Cited on [60]

[4] F. Baker. Requirements for IP Version 4 Routers. RFC 1812, June 1995. Cited on [50]

[5] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. Towards Predictable Datacenter Networks. In *ACM Sigcomm*, 2011. Cited on [39, 73]

[6] Rob Barrett, Eser Kandogan, Paul P. Maglio, Eben M. Haber, Leila Takayama, and Madhu Prabaker. Field Studies of Computer System Administrators: Analysis of System Management Tools and Practices. In *Computer-Supported Cooperative Work and Social Computing*, pages 388–395, 2004. Cited on [4]

[7] Theophilus Benson, Aditya Akella, and David A. Maltz. Mining Policies from Enterprise Network Configuration. In *Internet Measurement Conference (IMC)*, 2009. Cited on [71]

[8] Theophilus Benson, Aditya Akella, and David A. Maltz. Unraveling the Complexity of Network Management. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2009. Cited on [3]

[9] Theophilus Benson, Aditya Akella, Anees Shaikh, and Sambit Sahu. CloudNaaS: A Cloud Networking Platform for Enterprise Applications. In *Symposium on Cloud Computing (SOCC)*, 2011. Cited on [39]

[10] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. MicroTE: Fine Grained Traffic Engineering for Data Centers. In *Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2011. Cited on [13, 76]

[11] Dario Bonfiglio, Marco Mellia, Michela Meo, and Dario Rossi. Detailed Analysis of Skype Traffic. *IEEE Trans. on Multimedia*, 11(1):117–127, 2009. Cited on [1, 32]

[12] Pat Bosshart, Dan Daly, Martin Izzard, Nick McKeown, Jennifer Rexford, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. Programming Protocol-Independent Packet Processors. *arXiv:1312.1719 [cs.NI]*, 2013. Cited on [51, 60]

[13] R. Braden, D. Clark, and S. Shenker. Integrated Services in the Internet Architecture: an Overview. RFC 1633, June 1994. Cited on [6]

[14] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource ReSerVation Protocol (RSVP). RFC 2205, September 1997. Cited on [6, 38]

[15] Marco Canini, Petr Kuznetsov, Dan Levin, and Stefan Schmid. Software Transactional Networking: Concurrent and Consistent Policy Composition. In *Workshop on Hot Topics in SDN (HotSDN)*, 2013. Cited on [76]

[16] Venanzio Capretta, Bernard Stepien, Amy Felty, and Stan Matwin. Formal Correctness of Conflict Detection for Firewalls. In *Workshop on Formal Methods in Security Engineering*, 2007. Cited on [61, 71]

[17] Martín Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking Control of the Enterprise. In *ACM Sigcomm*, 2007. Cited on [3, 61, 66, 68, 70]

[18] Martín Casado, Tal Garfinkel, Aditya Akella, Michael J. Freedman, Dan Boneh, Nick McKeown, and Scott Shenker. SANE: A Protection Architecture for Enterprise Networks. In *USENIX Security Symposium*, 2006. Cited on [3, 70]

[19] Martín Casado, Teemu Koponen, Rajiv Ramanathan, and Scott Shenker. Virtualizing the Network Forwarding Plane. In *Workshop on Programmable Routers for Extensible Services of Tomorrow (PRESTO)*, 2010. Cited on [67]

[20] Martín Casado, Teemu Koponen, Scott Shenker, and Amin Tootoonchian. Fabric: A Retrospective on Evolving SDN. In *Workshop on Hot Topics in SDN (HotSDN)*, 2012. Cited on [3, 68, 69, 70]

[21] Prashant Chandra, Allan Fisher, Corey Kosak, T. S. Eugene Ng, Peter Steenkiste, Eduardo Takashi, and Hui Zhang. Darwin: Resource Management for Value-added Customizable Network Service. In *IEEE International Conference on Network Protocols (ICNP)*, 1998. Cited on [38]

[22] Xu Chen, Yun Mao, Z. Morley Mao, and Jacobus Van der Merwe. Declarative Configuration Management for Complex and Dynamic Networks. In *Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2010. Cited on [39]

[23] Coq Development Team. The Coq Proof Assistant Reference Manual – Version 8.3. `http://coq.inria.fr/`, 2011. Cited on [23]

[24] Andrew R. Curtis, Wonho Kim, and Praveen Yalagandula. Mahout: Low-Overhead Datacenter Traffic Management using End-Host-Based Elephant Detection. In *IEEE Conference on Computer Communications (INFOCOM)*, 2011. Cited on [13]

[25] Andrew R. Curtis, Jeffrey C. Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee. DevoFlow: Scaling Flow Management for High-performance Networks. In *ACM Sigcomm*, 2011. Cited on [64, 76]

[26] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008. Cited on [35]

[27] David Erickson, Glen Gibb, Brandon Heller, David Underhill, Jad Naous, Guido Appenzeller, Guru Parulkar, Nick McKeown, Mendel Rosenblum, Monica Lam, Sailesh Kumar, Valentina Alaria, Pere Monclus, Flavio Bonomi, Jean Tourrilhes, Praveen Yalagandula, Sujata Banerjee, Charles Clark, and Rick McGeer. A Demonstration of Virtual Machine Mobility in an OpenFlow Network. In *ACM Sigcomm (Demo)*, 2008. Cited on [3]

[28] Nick Feamster and Hari Balakrishnan. Detecting BGP configuration faults with static analysis. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2005. Cited on [39]

[29] Andrew D. Ferguson, Arjun Guha, Chen Liang, Rodrigo Fonseca, and Shriram Krishnamurthi. Hierarchical Policies for Software Defined Networks. In *Workshop on Hot Topics in SDN (HotSDN)*, 2012. Cited on [4, 39]

[30] Andrew D. Ferguson, Arjun Guha, Chen Liang, Rodrigo Fonseca, and Shriram Krishnamurthi. Participatory Networking: An API for Application Control of SDNs. In *ACM Sigcomm*, 2013. Cited on [4, 65]

[31] Andrew D. Ferguson, Arjun Guha, Jordan Place, Rodrigo Fonseca, and Shriram Krishnamurthi. Participatory Networking. In *Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE)*, 2012. Cited on [4]

[32] Floodlight. `http://www.projectfloodlight.org/floodlight/`. Cited on [46]

[33] Nate Foster, Michael J. Freedman, Rob Harrison, Jennifer Rexford, Matthew L. Meola, and David Walker. Frenetic: A High-Level Language for OpenFlow Networks. In *Workshop on Programmable Routers for Extensible Services of Tomorrow (PRESTO)*, 2010. Cited on [4, 13, 19, 39]

[34] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A Network Programming Language. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2011. Cited on [46, 47, 65]

[35] `https://github.com/blog/1346-network-problems-last-friday`. Cited on [1]

[36] Simon Godik and Tim Moses (editors). eXtensible Access Control Markup Language, version 1.1, August 2003. Cited on [40]

[37] Albert Greenberg, Gisli Hjalmtysson, David A. Maltz, Andy Myers, Jennifer Rexford, Geoffrey Xie, Hong Yan, Jibin Zhan, and Hui Zhang. A Clean Slate 4D Approach to Network Control and Management. *ACM Computer Communication Review (CCR)*, 35:41–54, 2005. Cited on [2, 27, 39, 70]

[38] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. NOX: Towards an Operating System for Networks. *ACM Computer Communication Review (CCR)*, 38:105–110, July 2008. Cited on [39, 46]

[39] Arjun Guha, Mark Reitblatt, and Nate Foster. Machine-Verified Network Controllers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2013. Cited on [47]

[40] Stephen Gutz, Alec Story, Cole Schlesinger, and Nate Foster. Splendid Isolation: A Slice Abstraction for Software-Defined Networks. In *Workshop on Hot Topics in SDN (HotSDN)*, 2012. Cited on [76]

[41] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazieres, and Nick McKeown. I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2014. Cited on [76]

[42] Brandon Heller, Srini Seetharaman, Priya Mahadevan, Yannis Yakoumis, Puneet Sharma, Sujata Banerjee, and Nick McKeown. ElasticTree: Saving Energy in Data Center Networks. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2010. Cited on [76]

[43] Timothy L. Hinrichs, Natasha Gude, Martín Casado, John C. Mitchell, and Scott Shenker. Practical Declarative Network Management. In *Workshop on Research in Enterprise Networking (WREN)*, 2009. Cited on [39]

[44] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait free coordination for Internet-scale systems. In *USENIX Annual Technical Conference (ATC)*, 2010. Cited on [33]

[45] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a Globally-Deployed Software Defined WAN. In *ACM Sigcomm*, 2013. Cited on [3, 69, 71, 76]

[46] Nanxi Kang, Zhenming Liu, Jennifer Rexford, and David Walker. Optimizing the "One Big Switch" Abstraction in Software-defined Networks. In *Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2013. Cited on [67, 76]

[47] Yossi Kanizo, David Hay, and Isaac Keslassy. Palette: Distributing Tables in Software-Defined Networks. In *IEEE Conference on Computer Communications (INFOCOM)*, 2013. Cited on [67, 76]

[48] Naga Katta, Jennifer Rexford, and David Walker. Logic Programming for Software-Defined Networks. In *Workshop on Cross-Model Design and Validation (XLDI)*, 2012. Cited on [46, 47, 65]

[49] Naga Katta, Jennifer Rexford, and David Walker. Infinite CacheFlow in Software-Defined Networks. Technical Report TR-966-13, Department of Computer Science, Princeton University, October 2013. Cited on [76]

[50] Peyman Kazemian, George Varghese, and Nick McKeown. Header Space Analysis: Static Checking for Networks. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2012. Cited on [62, 72]

[51] Hyojoon Kim, Theophilus Benson, Aditya Akella, and Nick Feamster. The Evolution of Network Configuration: A Tale of Two Campuses. In *Internet Measurement Conference (IMC)*, 2011. Cited on [3, 60, 73]

[52] Hyojoon Kim, Arpit Gupta, Muhammad Shahbaz, Joshua Reich, Nick Feamster, and Russ Clark. Simpler Network Configuration with State-Based Network Policies. Technical Report GT-CS-13-04, Georgia Tech, 2013. Cited on [68]

[53] Wonho Kim, Puneet Sharma, Jeongkeun Lee, Sujata Banerjee, Jean Tourrilhes, Sung-Ju Lee, and Praveen Yalagandula. Automated and Scalable QoS Control for Network Convergence. In *Internet Network Management Workshop/Workshop on Research on Enterprise Networking (INM/WREN)*, 2010. Cited on [3, 38]

[54] Teemu Koponen, Keith Amidon, Peter Balland, Martín Casado, Anupam Chanda, Bryan Fulton, Igor Ganichev, Jesse Gross, Natasha Gude, Paul Ingram, Ethan Jackson, Andrew Lambeth, Romain Lenglet, Shih-Hao Li, Amar Padmanabhan, Justin Pettit, Ben Pfaff, Rajiv Ramanathan, Scott Shenker, Alan Shieh, Jeremy Stribling, Pankaj Thakkar, Dan Wendlandt, Alexander Yip, and Ronghua Zhang. Network Virtualization in Multi-tenant Datacenters. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2014. Cited on [47]

[55] Teemu Koponen, Martín Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. Onix: A Distributed Control Platform for Large-scale Production Networks. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2010. Cited on [28, 39]

[56] Leslie Lamport. The Part-Time Parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998. Cited on [33]

[57] Bob Lantz, Brandon Heller, and Nick McKeown. A Network in a Laptop: Rapid Prototyping for Software-Defined Networks. In *Workshop on Hot Topics in Networks (HotNets)*, 2010. Cited on [31, 57]

[58] Dan Levin, Marco Canini, Stefan Schmid, and Anja Feldmann. Panopticon: Reaping the Benefits of Partial SDN Deployment in Enterprise Networks. Technical report, TU Berlin / T-Labs, May 2013. Cited on [3, 42, 69, 70]

[59] Ratul Mahajan and Roger Wattenhofer. On Consistent Updates in Software Defined Networks. In *Workshop on Hot Topics in SDN (HotSDN)*, 2013. Cited on [76]

[60] J. Manner, G. Karagiannis, and A. McDonald. NSIS Signaling Layer Protocol (NSLP) for Quality-of-Service Signaling. RFC 5974, October 2010. Cited on [6, 38]

[61] James McCauley, Aurojit Panda, Martín Casado, Teemu Koponen, and Scott Shenker. Extending SDN to Large-Scale Networks. In *Open Networking Summit (ONS)*, 2013. Cited on [76]

[62] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. *ACM Computer Communication Review (CCR)*, 38:69–74, 2008. Cited on [2, 3, 39, 47, 70]

[63] Christopher Monsanto, Nate Foster, Rob Harrison, and David Walker. A Compiler and Run-time System for Network Programming Languages. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2012. Cited on [21, 47]

[64] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing Software-Defined Networks. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2013. Cited on [4, 46, 47, 65, 67]

[65] Jad Naous, Ryan Stutsman, David Mazières, Nick McKeown, and Nickolai Zeldovich. Enabling Delegation with More Information. In *Workshop on Research in Enterprise Networking (WREN)*, 2009. Cited on [37]

[66] Ankur Kumar Nayak, Alex Reimers, Nick Feamster, and Russ Clark. Resonance: Dynamic access control for enterprise networks. In *Workshop on Research in Enterprise Networking (WREN)*, 2009. Cited on [3, 39]

[67] Timothy Nelson, Christopher Barratt, Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. The Margrave Tool for Firewall Analysis. In *USENIX Large Installation System Administration Conference (LISA)*, 2010. Cited on [39, 53, 61, 71]

[68] Timothy Nelson, Andrew D. Ferguson, Michael J. G. Scheer, and Shriram Krishnamurthi. Tierless Programming and Reasoning for Software-Defined Networks. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2014. Cited on [4, 41, 46, 47, 65, 71]

[69] Peter Peresini, Maciej Kuzniar, Nedeljko Vasic, Marco Canini, and Dejan Kostic. OF.CPP: Consistent Packet Processing for OpenFlow. In *Workshop on Hot Topics in SDN (HotSDN)*, 2013. Cited on [76]

[70] Phillip Porras, Seungwon Shin, Vinod Yegneswaran, Martin Fong, Mabry Tyson, and Guofei Gu. A Security Enforcement Kernel for OpenFlow Networks. In *Workshop on Hot Topics in SDN (HotSDN)*, 2012. Cited on [76]

[71] POX. `http://www.noxrepo.org/pox/about-pox/`. Cited on [46]

[72] Google Protocol Buffers. `https://code.google.com/p/protobuf/`. Cited on [58]

[73] Barath Raghavan, Kashi Venkatesh Vishwanath, Sriram Ramabhadran, Ken Yocum, and Alex C. Snoeren. Cloud Control with Distributed Rate Limiting. In *ACM Sigcomm*, 2007. Cited on [12, 38]

[74] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for Network Update. In *ACM Sigcomm*, 2012. Cited on [28, 76]

[75] Christian Esteve Rothenberg, Marcelo Ribeiro Nascimento, Marcos Rogerio Salvador, Carlos Nilton Araujo Corrêa, Sidney Cunha de Lucena, and Robert Raszuk. Revisiting Routing Control Platforms with the Eyes and Muscles of Software-defined Networking. In *Workshop on Hot Topics in SDN (HotSDN)*, 2012. Cited on [69, 71]

[76] Charalampos Rotsos, Nadi Sarrar, Steve Uhlig, Rob Sherwood, and Andrew W. Moore. OFLOPS: An Open Framework for OpenFlow Switch Evaluation. In *Passive and Active Measurements Conference (PAM)*, 2012. Cited on [60, 64, 76]

[77] Arjun Roy, Stephen M. Rumble, Ryan Stutsman, Philip Levis, David Mazières, and Nickolai Zeldovich. Energy Management in Mobile Devices with the Cinder Operating System. In *European Conference on Computer Systems (EuroSys)*, 2011. Cited on [39]

[78] Scott Shenker. The future of networking and the past of protocols. Talk at Open Networking Summit, Oct. 2011. Cited on [67]

[79] Rob Sherwood, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martín Casado, Nick McKeown, and Guru Parulkar. Can the Production Network Be the Testbed? In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2010. Cited on [3, 39, 76]

[80] Alan Shieh, Emin Gün Sirer, and Fred B. Schneider. Netquery: A Knowledge Plane For Reasoning About Network Properties. In *ACM Sigcomm*, 2011. Cited on [13]

[81] Seungwon Shin, Phil Porras, Vinod Yagneswaran, Martin Fong, Guofei Gu, and Mabry Tyson. FRESCO: Modular Composable Security Services for Software-Defined Networks. In *Network and Distributed System Security (NDSS) Symposium*, 2013. Cited on [76]

[82] Emin Gün Sirer, Willem de Bruijn, Patrick Reynolds, Alan Shieh, Kevin Walsh, Dan Williams, and Fred B. Schneider. Logical Attestation: An Authorization Architecture For Trustworthy Computing. In *Symposium on Operating System Principles (SOSP)*, 2011. Cited on [17]

[83] P. Srisuresh and K. Egevang. Traditional IP Network Address Translator (Traditional NAT). RFC 3022, January 2001. Cited on [65]

[84] Ion Stoica, Hui Zhang, and T. S. Eugene Ng. A hierarchical fair service curve algorithm for link-sharing, real-time and priority services. In *ACM Sigcomm*, 1997. Cited on [38]

[85] Yu-Wei Eric Sung, Xin Sun, Sanjay Rao, Geoffrey Xie, and David A. Maltz. Towards Systematic Design of Enterprise Networks. *IEEE/ACM Transactions on Networking*, 19(3):695–708, 2011. Cited on [3]

[86] David L. Tennenhouse, Jonathan M. Smith, W. David Sincoskie, David Wetherall, and Gary Minden. A Survey of Active Network Research. In *IEEE Communications Magazine*, January 1997. Cited on [6, 37]

[87] Amin Tootoonchian and Yashar Ganjali. HyperFlow: A Distributed Control Plane for OpenFlow. In *Internet Network Management Workshop/Workshop on Research on Enterprise Networking (INM/WREN)*, 2010. Cited on [76]

[88] UPnP Device Architecture version 1.1. UPnP Forum., Oct. 2008. Cited on [38]

[89] Andreas Voellmy and Paul Hudak. Nettle: Taking the Sting Out of Programming Network Routers. In *Practical Aspects of Declarative Languages (PADL)*, 2011. Cited on [27, 39, 46, 65]

[90] Andreas Voellmy, Hyojoon Kim, and Nick Feamster. Procera: A Language for High-Level Reactive Network Control. In *Workshop on Hot Topics in SDN (HotSDN)*, 2012. Cited on []

[91] Andreas Voellmy, Junchang Wang, Y. Richard Yang, Bryan Ford, and Paul Hudak. Maple: Simplifying SDN Programming Using Algorithmic Policies. In *ACM Sigcomm*, 2013. Cited on [46, 47, 65]

[92] Guohui Wang, T. S. Eugene Ng, and Anees Shaikh. Programming Your Network at Run-time for Big Data Applications. In *Workshop on Hot Topics in SDN (HotSDN)*, 2012. Cited on [37]

[93] Richard Wang, Dana Butnariu, and Jennifer Rexford. OpenFlow-based server load balancing gone wild. In *Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE)*, 2011. Cited on [3]

[94] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowstron. Better never than late: Meeting deadlines in datacenter networks. In *ACM Sigcomm*, 2011. Cited on [13]

[95] Andreas Wundsam, Dan Levin, Srini Seetharaman, and Anja Feldman. OFRewind: Enabling Record and Replay Troubleshooting for Networks. In *USENIX Annual Technical Conference (ATC)*, 2011. Cited on [76]

[96] Hong Yan, David A. Maltz, T. S. Eugene Ng, Hemant Gogineni, Hui Zhang, and Zheng Cai. Tesseract: A 4D Network Control Plane. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2007. Cited on [39]

[97] Ziaowei Yang, David Wetherall, and Tom Anderson. A DoS-limiting Network Architecture. In *ACM Sigcomm*, 2005. Cited on [38]

[98] Kok-Kiong Yap, Te-Yuan Huang, Ben Dodson, Monica S. Lam, and Nick McKeown. Towards Software-Friendly Networks. In *Asia-Pacific Workshop on Systems (APSys)*, 2010. Cited on [38]

[99] Soheil Hassas Yeganeh and Yashar Ganjali. Kandoo: A Framework for Effhcient and Scalable Offjoading of Control Applications. In *Workshop on Hot Topics in SDN (HotSDN)*, 2012. Cited on [76]

[100] Minlan Yu, Jennifer Rexford, Michael J. Freedman, and Jia Wang. Scalable flow-based networking with difane. In *ACM Sigcomm*, 2010. Cited on [76]

[101] Minlan Yu, Xin Sun, Nick Feamster, Sanjay Rao, and Jennifer Rexford. A Survey of virtual LAN usage in campus networks. *Network & Service Management Series, IEEE Communications Magazine*, July 2011. Cited on [3]

[102] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *European Conference on Computer Systems (EuroSys)*, 2010. Cited on [35]

[103] Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. Automatic Test Packet Generation. In *Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2012. Cited on [3, 59, 62, 72]

[104] Shuyuan Zhang, Abdulrahman Mahmoud, Sharad Malik, and Sanjai Narain. Verification and Synthesis of Firewalls using SAT and QBF. *IEEE International Conference on Network Protocols (ICNP)*, 2012. Cited on [61, 71]