Integrating Network Management For Cloud Computing Services

Peng Sun

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Computer Science
Adviser: Professor Jennifer Rexford

June 2015

© Copyright by Peng Sun, 2015.

All rights reserved.

Abstract

Cloud computing is known to lower costs of corporate IT. Thus enterprises are eager to move IT applications into public or private cloud. Because of this trend, networks connecting enterprises and cloud providers now play a critical role in delivering high-quality cloud applications.

Simply buying better devices is not viable for improving network quality, due to high capital costs. A more attractive approach is to better utilize network resources with proper network management. However, there are two problems with current network management: separately managing network components along the end-to-end path, and heavily relying on vendor-specific interfaces with devices.

This dissertation takes a practical approach driven by operational experiences of cloud services to tackle the two problems. With knowledge of real-world challenges, we have designed proper abstractions for low-level device interactions, and have built efficient and scalable systems to integrate the management of various network components. With commercial deployment, our operational experiences feed back into revision of the abstraction and system design.

In this dissertation, we make three major contributions. We first propose to consolidate the traffic and infrastructure management in datacenters. Our system, called Statesman, simplifies management solutions by providing a uniform abstraction to interact with various aspects of devices. Statesman then allows multiple solutions to run together, resolves their conflicts, and prevents network-wide failures caused by their collective actions. Statesman has been operational worldwide in Microsoft's public cloud offering since October 2013.

The second contribution consists of joining end hosts with networks for cooperative traffic management. Our Hone system brings in the fine-grained knowledge of cloud applications in the hosts, and offers an expressive programming framework with a

uniform view of both host and network data. Hone has been integrated into Verizon Business Cloud.

The final contribution consists of bridging enterprises and Internet service providers (ISPs) for fine-grained control of inbound traffic from cloud applications. Our Sprite system enables enterprises to directly decide how traffic enters the enterprise networks via which ISPs, offering expressive interface and scalable execution. In collaboration with Princeton's Office of Information Technology, Sprite was tested with campus-network data and live Internet experiments.

Acknowledgments

I am very grateful to have had Jennifer Rexford as my advisor through the ups and downs in my PhD journey. Jen has been a great advisor on computer-networking research. Her solid expertise and sharp intellect have guided me through the challenges of research, e.g., brainstorming new ideas, clarifying research problems, delving into technical solutions, improving paper writing, making engaging presentation, etc. Also being a mentor for me, Jen encouraged me to explore different career paths, and help me discover and start the best one for my career development. Thanks to her best imaginable supports, I spent significant portion of my time on industrial research projects seeking both academic and production impacts, and I built my confidence and passion in practically applying technologies from cutting-edge research. I have always been inspired by her thoughtfulness, humility, and kindness to everyone around her, which make her my role model for years to come. My five years at Princeton were one of the best times in my life because of the fortune of working with and learning from Jen.

I am also very fortunate to have worked with Lihua Yuan and Dave Maltz. As mentors of my two-year internship in Microsoft Azure, they put enormous trust on me to drive a complete roadmap of applying my research: from architecting solutions, securing resources from the management layer, to developing and delivering the product. Their rigorous attitude and expert skills in solving practical problems set a great example for my career.

I would like to thank David Walker, Nick Feamster, Margaret Martonosi, and Kai Li for serving on my thesis committee, and giving me valuable feedback on my works.

I also want to thank my colleagues for the papers we co-authored: Ming Zhang, Ratul Mahajan, Ahsan Arefin, David Walker, Minlan Yu, Michael Freedman, and Laurent Vanbever. The major chapters of this dissertation would not be possible without their invaluable helps and contributions. Our collaboration has been a truly

inspiring and rewarding experience. I also would like to thank Chuanxiong Guo, Guohan Lu, Randy Kern, Albert Greenberg, Walter Willinger, and many others for their valuable feedback to my research.

My research at Princeton was supported by the National Science Foundation [34, 52], Intel [33], DARPA [35], and the Office of Naval Research [101]. My research at Microsoft was supported by Microsoft Azure and Microsoft Research. Many thanks to them for making my works possible.

My Princeton experience would have been incomplete without my friends. I benefited a lot from the candid conversations and the fun social activities with the members of our Cabernet group. Minlan Yu, Eric Keller, and Rob Harrison helped me boot the graduate student career in my first year on both research and life. In the same class with me, Srinivas Narayana gave me kindhearted supports in many joys and struggles that we shared in the graduate school. I would also like to thank Xin Jin, Naga Katta, Nanxi Kang, Mojgan Ghasemi, Arpit Gupta, Theophilus Benson, Laurent Vanbever, and Joshua Reich for the fun discussions on my work and life in Princeton.

I also want to thank my friends in the rest of the Computer Science Department for making me feel like home: Jude Nelson, Matt Zoufly, Jeff Terrace, Wyatt Lloyd, David Shue, Erik Nordstrom, Xiaozhou Li, Tianqiang Liu, Yiming Liu, Jingwan Lu, Xiaobai Chen, Tianlong Wang, Sapan Bhatia, Andy Bavier, Christopher Tengi, Yida Wang, Cole Schlesinger, Christopher Monsanto, and many others. I also owe a special thank you to Melissa Lawson, our retired graduate coordinator, for her kind helps on making my extended internship possible. I thank Brian Kernighan for helping me improve the communication and teaching skills.

I also thank my friends in other departments in Princeton (Zhuo Zhang, Zhikai Xu, Shaowei Ke, Xiaochen Feng, Haoshu Tian, Liechao Huang, Pingmei Xu), in Microsoft (George Chen, Huoping Chen, Murat Acikgoz, Kamil Cudnik, Jiaxin Cao, Shikhar

Suri, Dong Xiang, Chao Zhang, Varugis Kurien, Hongqiang Liu, Yibo Zhu), and too many others to mention for making my life in Princeton and Seattle much more fun.

I thank my parents, Shufang Ma and Shusen Sun, for their enduring love and belief, without which I would not have finished the journey.

Above all, I want to thank my fiancée, Jiaxuan Li, for her love and unwavering faith in me. It is the happiness and optimism from her that drives me through the five years of PhD. I dedicate this dissertation to her.

To my fiancée and my parents.

Contents

Abstract . . . iii

Acknowledgments . . . v

List of Tables . . . xiii

List of Figures . . . xiv

1 Introduction **1**

1.1 Problems of Current Network Management . . . 4

1.2 Research Approach . . . 7

1.3 Contributions . . . 8

1.3.1 Safe Datacenter Traffic/Infrastructure Management . . . 9

1.3.2 End-host/Network Cooperative Traffic Management . . . 10

1.3.3 Direct Control of Entrant ISP for Enterprise Traffic . . . 10

2 Statesman: Integrating Network Infrastructure Management **12**

2.1 Introduction . . . 12

2.2 Network State Abstraction . . . 17

2.2.1 Three Views of Network State . . . 17

2.2.2 Dependency Model of State Variables . . . 18

2.2.3 Application Workflow . . . 19

2.3 System Overview . . . 20

2.4 Managing Network State . . . 23

2.4.1 The State Dependency Model 23
2.4.2 Using and Extending the Dependency Model 25
2.5 Checking Network State 27
2.5.1 Resolving Conflicts 27
2.5.2 Choosing and Checking Invariants 29
2.5.3 Partitioning by Impact Group 31
2.6 System Design and Implementation 32
2.6.1 Globally Available and Distributed Storage Service 32
2.6.2 Stateless Update on Heterogeneous Devices 34
2.6.3 Network Monitors 36
2.6.4 Read-Write APIs 36
2.7 Operational Experiences 37
2.7.1 Deployment in Microsoft Azure 38
2.7.2 Maintaining Network-wide Invariants 39
2.7.3 Resolving Conflicts of Management Solutions 41
2.7.4 Handling Operational Failures 43
2.8 System Evaluation 45
2.9 Related Work 49
2.10 Conclusion 50

3 Hone: Combining End Host and Network for Traffic Management 51

3.1 Introduction 51
3.2 Hone Programming Framework 55
3.2.1 Measurement: Query on Global Tables 56
3.2.2 Analysis: Data-Parallel Operators 59
3.2.3 Control: Uniform and Dynamic Policy 62
3.2.4 All Three Stages Together 64
3.3 Efficient and Scalable Execution 64

3.3.1 Distributed Directory Service . . . 65
3.3.2 Lazily Materialized Tables . . . 66
3.3.3 Host-Controller Partitioning . . . 67
3.3.4 Hierarchical Data Aggregation . . . 68
3.4 Performance Evaluation . . . 70
3.4.1 Performance of Host-Based Measurement . . . 71
3.4.2 Performance of Management Solutions . . . 73
3.4.3 Effects of Lazy Materialization . . . 75
3.4.4 Evaluation of Scalability in Hone . . . 76
3.5 Case Studies . . . 80
3.5.1 Elephant Flow Scheduling . . . 80
3.5.2 Distributed Rate Limiting . . . 81
3.6 Related Work . . . 83
3.7 Conclusion . . . 84

4 Sprite: Bridging Enterprise and ISP for Inbound Traffic Control 86
4.1 Introduction . . . 86
4.2 Inbound TE Using Source NAT . . . 89
4.3 Scalable Sprite Architecture . . . 90
4.3.1 Data Plane: Edge Switches Near Hosts . . . 91
4.3.2 Control Plane: Local Agents Near Switches . . . 93
4.4 Dynamic Policy Adaptation . . . 94
4.4.1 High-level Traffic Engineering Objectives . . . 94
4.4.2 Computing Network Policy . . . 95
4.5 Implementation . . . 98
4.5.1 Design for Fault Tolerance . . . 98
4.5.2 How Components Communicate . . . 99
4.5.3 Routing Control for Returning Packets . . . 100

4.5.4 BGP Stability 101
4.6 Evaluation 102
4.6.1 Princeton Campus Network Data 102
4.6.2 Multi-ISP Deployment Setup 104
4.6.3 Inbound-ISP Performance Variance 105
4.6.4 Effects of Dynamic Balancing 106
4.7 Related Work 107
4.8 Conclusion 108

5 Conclusion 109

5.1 Summary of Contributions 110
5.2 Open Issues and Future Works 111
5.2.1 Combining Statesman and Hone in Datacenters 111
5.2.2 Supporting Transactional Semantics in Statesman 112
5.2.3 Hone for Multi-tenant Cloud Environment 112
5.3 Concluding Remarks 113

Bibliography 114

List of Tables

2.1 Input and Output of Each Component in Statesman 22

2.2 Examples of Network State Variables 25

2.3 Read-Write APIs of Statesman 37

3.1 Global Tables Supported in Hone Prototype 57

3.2 Measurement Query Language Syntax 59

3.3 Control Policy in Hone Prototype 63

3.4 Average CPU and Memory Usage of Execution 75

3.5 Average CPU/Memory Usage with Hierarchical Aggregation 80

3.6 Hone-based Traffic Management Solutions 81

4.1 Syntax of High-level Objective 95

4.2 Total Inbound Volume Distribution among ISPs 103

List of Figures

1.1 End-to-end Structure of Cloud-based Software Services 2

1.2 Practical and Iterative Research Approach 7

1.3 Scope of Integration for Projects in the Dissertation 9

2.1 Example of Conflicts between Management Solutions 14

2.2 Example of Safety Violation by Collective Actions 15

2.3 Statesman Architecture Overview 21

2.4 Network State Dependency Model 23

2.5 Flow of The Checker's Operation 28

2.6 Statesman System Design 33

2.7 Network Topology for The Scenario in §2.7.2 39

2.8 Device Upgrade Process while Statesman Maintains Safety 40

2.9 WAN Topology for The Scenario in §2.7.3 42

2.10 Conflict-Free TE and Device Upgrade with Statesman 43

2.11 Time Series of Firmware Upgrade at Scale 44

2.12 External-Force Failure in Device Upgrade 45

2.13 End-to-end Latency Breakdown 46

2.14 Network State Scale & Checker Performance 47

2.15 Read-write Micro-benchmark Performance 48

3.1 Overview of the Hone System 54

3.2 Three Stages of Traffic Management 56
3.3 Partitioned Execution Plan of Elephant-Flow Solution 68
3.4 Aggregation Tree: 8 Hosts with Branching of 2 70
3.5 Overhead of Collecting Connection Statistics 72
3.6 Latency of One Round of Execution of Management Solutions 75
3.7 Breakdown of Execution Latency 76
3.8 Effects of Lazy Materialization 77
3.9 Buffering Delay of Merging Data from Hosts on Controller 78
3.10 End-to-end Execution Latency with Hierarchical Aggregation 79
3.11 Time Series of Application Throughput 83

4.1 Example of How Sprite Works 90
4.2 Three Levels of Abstraction in Sprite 91
4.3 Sprite System Components 92
4.4 Workflow of Network Policy Adaptation 96
4.5 Network Policy Adaptation for Dynamic Perf-driven Balancing 97
4.6 System Architecture of Sprite Implementation 99
4.7 Stacked Chart of Inbound and Outbound Traffic Volume 103
4.8 Stacked Chart of Inbound Traffic via Three ISPs 104
4.9 Setup of the Multihomed Testbed on AWS VPC 105
4.10 Histogram of Video Quality via Two ISPs 106
4.11 Time Series of Average Per-User Throughput of YouTube 107

Chapter 1

Introduction

Cloud computing is reshaping the Information Technology (IT) industry. It offers utility computing by delivering the applications as services over the Internet and providing these services with well-organized hardware and software in datacenters. The utility computing model eliminates the large capital barrier of purchasing hardware for enterprises, and it also lowers the IT operational costs by allowing enterprises to pay for what they actually use. These benefits motivate an ongoing effort of enterprises to move their IT applications into the public cloud and/or build their private cloud [1, 15, 23].

The delivery of the promise of cloud computing depends on the quality of the end-to-end network. As shown in Figure 1.1, the Internet now plays a critical role in carrying the traffic of IT applications between enterprises and datacenters of public/private cloud providers. The performance of cloud-based IT applications depends on not only the application software in the cloud, but also the reliability, efficiency, and performance of the networks in the middle.

Improving the network quality could be achieved by deploying more network devices with higher bandwidth. However, this brute-force approach no longer works due to the high capital costs and the rapidly growing traffic demands [7]. A more

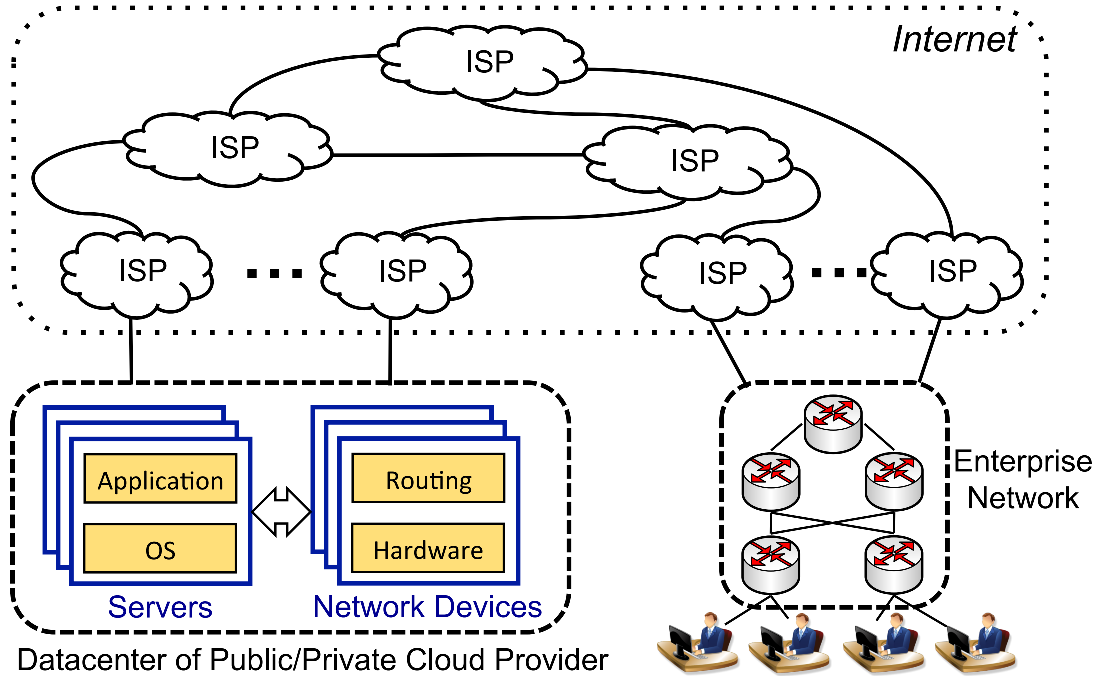

Figure 1.1: End-to-end Structure of Cloud-based Software Services

attractive approach is to build proper network management solutions to better utilize the existing network resources. Yet the current practice of network management has two main problems which have become the main bottlenecks in adopting cloud computing. The two problems of network management solutions are:

- **Disjoint management of the end-to-end components:**

  Providing an efficient network involves multiple components on the end-to-end path: from the network stack of the datacenter servers' operating systems, the hardware configuration and the routing control of network devices in datacenters, the traffic exchange of Internet service providers (ISPs), to the setup of enterprise networks. Each component can affect the quality of the networks. Yet these components (e.g., servers, network device hardware, and traffic routing) are managed separately with different systems, since traditionally these components are spread across multiple places. In the cloud era, these components become much more

concentrated in the datacenters than before, and the lack of integration among management systems limits the quality improvement of the networks.

- **Low-level interfaces for interacting with network devices:**
  Network devices are heterogeneous with different models from various vendors of varying ages. Interacting with the devices is complicated since the configuration APIs with devices are usually low-level and vendor-specific. Human network operators have to heavily use these low-level APIs in day-to-day operations, and it has been an error-prone process to configure the devices to run the right protocols with the right parameters using the right APIs. Using these APIs also tightly binds the solutions to specific device features by vendors, making it difficult to adapt the solutions to evolving business objectives. In datacenters, the heavy reliance on low-level device interfaces becomes one of the major sources of failures in network operations, especially when the datacenter network is growing in scale and adopting more commodity hardware from multiple vendors [8, 27, 28].

The programmability of management solutions has received much attention in the research community [64]. The concept of Software-Defined Networking (SDN) aims at providing a better way to program traffic management solutions. The literature on SDN, especially the ones surrounding the OpenFlow technology [45, 74, 97], promise to automate managing the traffic routing in networks with high-level programming paradigms [66, 67, 99, 128]. With existing literature focusing on programming traffic management on network devices, two other problems are much less explored beyond just traffic management: disjoint management of network components (e.g., server, device hardware), and low-level device interaction limiting a broader scope of network management (e.g., infrastructure management).

This dissertation focuses on solving these two problems. Rather than proposing clean-slate solutions, the dissertation takes a practical approach driven by operational experiences of cloud services. We identify the real-world opportunities and challenges

of integrating the management of different network components on the end-to-end path, and then propose proper abstractions to unify and hide the low-level component interactions. Guided by the abstractions, we design and build systems of integrated management platforms for cloud providers and enterprises that are simple to use, and safe, efficient, and scalable in day-to-day operations [120, 123]. Having deployed the systems in major cloud providers [17, 26], we leverage the operational experiences as feedback to revisit the abstraction and system design.

We first elaborate the problems of network management with examples of management solutions in §1.1. We then discuss the research approach of this dissertation in §1.2, and summarize the major contributions of this dissertation in §1.3.

1.1 Problems of Current Network Management

Current practices of network management have two main problems: 1) disjoint management of the network components, and 2) heavy reliance on the low-level interfaces with devices. These problems become more severe in the era of cloud computing, and have become the main limiting factors in building more reliable and more efficient network management solutions.

Considering the first problem, many components are involved in the end-to-end path of cloud applications. Applications are running on the servers in datacenters, and the servers send traffic to other servers or the outside via datacenter networks. A series of ISPs deliver the application traffic to the enterprise networks, which finally deliver it to the users. Traditionally, these components are managed as three separate areas: *1) Application and server provisioning* for placing the applications and controlling the servers' operating environments, *2) network infrastructure* for configuring and operating the device hardware from power control to topology setup, and *3) traffic engineering* for routing the application traffic through the network infrastructure.

The status quo results from the traditional division of labor among the entities of the Internet business in the pre-cloud-computing era. In the past, the entity who hosts applications has the majority of traffic going directly to users; not among the application instances inside. Thus it focuses on server provisioning with lower stakes in network operations. In contrast, the ISPs focus on the networks, separating infrastructure management and traffic engineering because the two run at vastly different paces (e.g., infrastructure changes over years while traffic changes in minutes).

The emergence of the cloud breaks the balance. The datacenters of cloud providers see a new majority of traffic coming from the server-to-server communication internally among application instances [41, 83], because more applications are built with multi-tier architecture as distributed systems [43, 57]. This new traffic pattern motivates big changes of datacenter network architecture with intensive usage of commodity devices [38, 73, 76, 77, 87, 104]. The architectural change significantly increases the quantity and the evolution speed of network infrastructure in datacenters. As interest in application provisioning, infrastructure management, and traffic engineering concentrates on the cloud providers, the current separation of management systems becomes a bottleneck in improving the performance of applications.

Additionally, the cloud-based applications have highly asymmetric traffic patterns to the enterprises [7]. For instance, our measurements at Princeton University show that the campus receives an average of eight times more traffic than it sends. Enterprises would like to control the incoming traffic, but they lack direct control because the function of traffic engineering mainly belongs to the ISPs [51].

In fact, there exists a great opportunity to consolidate disjoint management systems, especially within the scope of cloud providers. For instance, Google [82] and Microsoft [81] have both built traffic management systems on their wide area networks that combine the knowledge and functions of servers and networks. The systems collect application traffic demands directly from servers, and combine them with the

network statistics to compute a solution to scheduling the traffic. The systems then enforce the solution with both the rate limiting on servers and the routing control in networks. These integrated systems have significantly improved the bandwidth utilization levels of the wide area networks of Google and Microsoft.

Besides the separation problem, network management is also limited by the low-level interfaces with heterogeneous devices. Each device vendor develops their own interfaces with their network devices. As device hardware grows more specialized when vendors differentiate themselves, interfaces become tightly bound with specific vendors and specific hardware features. As specialized devices are heavily used, the network operators have to rely on low-level vendor APIs to configure and run the networks. This makes network operation heavily dependent on the experiences and manual control of the network operators.

Use of low-level device interfaces by network operators becomes a barrier to automating management of datacenter networks of cloud providers. Because of the architectural changes, the number of network devices is much larger in datacenters of cloud providers than in traditional networks. The management process has to be automated with software, rather than by manual configurations to be scalable and sustainable for future growth. Also, datacenters of cloud providers use large amounts of commodity devices, rather than specialized hardware from specific vendors, to save the purchase and operational costs for the economy of scale. The difference in interfaces among vendors and models of hardware actually becomes an obstacle in developing network management solutions for cloud services.

Major cloud providers have already started to reduce their reliance on vendor-specific APIs, and use commodity device features to build their management solutions with the help of server-based software [68, 107, 132]. For example, load balancing used to be carried out by specialized devices. However, configuring and maintaining the hardware load balancers is costly, and is the major source of operation failures. In-

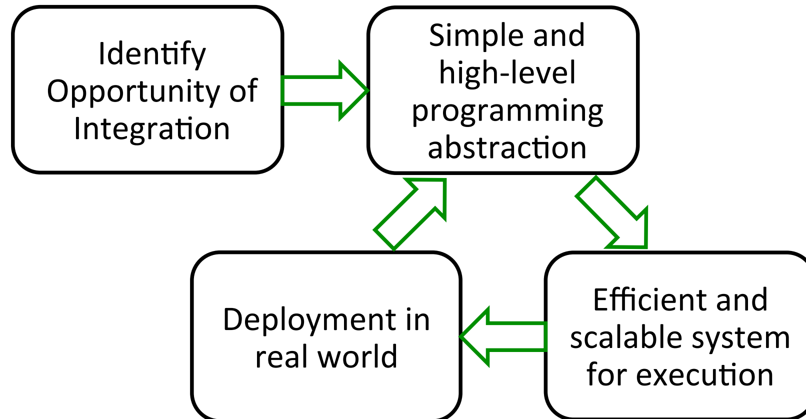

Figure 1.2: Practical and Iterative Research Approach

stead, Microsoft and Amazon [2] build software-based load balancing systems. These systems use the basic features of network devices (e.g., ECMP), and run software on a group of servers to distribute the traffic among application instances. Such systems greatly improve the stability of the cloud services by reducing operational failures.

1.2 Research Approach

Instead of designing clean-slate solutions, we take a practical approach driven by operational experience in cloud computing services. Figure 1.2 outlines the iteration cycle of our approach.

The first step is to identify real-world management scenarios where an integrated management system could benefit. We work closely with major cloud providers and enterprises to find problems that they encounter in the network operations. We then examine these problems, and understand how we can improve the operations by rethinking the division of labor.

Upon finding an opportunity for improvement, we start the iteration of designing and building a network management platform for integration. The design starts with high-level abstraction to unify the differences among the network components or the management entities, and hide the low-level interactions with device hardware. The

design of abstraction aims at a simple-to-use programming interface to build network management solutions which can utilize the benefits from the integration.

Based on the design of the programming abstraction, we then design the system part and build the management platform to execute the solutions across the newly consolidated network components. The goals of our systems are safety, efficiency, and scalability in the operations of datacenter networks in the cloud providers and enterprise networks of the cloud-service users.

In the process of building our management platforms, we actively seek deployment in commercial environments. Collaborating with several cloud providers and enterprises, we run the platforms to accumulate operational experiences. These experiences and the real-world constraints of deployment are fed back to the design of abstraction and systems to make our platforms more practical.

This research approach represents our careful balance between the advancement of research frontier in the area of network management and the engineering efforts in impacting the commercial operations of the cloud computing services.

1.3 Contributions

In this dissertation, we have designed and built three integrated management platforms to consolidate three areas along the end-to-end path of cloud services. Figure 1.3 shows which network components each platform has consolidated. Our works simplify and improve the traffic and infrastructure management in datacenters of cloud providers, and provide enterprises with more direct control over incoming traffic from cloud services. We make three major contributions as illustrated below.

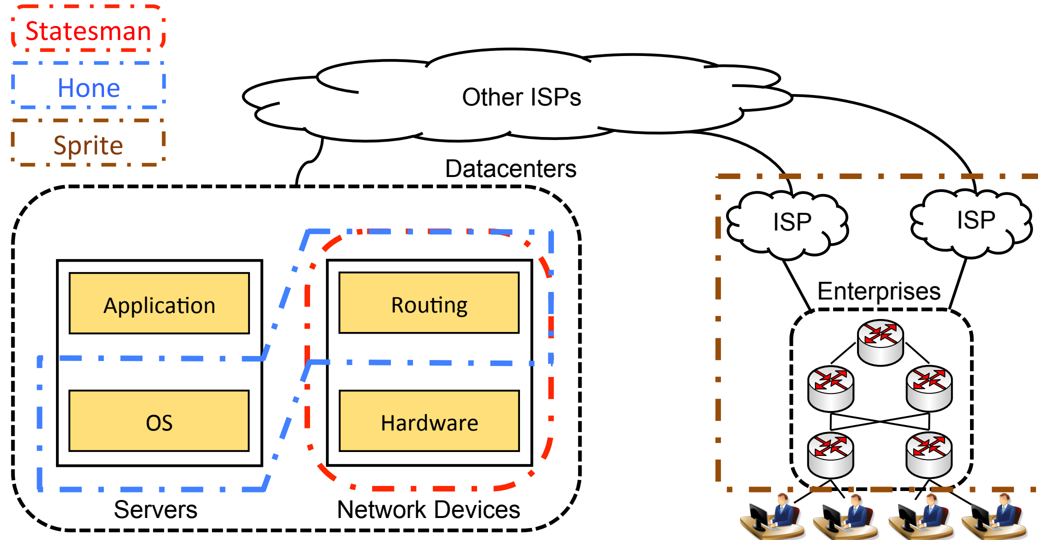

Figure 1.3: Scope of Integration for Projects in the Dissertation

1.3.1 Safe Datacenter Traffic/Infrastructure Management

The cloud providers run many automated solutions simultaneously for traffic and infrastructure management in datacenters. Individually complicated to build and operate, these solutions can have conflicts which inadvertently affect the operations of each other, and they can collectively cause unintentional network-wide failures.

We propose a system called Statesman to consolidate the operations of traffic and infrastructure management solutions, and offer a common layer of interacting with network devices to resolve conflicts and maintain network-wide safety invariants (e.g., maintaining the network connectivity). The core feature of Statesman is the loosely coupled model of running multiple management solutions with the abstraction of network state. Network state captures various aspects of the network, such as which links are alive and how devices are forwarding traffic. Statesman offers three distinct views of the network state as a pipeline to build and run management solutions. Working together with Microsoft, we have deployed the Statesman system in the datacenters of the Azure public cloud service worldwide. Operational since October

2013, Statesman becomes the fundamental layer for Microsoft Azure networking. The work is published in ACM SIGCOMM 2014 [120].

1.3.2 End-host/Network Cooperative Traffic Management

The providers of cloud services and cloud-based applications carefully manage their traffic through the underlying networks for various performance objectives. These traffic management solutions are confined in the scope of network devices. In addition to the limited CPU and memory resources, network devices cannot provide knowledge in layers higher than the network layer, limiting the solutions' insights into the application-traffic behaviors.

We propose to join the end hosts with the network devices, so we can utilize the rich application-traffic statistics in the end hosts to build better traffic management solutions. Our system, named Hone, abstracts the diverse collection of statistics on both end hosts and network devices into a uniform view of data. We then design a framework based on functional reactive programming [61, 103] to simplify programming management solutions with the uniform data model. Hone has been adopted by Overture Networks to use in the Verizon Business Cloud service. With the host-side data from Hone, the customers of Verizion Business Cloud enjoy better quality of their connections with the datacenters of Verizon for the improved performance of cloud-based applications. Our work on Hone is published in Springer Journal of Network and Systems Management [122, 123].

1.3.3 Direct Control of Entrant ISP for Enterprise Traffic

With the rise of cloud computing services, enterprise networks receive much more traffic than they send. Although enterprise networks typically connect with multiple upstream ISPs, they have very limited control over which of their ISPs to carry the incoming traffic for each cloud application. The limit of control comes from the

functionalities divided between the enterprises and the ISPs in routing the traffic, i.e., the path that the traffic takes is mainly decided by the ISPs.

We propose to bridge the boundary between enterprises and their upstream ISPs to exert direct and fine-grained control over the entrant ISP for incoming traffic. In our proposed system called Sprite, we design three levels of abstraction to simplify the expression of traffic engineering objectives with the highest level of abstraction, and dynamically translate the objectives into lower levels of abstraction for efficient and scalable execution. In collaboration with the Office of Information Technology of Princeton University, we have tested Sprite with the traffic data collected on the campus network and with live Internet experiments on the PEERING testbed [115, 125]. The work is published in ACM SIGCOMM Symposium on SDN Research 2015 [121].

Chapters 2, 3, and 4 describe Statesman, Hone, and Sprite in detail respectively. Chapter 5 presents open issues and future works, and concludes the dissertation.

Chapter 2

Statesman: Integrating Network Infrastructure Management

2.1 Introduction

Today's cloud-based applications (e.g., search, social networking, and online storage) depend on large datacenter networks. Keeping these networks running smoothly is difficult, due to the sheer number of network devices, and the dynamic nature of the environment. At any given moment, multiple network devices may experience component failures, may be brought down for planned maintenance or saving energy, may be upgraded with new firmware, or may be reconfigured to adapt to prevailing traffic demand. In response, the cloud providers have developed an array of automated systems for managing the *traffic* (e.g., traffic engineering [81, 82], server load balancing [68, 107], and network virtualization [89]) and the *infrastructure* (e.g., hardware power control for failure mitigation or energy saving [78, 132], device firmware upgrade, and device configuration management [48, 49]).

Each management solution is highly sophisticated in its own right, usually requiring several years to design, develop, and deploy. It typically takes the form of

a "control loop" that measures the current state of the network, performs computation, and then reconfigures the network. For example, a traffic engineering (TE) solution measures the current traffic demand and network topology, solves an optimization problem, and then changes the routing configuration to match demand. These solutions are complicated because they must work correctly even in the presence of failures, variable delays in communicating with a distributed set of devices, and frequent changes in network conditions.

Designing and running a single network management solution is challenging. Large datacenter networks must simultaneously run multiple management solutions—created by different teams, each reading and writing some part of the network state. For instance, both a TE solution and a solution to mitigate link failures need to run continuously to, respectively, adjust the routing configuration continuously and detect and resolve failures quickly.

These management solutions can conflict with each other, even if they interact with the network at different levels, such as establishing network paths, assigning IP addresses to interfaces, or installing firmware on devices. One solution can inadvertently affect the operation of another. As an example in Figure 2.1, suppose the TE solution wants to create a tunnel through the switch B, while the firmware-upgrade solution wants to upgrade B. Depending on which action happens first, either the TE solution fails to create the tunnel (because B is already down), or the already-established tunnel ultimately drops traffic during the firmware upgrade.

Running multiple management solutions also raises the risk of network-wide failures because their complex interactions make it hard to reason about their combined effect. Figure 2.2 shows an example where one management solution wants to shut down switch $AggB$ to upgrade its firmware, while another wants to shut down switch $AggA$ to mitigate packet corruption. While each solution acting alone is fine, their joint actions would disconnect the top-of-rack (ToR) switches. To prevent such disas-

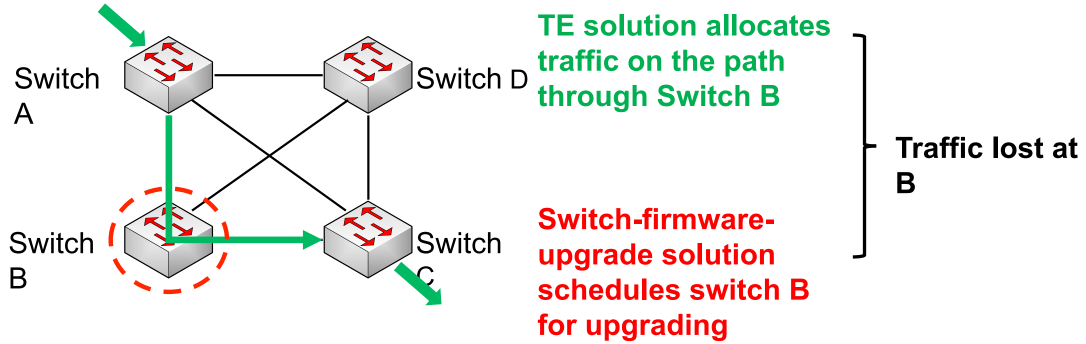

Figure 2.1: Example of Conflicts between Management Solutions

ters, it is imperative to ensure that the collective actions of the management solutions do not violate certain network-wide invariants, which specify basic safety and performance requirements for the network. For instance, a pod of servers must not be disconnected from the rest of datacenter, and there must be some minimum bandwidth between each pair of pods.

The cloud providers could conceivably circumvent the problems that stem from running multiple solutions by developing a single management solution that performs *all* functions, e.g., combining TE, firmware upgrade, and failure mitigation. However, this monolithic solution would be highly complex, and worse, it would need to be extended repeatedly as new needs arise. Thus, the cloud providers need a way to keep the management solutions separate.

Another option is to have explicit coordination among the management solutions. Corybantic [98] is one recent proposal that follows this approach. While coordination may be useful for some management solutions, using it to solve the problem of multi-solution coexistence imposes high overhead on each solution, requiring each solution to understand the intended network changes of all others. To make matters worse, every time a solution is changed or a new one is developed, the cloud providers would need to test again, and potentially retrofit some existing solutions, in order to ensure that all of them continue to coexist safely.

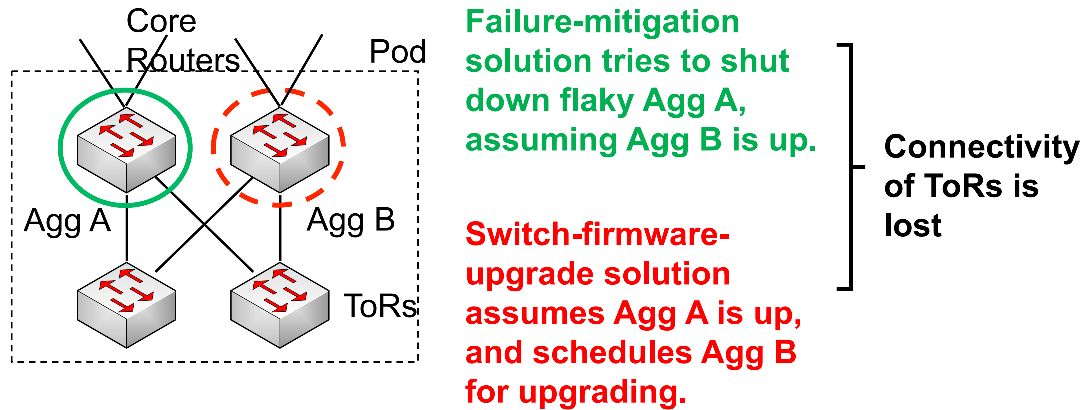

Figure 2.2: Example of Safety Violation by Collective Actions

We argue that network management solutions should be built and run in a loosely coupled manner, without explicit or implicit dependencies on each other, and conflict resolution and invariant enforcement should be handled by a separate management system. This architecture would simplify development of management solution, and its simplicity would boost network stability and predictability. It may forgo some performance gains possible through tight coupling and joint optimization. However, as noted above, such coupling greatly increases the complexity of management solution. Furthermore, since management solutions may have competing objectives, a management system to resolve conflicts and maintain invariants would be needed anyway. We thus believe that loose coupling of management solutions is a worthwhile tradeoff in exchange for significant reduction in complexity.

We propose Statesman, a network-state management service that supports multiple loosely-coupled management solutions in large datacenter networks. Each solution operates by reading and writing some part of the network state at its own pace, and Statesman functions as the conflict resolver and the invariant guardian. Our design introduces two main ideas that simplify the design and operation of network management solutions:

- **Three views of network state (observed, proposed, target):** In order to prevent conflicts and invariant violations, management solutions cannot change the state of the network directly. Instead, each solution applies its own logic to the network's *observed* state to generate a *proposed* state that may change one or more state variables of the network. Statesman merges all proposed states into one *target* state. In the merging process, it examines all proposed states to resolve conflicts and ensures that the target state satisfies an extensible set of network-wide invariants. Our design is inspired by version control systems like `Git`. Each solution corresponds to a different `Git` user and (i) the observed state corresponds to the code each user "pulls", (ii) the proposed state corresponds to the code the user wants to "push", and (iii) the target state corresponds to the merged code that is ultimately stored back in the shared repository.
- **Dependency model of state variables:** Prior works on abstracting network state model the state as independent variable-value pairs [90, 91]. However, this model does not contain enough semantic knowledge about how various state variables relate to each other, which hinders detecting conflicts and invariant violations. For example, a tunnel cannot carry traffic if the path includes an administratively down device. To ensure safe merging of proposed states, Statesman uses a *dependency model* to capture the domain-specific dependencies among state variables.

Statesman has been deployed and operational in ten datacenters of Microsoft Azure cloud service since October 2013. It currently manages over 1.5 million state variables from links and devices across the globe. We have also deployed two management solutions—device firmware upgrade and link failure mitigation, while a third one—inter-datacenter TE—is undergoing pre-deployment testing. The diverse functionalities of these solutions showcase how Statesman can safely support multiple management solutions, without them hurting each other or the network. We also show that these benefits come with reasonably low overhead. For instance, the la-

tency for conflict resolution and invariant checking is under 10 seconds even in the largest datacenter network with 394K state variables. We believe that our experience with Statesman can inform the design of future management systems for cloud providers.

2.2 Network State Abstraction

In this section, we provide more details on the abstraction of network state underlying Statesman, and how management solutions use the abstraction.

2.2.1 Three Views of Network State

Although management solutions have different functionalities, they typically follow a control loop of reading some aspects of the network state, running some computation on the state, and accordingly changing the network. One could imagine that each management solution reads and writes states to the network devices directly.

However, direct interaction between the devices and the management solutions is undesirable for two reasons. First, it cannot ensure that individual solution or their collective actions will not violate network-wide invariants. Second, reliably reading and writing network state is difficult because of response-time variances and device failures. When a command to a device takes a long time to execute, the management solution has to decide when to retry, how many times, and when to give up. When a command fails, the management solution has to parse the error code and decide how to react.

Given the issues above, Statesman abstracts the network state as multiple variable-value pairs. Furthermore, it maintains three different types of views of network state. Two of these are *observed state* (OS) and *target state* (TS). The OS is (a latest view of) the actual state of the network, which Statesman keeps up-to-date. Management

solutions read the OS to learn about current network state. The TS is the desired state of the network, and Statesman is responsible for updating the network to match the TS. Any success or failure of updating the network towards the TS will be (eventually) reflected into the OS, from where the management solutions will learn about the prevailing network conditions.

The OS and TS views are not sufficient for resolving conflicts and enforcing safety invariants. If management solutions directly write to the TS, the examples in Figure 2.1 and 2.2 can still happen. We thus introduce the third type of view called *proposed state* (PS) that captures the state desired by management solutions. Each solution writes its own PS.

Statesman examines the various PSes and detects conflicts among them and with the TS. It also validates them against a set of network-wide invariants. The invariants capture the basic safety and performance requirements for the network (e.g., the network should be physically connected and each pair of server pods should be able to survive a single-device failure). The invariants are independent of which management solutions are running. Only non-conflicting and invariant-compliant PSes are accepted and merged into the TS.

2.2.2 Dependency Model of State Variables

Management solutions read and write different state variables of the network, e.g., hardware power, device configuration, traffic routing, and multi-device tunneling. Statesman thus provides the state variables at multiple levels of granularity for the needs of management solutions (more details in §2.4 with examples in Table 2.2).

However the state variables are not independent. The "writability" of one state variable can depend on the values of other state variables. For example, when a link interface is configured to use the traditional control-plane protocol (e.g., BGP or OSPF), OpenFlow rules cannot be installed on that interface. In another example,

when the firmware of a device is being upgraded, its configuration cannot be changed and tunnels cannot be established through it. Thus, when proposing new values of state variables, conflicts can arise because a state variable in one solution's PS may become unchangeable due to some *dependent* state variables in another solution's PS.

Requiring management solutions to understand the complex cross-variable dependency will go against our goal of running them in a loosely coupled manner. For instance, it will be difficult for a TE solution to have to consider how one specific device configuration affects its operation. Therefore, Statesman does not treat the network state as a collection of independent variables but includes a model of dependencies among the state variables. These dependencies are used when checking for conflicts and invariant violations. Based on the dependency model, Statesman also exposes the "controllability" of each state variable as an additional logical variable. Thus, a management solution can read just the state variables of interest and their controllability variables to decide whether it is able to propose new values for those variables of interest. For example, a TE solution can read the "path" variable (i.e., a tunnel through multiple devices) and determine whether it is currently controllable. This is computed by Statesman based on various hardware and routing configurations of the devices along the path; the TE solution does not need to reason about the related hardware and routing configurations itself.

2.2.3 Application Workflow

In the observed-proposed-target pipeline, the workflow of management solutions is simple. Each solution reads the OS, runs its computational logic, and writes a newly generated PS. Statesman generates the new TS after resolving conflicts and invariant violations in the PSes and merging the accepted ones.

In this model, some proposals may be rejected. Handling this rejection does not impose extra overhead on management solutions; even if the interaction with network

devices was not mediated by Statesman, management solutions have to be prepared to be unable to update the network to the desired state (e.g., due to failures during the update process). When Statesman rejects a proposal, the proposer gets detailed feedback on the reason for rejection (§2.5), at which point it can propose a new PS in an informed manner.

Per our desire for loose coupling, management solutions make proposals independently. It is thus possible that PSes of two solutions frequently conflict or we do not compute a target state that satisfies all management solutions (even though such a state may exist). In our operational experience with Statesman, such cases are not common. Thus, we believe that the simplicity of loose coupling outweighs the complexity of tight coupling. In cases where coordination among two management solutions is highly beneficial, it may be done out of band such that the solutions make proposals after consulting each other. In this way, most management solutions and the Statesman system as a whole stay simple, and the complexity cost of coordination is borne only in parties that benefit the most from the coordination.

2.3 System Overview

Figure 2.3 shows the architecture of Statesman. It has four components: storage service, checker, monitor, and updater. We outline the role of each below, and the following sections present more details.

Storage service is at the center of the system. It persistently stores the state variables of OS, PS, and TS and offers a highly-available, read-write interface for other components and management solutions. It also handles all data availability and consistency issues, which allows all other components to be completely stateless—in each round of their operations, they just read the latest values of the needed

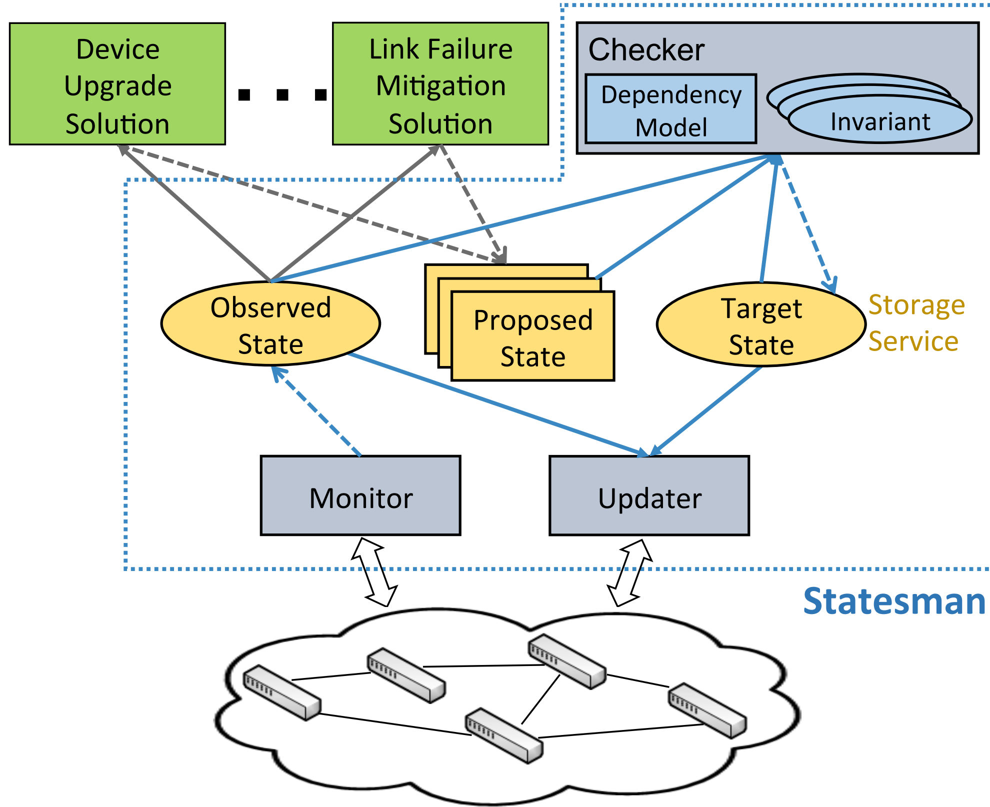

Figure 2.3: Statesman Architecture Overview

state variables. This stateless mode of operation simplifies the design of the other components.

The checker, monitor, and updater independently interact with the storage service, and the latter two also interact with the network devices. Table 2.1 summarizes the input and output of each component.

Checker plays a pivotal role of generating the TS. After reading the OS, PSes, and TS from the storage service, the checker first examines whether the PSes are still applicable with respect to the latest OS (e.g., the proposed change may have already been made or cannot be made at all due to a failure). It then detects conflicts among PSes with the state dependency model and resolves them with one of two configurable mechanisms: last-writer-wins or priority-based locking. After merging the valid and non-conflicting PSes into the TS, the checker examines the TS for the

Component	Input	Output
Monitor	Device/link statistics	OS
Checker	OS PSes TS	TS
Updater	OS TS	Device update commands

Table 2.1: Input and Output of Each Component in Statesman

operator-specified safety invariants. It writes the TS to the storage service only if the TS complies with the invariants. It also writes the acceptance or rejection results of the PSes to the storage service, so management solutions can learn about the outcomes and react accordingly.

Monitor periodically collects the statistics from the devices and links, transforms them into values of OS variables, and writes the new values to the storage service. In addition to simplifying other components and management solutions to learn about current network state, the monitor also shields them from the heterogeneity among devices. Based on the device vendor and the supported technologies, it uses the corresponding protocol (e.g., SNMP or OpenFlow) to collect the network statistics, and it translates protocol-specific data to protocol-agnostic state variables. Other components and management solutions use these abstract variables without worrying about the specifics of the underlying infrastructure.

Updater reads the OS and TS and translates their difference into update commands that are then sent to the devices. The updater is memoryless—it applies the latest difference between the OS and TS without regard to what happened in the past. Like the monitor, the updater handles how to update heterogeneous devices with a command template pool, and allows other components and management solutions to work with device- and protocol-agnostic state variables.

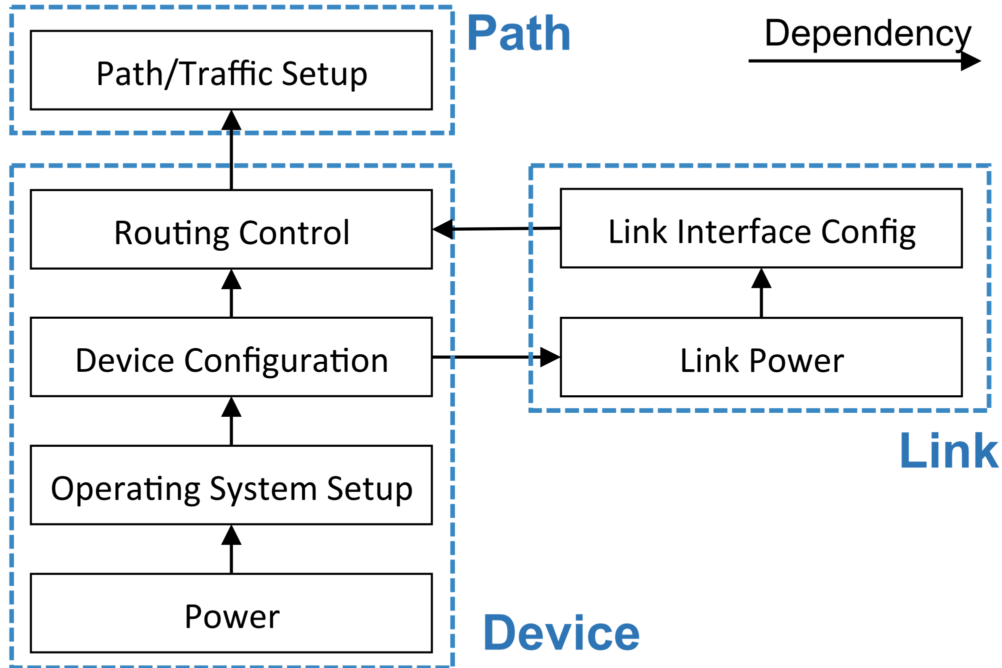

Figure 2.4: Network State Dependency Model

2.4 Managing Network State

We now describe the various aspects of Statesman in more details, starting with the network-state data model. We use the examples in Table 2.2 to illustrate how we build the state dependency model, and how to use and extend the model.

2.4.1 The State Dependency Model

Managing a datacenter network involves multiple levels of control. To perform the final function of carrying traffic, the network needs to be properly powered and configured. Statesman aims to support operations in the complete process of bringing up a large datacenter network from scratch to normal operations. In order to capture the relationship among the state variables at different levels of the management process, we use the state dependency model of Figure 2.4. We use the process of bootstrapping a datacenter network as an example to explain this model.

At the bottom of the dependency model is the power state of network devices. With the power cable properly plugged in and electricity properly distributed to the devices, we then need to control which operating system (i.e., firmware) runs. Running a functioning firmware on a device is the prerequisite for managing device configuration, e.g., use device vendor's API to configure the management interface, boot up compatible OpenFlow agent, etc.

With device configuration states ready, we are able to control the link interfaces on the device now. The fundamental state variable of a link is its being up or down. The configuration of a link interface follows when the link is ready to be up. There are various link-interface configuration states, such as IP assignment, VLAN setup, ECMP-group assignment, etc. Consider an example of control plane setup where a link interface can be configured to use the OpenFlow protocol or traditional protocols like BGP. For the chosen option, we need to set it up: either an OpenFlow agent needs to boot and take control of the link, or the BGP session needs to start with proper policies. These control plane states of the link determine whether and how the device's routing can be controlled.

We can manage the routing states of the devices when all the dependent states are correct. We represent the routing state in a data structure of the flow-link pairs, which is agnostic to the supported routing protocols. For example, the routing states can map to the routing rules in OpenFlow or the prefix-route announcement or withdrawal in BGP. When management solutions change the value of the routing state variable, Statesman (specifically the updater) automatically translates the value to appropriate control-plane commands.

One level higher is the *path* state which controls tunnels through multiple devices. Creating a tunnel and assigning traffic along the path depend on all devices on the path having their routing states ready to manage. Again, Statesman is responsible

Entity	Level in dependency	Example state variables	Permission
Path	Path/traffic setup	Devices on path MPLS or VLAN configuration	ReadWrite ReadWrite
Link	Link interface configuration	IP assignment Control plane setup	ReadWrite ReadWrite
	Link power	Interface admin status Interface oper status	ReadWrite ReadOnly
	N/A (counters)	Traffic load Packet drop rate	ReadOnly ReadOnly
Device	Routing control	Flow-link routing rules Link weight allocation	ReadWrite ReadWrite
	Device configuration	Management interface setup OpenFlow agent status	ReadWrite ReadWrite
	Operating system setup	Firmware version Boot image	ReadWrite ReadWrite
	Power	Admin power status Power unit reachability	ReadWrite ReadOnly
	N/A (counters)	CPU utilization Memory utilization	ReadOnly ReadOnly

Table 2.2: Examples of Network State Variables

for translating the path's states into the routing states of all devices on the path, and the management solution only needs to read or write the path states.

2.4.2 Using and Extending the Dependency Model

For simplicity, management solutions should not be required to understand all the state variables and their complex dependencies. They should be able to simply work with the subset of variables that they need to manage based on their goals. For instance, a firmware-upgrade solution should be able to focus on only the *Device-FirmwareVersion* variable. However, this variable depends on lower-level variables, such as device power state whose impact cannot be completely ignored; firmware cannot be upgraded if the device is down.

We find that it suffices to represent the impact of these dependencies by using a logical variable that we call *controllability* and expose it to management solutions. This boolean-valued variable denotes whether the parent state variable is currently controllable, and its value is computed by Statesman based on lower-level dependencies. For instance, *DeviceFirmwareVersion* is controllable only if the values of variables such as device power and admin states are appropriate. Now the firmware-upgrade solution can simply work with *DeviceFirmwareVersion* and *DeviceFirmwareVersion-IsControllable* variables to finish its job.

To give another concrete example, the variable *DeviceConfigIsControllable* tells whether management solutions can change various state variables of device configuration, such as the management interface setup. The value of *DeviceConfigIsControllable* is calculated based on whether the device is powered up, whether the device can be reachable via SSH/Telnet from the management network (indicating the firmware is functioning), and whether the device is healthy according to the health criterion (e.g., CPU/memory utilizations are not continuously 100% for certain amount of time). Similarly, links have *LinkAdminPowerIsControllable* calculated with the *DeviceConfigIsControllable* of the two devices on the link's ends.

Exposing just the controllability variables makes the dependency model extensible. The functions calculating the controllability are implemented in the storage service of Statesman. When a new state variable is added to Statesman, we just need to place it in the dependency model, i.e., find what state variables will be dependent on the new one. Then we modify the controllability functions of the corresponding state variables to consider the impact of the new variable. For management solutions that are not interested in the new variable, no modifications are necessary.

2.5 Checking Network State

The checker plays a pivotal role in Statesman. It needs to resolve conflicts among management solutions and enforce network-wide invariants. Its design also faces scalability challenges when handling large networks. We first explain the state checking process and then describe techniques for boosting scalability of the checker. Figure 2.5 outlines the checker's operation.

2.5.1 Resolving Conflicts

Multiple types of conflicts can occur in the network state due to the dynamic nature of datacenter networks and uncoordinated management solutions:

- **TS–OS:** The TS can conflict with the latest OS when changes in the network render some state variables uncontrollable, although they have new values in the TS. For instance, if the OpenFlow agent on a device crashes, which is reflected as *DeviceAgentBootStatus=Down* in the OS, any routing changes in the TS cannot be applied.
- **PS–OS:** When the checker examines a PS, the OS may have changed from the time that the PS was generated, and some variables in the PS may not be controllable anymore. For example, a PS could contain the new value of a *LinkEndAddress* variable, but when the checker reads the PS, the link may have been shut down for maintenance.
- **PS–TS:** The TS is essentially the accumulation of all accepted PSes in the past. A PS can conflict with the TS due to an accepted PS from another management solution. For example, assume that a firmware-upgrade solution wants to upgrade a device and proposes to bring it down; its PS is accepted and merged into the TS. Now, when a TE solution proposes to change the device's routing state, it conflicts with the TS even though the device is online (i.e., no conflict with the OS).

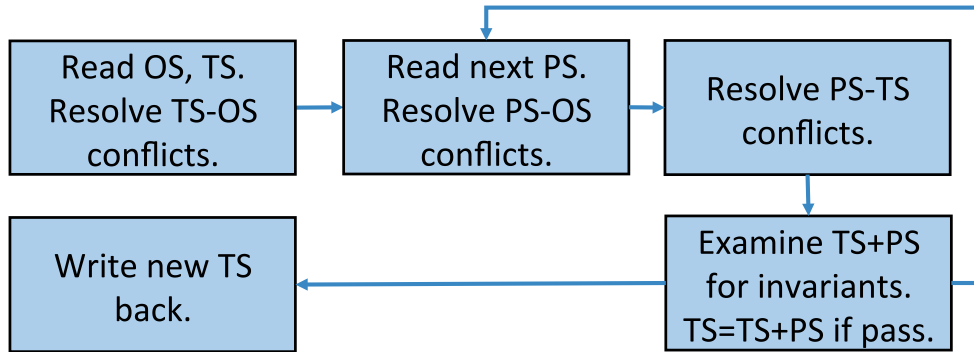

Figure 2.5: Flow of The Checker's Operation

The first two types of conflicts are because of the changing OS, which makes some variables in the PS or TS uncontrollable. To detect these conflicts, the checker reads the controllability values from the OS, which are calculated by the storage service based on the dependency model when relevant variable values are updated. It then locates the uncontrollable variables in the PS or TS. To resolve TS–OS conflicts, we introduce a flag called *SkipUpdate* for each variable in the TS. If set, the updater will bypass the network update of the state variable, thus temporarily ignoring the target value to resolve the conflict. The checker will clear the flag once the state variable is controllable again.

For uncontrollable state variables in a PS, the checker removes them from the PS, i.e., rejecting the part of PS that is inapplicable on the current network. The choice of partial rejection is a tradeoff between the progress of management solutions and the potential side-effects of accepting a partial PS. An asynchrony is normal between the OS views of the management solution and the checker. By rejecting the whole PS due to a small fraction of conflicting variables, Statesman will be too conservative and will hinder the progress of management solution. We thus allow partial rejection. We have not yet observed any negative consequences in our deployment from this choice. In the future, we will extend the PS data structure such that solutions can group variables. When a variable in a group is uncontrollable, the entire group is rejected, but other groups can still be accepted.

For PS–TS conflicts, which are caused by the conflicting proposals, Statesman supports an extensible set of conflict resolution mechanisms. It currently offers two mechanisms. The basic one is last-writer-wins, in which the checker favors the value of state variable from the newer PS. The more advanced mechanism is priority-based locking. Statesman provides two levels of priorities of locks for each device and link. Management solutions can acquire a low-priority or a high-priority lock before proposing a PS. In the presence of a lock, management solutions other than the lock holder cannot change the state variables of the device or link. However, the high-priority lock can override the low-priority one. The choice of the conflict resolution mechanism is not system-wide and can be configured at the level of individual devices and links.

Although simple, these two conflict resolution strategies have proved sufficient for the management solutions that we have built and deployed thus far. In fact, we find from our traces that last-writer-wins is good enough most of the time since it is rare for two management solutions to compete for the same state variable head-to-head. For our intra-datacenter infrastructure, we configure the checker with the last-writer-wins resolution; for our inter-datacenter network, we have enabled the priority-based locking. If needed, additional resolution strategies (e.g., based on access control) can be easily added to Statesman.

2.5.2 Choosing and Checking Invariants

Network-wide invariants are intended to ensure the infrastructure's operational stability in the face of development bugs or undesired effects of collective actions of the management solutions. They should capture minimum safety and performance requirements, independent of which management solutions are currently running.

Choosing invariants: The choice must balance two criterion. Firstly, the invariant should suffice to safeguard the basic operations of the network. As long as it

is not violated, most services using the network would continue to function normally. Second, the invariant should not be so stringent that it interferes with the management solutions. For instance, an invariant that require all devices up is likely too strict and interferes with a firmware-upgrade solution.

Following the criterion above, we currently use two invariants in the checker. Both are solution-agnostic and relate to the topological properties of the network. The first invariant considers network connectivity. It requires that every pair of ToR devices in the same datacenter are (topologically) connected and that every ToR is connected to the border routers of its datacenter network (for WAN connectivity). Here, we ignore the routing configurations on the devices (which could result in two ToRs not being able to communicate) because management solutions can intentionally partition a network at the routing level for multi-tenant isolation.

The second invariant considers network capacity. We define the capacity between two ToRs in the same datacenter as maximum possible traffic flow between them based on the network topology. The invariant is that the capacity between $p\%$ of ToR pairs should be at least $t\%$ of their baseline, when all links are functional. The values of p and t should be based on level of capacity redundancy in the network, tolerance of the hosted services to reduced capacity, and implications of blocking a management solution that violates the invariant. We currently use $p = 99$ and $t = 50$, i.e., 99% of the ToR pairs must have at least 50% of the baseline capacity.

Although Statesman currently maintains only two invariants, the set of invariants is extensible. The invariant checking is implemented as a boolean function over a graph data structure that has the network topology and all state variables. It is straightforward to add more invariants by implementing new functions with the same interface. For example, some networks may add an invariant that requires the absence of loops and black holes, which can be checked using the routing states of the devices.

Checking invariants: When checking whether the TS obeys the invariants, the checker first creates a base network graph using variable values from the OS. Then, it translates the difference between a variable's observed and target values into an operation on the network graph, e.g., bringing a device offline, changing the routing state to the target value, etc. Finally, the invariant checking functions are run with the new network graph.

The invariant checking is invoked in two places. The first is when the checker reads the TS from the storage service. The TS was compliant when written but can be in violation when read due to changes in the network (i.e., the TS–OS conflict). While running the invariant checking as described above, the checker clears or sets the *SkipUpdate* flags in the TS respective of its compliance status.

The second place is when the checker tries to merge a PS into the TS after conflict resolution. The checker analyzes TS+PS, as if the PS was merged into the TS. If TS+PS passes the check, the PS is accepted and merged into the TS. Otherwise, the PS is rejected.

The acceptance or rejection (and reasons) for each PS is recorded by the checker into the storage service. We categorize the rejection reasons into three groups: state variable became uncontrollable; invariant was violated; and TS was unreachable. The reasons are encoded as machine-readable status code as values of a special state variable called *ProposalStatus* for each PS. Management solutions use the same interface as for reading the OS to read their PSes' statuses. They can then react appropriately (e.g., generate a new PS that resolves the conflicts with the latest OS).

2.5.3 Partitioning by Impact Group

The checker needs to read all the state variables in the OS and TS to examine conflicts and invariants every round of operation. The sheer number of state variables in the datacenters and the WAN poses a scalability threat.

To help scale, we observe that the impact of state changes for a given device or link is limited. For instance, changes to an aggregation router in one datacenter do not affect the connectivity and capacity of ToRs in another datacenter. Similarly, changes to border routers in a datacenter do not impact the capacity and connectivity within the datacenter, though they do impact other datacenters' WAN reachability.

Based on this observation, we partition the checker's responsibility into multiple impact groups. We set up one impact group per datacenter and one additional group with border routers of all datacenters and the WAN links. These groups are independent with respect to the state checking process. In our deployment of Statesman, we run one instance of checker per impact group, which has enabled the state checking to scale (§2.8).

2.6 System Design and Implementation

We now describe in more details the design and implementation of Statesman. We start with the storage service, followed by the updater and monitor; we skip the checker as it was covered in details in §2.5. Figure 2.6 provides a closer look at various components in a Statesman deployment. Our implementation of Statesman has roughly 50 thousand lines of C# and C++ code, plus a number of internal libraries. At its core, it is a highly-available RESTful web service with persistent storage. Below, we also describe Statesman's read-write APIs.

2.6.1 Globally Available and Distributed Storage Service

The storage service needs to persistently and consistently store the network states. We thus use a Paxos-based distributed file system. However, two challenges prevent us from using a single big Paxos ring to maintain all the states for all our datacenters. The first is datacenter reachability. Due to WAN failures, one datacenter may lose

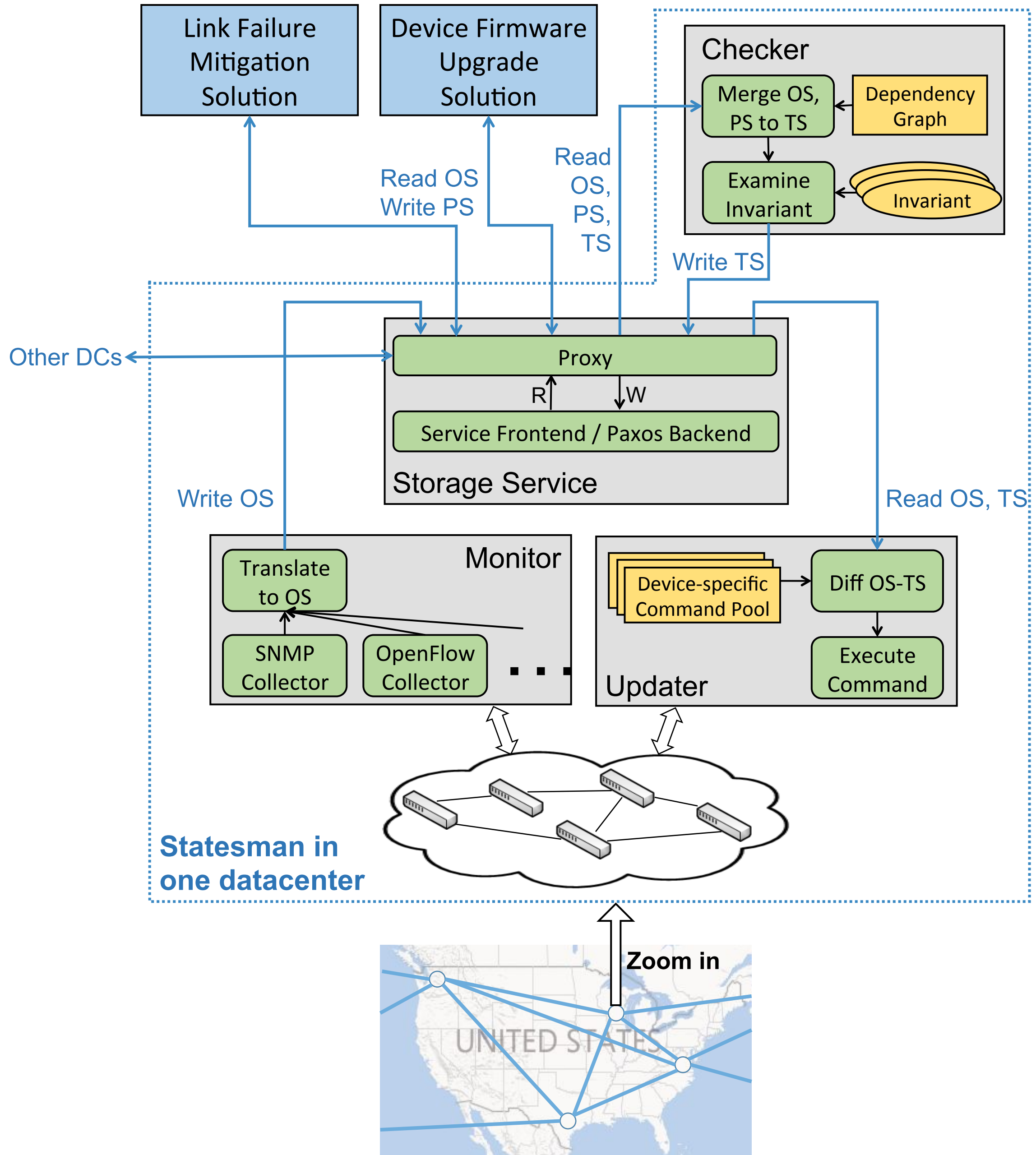

Figure 2.6: Statesman System Design

connectivity to all others, or two datacenters may not be able to reach each other. To protect Statesman from such failures, the storage instances need to be located in multiple datacenters.

A second challenge stems from the volume of state data. In datacenters, there are hundreds of thousands of devices and links, each with dozens of state variables.

It results in millions of state variables (§2.8). Manipulating all variables in a single Paxos ring would impose a heavy message-exchange load on the file system to reach consensus over the data value. This impact worsens if the exchange happens over the WAN (as storage instances are located in multiple datacenters for reliability). WAN latencies will hurt the scalability and performance of Statesman.

Therefore, we break a big Paxos ring into independent smaller rings for each datacenter. One Paxos ring of storage instances is located in each datacenter, and it only stores the state data of the devices and links in that datacenter. In this way, Statesman reduces the scale of state storage to individual datacenters, and it lowers the impact of Paxos consensus by limiting message exchanges inside the datacenter.

Although the underlying storage is partitioned and distributed, we still want to provide a uniform and highly available interface for the management solutions and other Statesman components. These users of the storage service should not be required to understand where and how various state variables are stored.

We thus deploy a globally available proxy layer that provides uniform access to the network states. Users read or write network states of any datacenter from any proxy without knowing the exact locations of the storage service. Inside the proxy, we maintain an in-memory hash table of the device and link names to corresponding datacenters for distributing the requests across the storage-service instances. The proxy instances sit behind a load balancer, which enables high availability and flexible capacity.

2.6.2 Stateless Update on Heterogeneous Devices

Network update is a challenging problem itself [81, 93, 127]. In the context of managing a large network, it becomes even more challenging for three reasons. First, the update process is device- and protocol-specific. Although OpenFlow provides a standard interface for changing the forwarding behaviors of devices, there is no stan-

dard interface today for management-related tasks such as changing the device power, firmware, or interface configuration. Second, because of scale and dynamism, network failures during updates are inevitable. Finally, the device's response can be slow and dominate the solution's control loop. Two aspects of the design of the Statesman updater help to address these challenges.

Command template pool for heterogeneous devices: The changes from management solutions (i.e., PSes) are device-agnostic network states. The updater translates the difference between a state variable's OS and TS values into device-specific commands. This translation is done using a pool of command templates that contains templates for each update action on each device model with supported control-plane protocol (e.g., OpenFlow or vendor-specific API). When the updater carries out an update action, it looks up the template from the pool based on the desired action and device details.

For instance, suppose we want to change the value of a device's *DeviceRoutingState*. If the device is an OpenFlow-enabled model, the updater looks up this model's OpenFlow command template to translate the routing state change into the insertion or deletion of OpenFlow rules, and issues rule update commands to the OpenFlow agent on the device. Alternatively, if the device is running a traditional routing protocol like BGP, the updater looks up the BGP command template to translate the routing state change into the BGP-route announcement or withdrawal.

Stateless and automatic failure handling: With all network states persistently stored by the storage service, the updater can be stateless and simply read the new values of OS and TS every round. This mode of operation makes the updater robust to failures in the network or in the updater itself. It can handle failures with an implicit and automatic retry. When any failure happens in one run of update, the state changes resulted by the failure reflect as a changed OS in the storage service. In the next run, the updater picks up the new OS which already includes the failure's

impact, and it calculates new commands based on the new OS-TS difference. In this manner, the updater always brings the latest OS towards the TS, no matter what failures have happened in the process.

Being stateless also means that we can run as many updater instances as needed to scale, as long as we are able to coherently partition the work among them. In our current deployment, we run one instance per state variable per device model. In this way, each updater instance is specialized for one task.

2.6.3 Network Monitors

The monitors collect the network states with various protocols from the devices, including SNMP and OpenFlow. The monitors then translate the protocol-specific data into the value of corresponding state variables, and write them into the storage service as the OS. We split the monitoring responsibility across many monitor instances, so each instance covers roughly 1,000 devices.

Currently the monitors run periodically to collect all devices' power states, firmware versions, configurations, and various counters (and routing states for a subset of devices). For links, our monitors cover the link power, configuration, and counters like the packet drop rate and the traffic load.

2.6.4 Read-Write APIs

The storage service is implemented as a HTTP web service with RESTful APIs. The management solutions, monitors, updaters, and checkers use the APIs to read or write *NetworkState* objects from the storage service. A *NetworkState* object consists of the entity name (i.e., the device, link, or path name), the state variable name, the variable value, and the last-update timestamp. The read-write APIs of Statesman are shown in Table 2.3.

GET	NetworkState/Read?Datacenter={dc}&Pool={p} &Freshness={c}&Entity={e}&Attribute={a}
POST	NetworkState/Write?Pool={p} (Body is list of NetworkState objects in JSON)

(a) HTTP Request

Datacenter	dc	Datacenter name
Pool	p	OS, PS, or TS
Freshness	c	Up-to-date or bounded-stale
Entity	e	Entity name (i.e., device, link, or path)
Attribute	a	State variable name

(b) Parameters

Table 2.3: Read-Write APIs of Statesman

There is a *freshness* parameter in the read API because Statesman offers different freshness modes for reading the network states. The up-to-date mode is for management solutions who are strict with the state staleness. For instance, the link-failure-mitigation solution needs to detect link failures as soon as possible when the failures happen. Statesman also offers 5-minute bounded-staleness mode by reading from caches [124]. Many management solutions do not need the most up-to-date network states and can safely tolerate some staleness in state data. For instance, the firmware-upgrade solution needs to upgrade the devices within hours; it does not matter if the value of *DeviceFirmwareVersion* is slightly stale. By allowing such management solutions to read from caches, we boost the read throughput of Statesman. At the same time, solutions that cannot tolerate staleness can use the up-to-date freshness mode.

2.7 Operational Experiences

In this section, we present our experiences of running Statesman in Microsoft Azure. Statesman is now deployed in ten datacenters. We have built two production and one pilot-stage management solutions on top of Statesman. We first describe the

deployment, and then use three representative scenarios to illustrate how Statesman facilitates the operations of management solutions.

2.7.1 Deployment in Microsoft Azure

Statesman currently manages ten geographically distributed datacenters of Microsoft Azure, covering all the devices and links in those datacenters and the WAN connecting the datacenters. The three management solutions that we have built leverage Statesman to manage different aspects (e.g., devices, links, and traffic flows) of our datacenter network.

- **Device upgrade:** When a new version of firmware is released by a device vendor, this solution automatically schedules all the devices from the vendor to upgrade by proposing a new value of *DeviceFirmwareVersion*. In the upgrade process, the checker of Statesman ensures that the datacenter network continues to meet the network-wide invariants.
- **Failure mitigation:** This solution periodically reads the Frame-Check-Sequence (FCS) error rates on all links. When detecting persistently high FCS error rates on certain links, it changes the *LinkAdminPower* state to shut down those faulty links to mitigate the impact of the failures [132]. The solution also initiates an out-of-band repair process for those links, e.g., by creating a repair ticket for the on-site team.
- **Inter-datacenter traffic engineering (TE):** As described in SWAN [81], Statesman collects the bandwidth demands from the *bandwidth brokers* sitting with the hosted services. In addition, the monitor of Statesman collects all the forwarding states, such as tunnel status and flow matching rules. Given this information, this solution computes and proposes new forwarding states, which are then pushed to all the relevant devices by the Statesman updater.

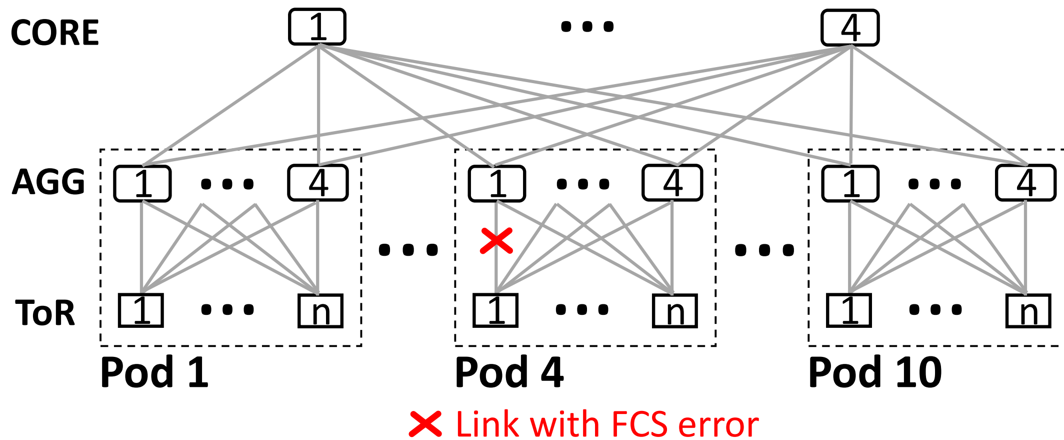

Figure 2.7: Network Topology for The Scenario in §2.7.2

The first two solutions have been fully tested and deployed, while the third one is in pilot stage and undergoing testing.

We state two issues of how management solutions interact with Statesman. First, they should understand that it takes time for Statesman to read and write a large amount of network states in large networks. Thus, their control loops should operate at the time scale of minutes, not seconds. Second, the management solutions should understand that their PSes may be rejected due to failures, conflicts, or invariant violations. Thus, they need to run iteratively to adapt to the latest OS and the acceptance or rejection of their previous PSes.

2.7.2 Maintaining Network-wide Invariants

We use a production trace to illustrate how Statesman helps two management solutions (device-upgrade and failure-mitigation) safely coexist in intra-datacenter networks, while Statesman maintains the capacity invariant—99% of the ToR pairs in the datacenter should have at least 50% of their baseline capacity.

Figure 2.7 shows the topology for the scenario. It is a portion of one datacenter with 10 pods, where each pod has 4 Agg routers and a number of ToRs. The device-upgrade solution wants to upgrade all the 40 Agg routers in a short amount of time.

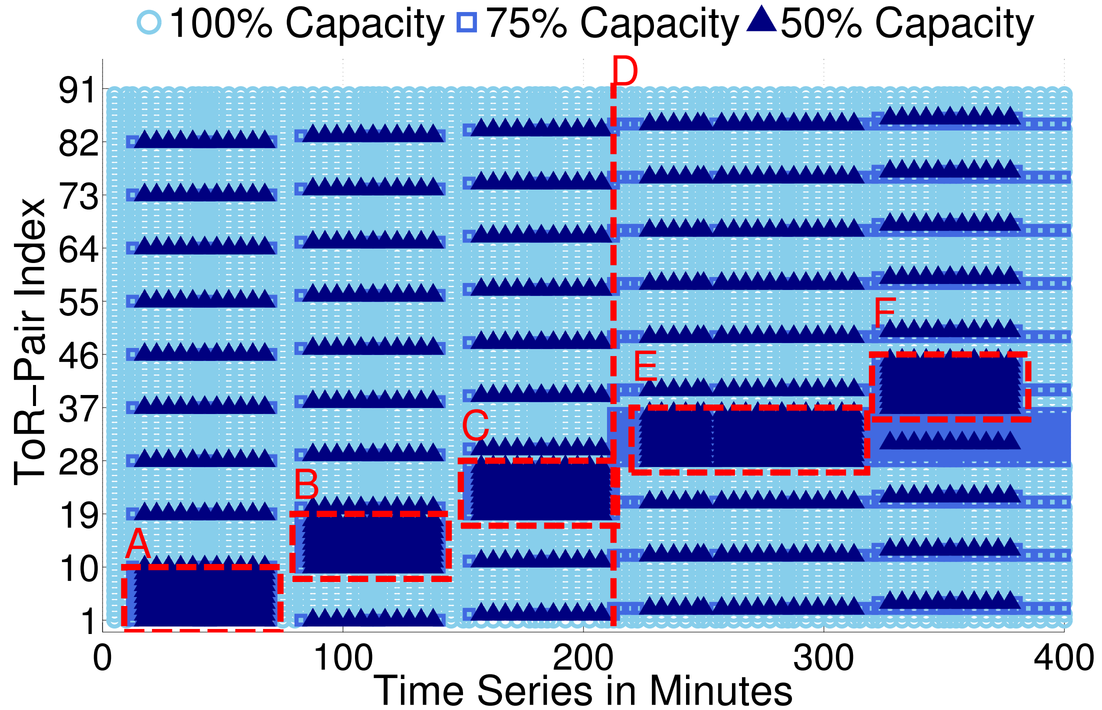

Figure 2.8: Device Upgrade Process while Statesman Maintains Safety

Specifically, it will upgrade the pods one by one. Within each pod, it will attempt to upgrade multiple Agg routers in parallel by continuing to write a PS for one Agg upgrade until it gets rejected by Statesman.

In Figure 2.8, we pick one ToR from each pod, and organize the 10 ToRs from the 10 pods into 90 ToR pairs (10×9, excluding the originating ToR itself). On the Y-axis, we put the 9 ToR pairs originating from the same ToR/pod together. Essentially, Figure 2.8 shows how the network capacity between each ToR pair changes over time. Boxes A, B, and C illustrate how the network capacity temporarily drops when the device-upgrade solution upgrades the Agg routers in Pod 1, 2, and 3 sequentially. To meet the 50%-capacity invariant, the device-upgrade solution can simultaneously upgrade up to 2 out of 4 Agg routers, leaving at least 2 Agg routers alive for traffic.

During the upgrade, the failure-mitigation solution discovers persistently high FCS error rate on link ToR_1-Agg_1 in Pod_4. As a result, it shuts down this link at time D. Since one ToR-Agg link is down, the capacity of all Pod_4-related ToR pairs

drops to 75%, which originate from Pod_4 (index # 28–36) or end at Pod_4 (index # 3, 12, 21, 40, 49, 58, 67, 76, & 85). When the device-upgrade solution starts to work on Pod_4, Statesman can only allow it to upgrade one Agg at a time to maintain the 50%-capacity invariant. Thus, as shown in box E, the device-upgrade solution automatically slows down when upgrading Pod_4. Its actual upgrade steps are Agg_1-Agg_2-together, then Agg_3, and finally Agg_4. Note that Agg_1 and Agg_2 can be upgraded in parallel, because link ToR_1-Agg_1 is already down and hence upgrading Agg_1 does not further reduce the ToR-pair capacity. The device-upgrade solution resumes normal speed when it upgrades Pod_5 in box F.

2.7.3 Resolving Conflicts of Management Solutions

The inter-datacenter TE solution is responsible for allocating inter-datacenter traffic along different WAN paths. Figure 2.9 shows the pilot WAN topology used in this experiment. This WAN inter-connects four datacenters in a full mesh, and each datacenter has two border routers. The device-upgrade solution is also running on the WAN.

One recurring scenario is that we need to upgrade all the border routers while inter-datacenter traffic is on. This can lead to a conflict between the two management solutions: the device-upgrade solution wants to reboot a router for firmware upgrade, while the TE solution wants the router to carry traffic. Without Statesman, the operators of the two management solutions have to manually coordinate, e.g., setting up a maintenance window. During this window, the operators must carefully watch the upgrade process for any unexpected events.

With Statesman, the whole process becomes much simpler. When there is no upgrade, the TE solution acquires the low-priority lock over each router, and changes the forwarding states as needed. When the device-upgrade solution wants to upgrade a router, it first acquires the high-priority lock over that router. Soon after, the TE

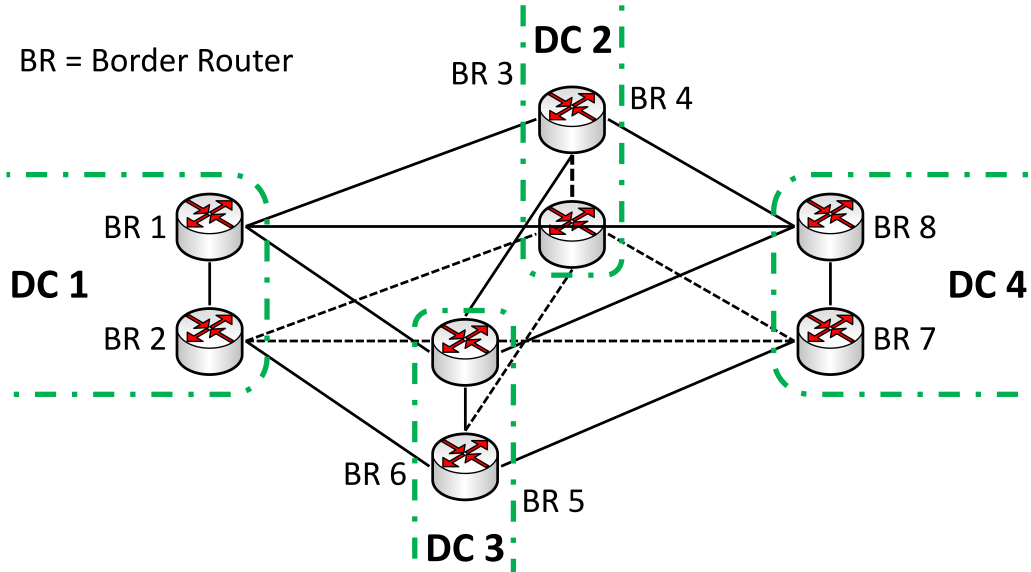

Figure 2.9: WAN Topology for The Scenario in §2.7.3

solution realizes that it cannot acquire a low-priority lock over the router, and it starts to shift traffic away from that router. Meanwhile, the device-upgrade solution keeps reading the traffic load of the locked router until the load drops to zero. At this moment, it kicks off the upgrade by writing a PS with a new value of *Device-FirmwareVersion*. Once the upgrade is done, the device-upgrade solution releases the high-priority lock of the router, and proceeds to the next candidate.

We collected the traffic load data during one upgrade event in off-peak hours. Since the load patterns of different routers are similar, we only illustrate the upgrade process of BorderRouter_1 (BR_1). Figure 2.10 shows the time series of the link load (note that both solutions run every 5 minutes). The Y-axis shows the 24 links (12 physical links $\times$ 2 directions) indexed by the originating router of each link. At time B, the TE solution fails to acquire the low-priority lock over BR_1, since the high-priority lock of BR_1 was acquired by the device-upgrade solution at time A. So the TE solution moves traffic away from BR_1. At time C, the load drops to zero on all the links originating from BR_1 (index # 1, 2, & 3) and ending at BR_1 (index # 7, 16, & 22). As expected, this increases the loads on the other links. After the device-upgrade

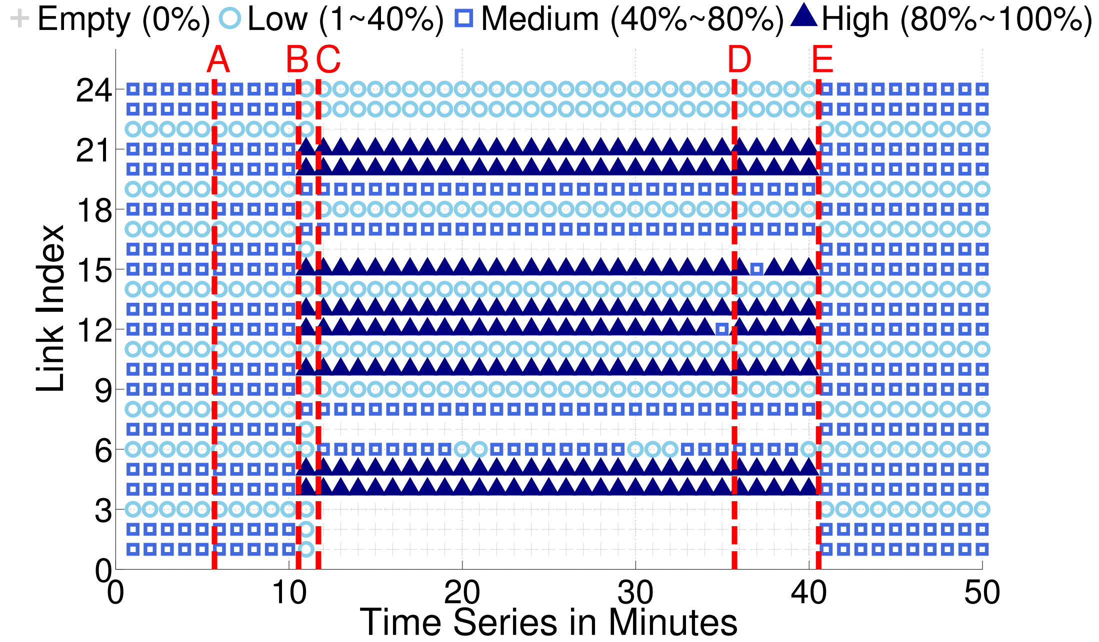

Figure 2.10: Conflict-Free TE and Device Upgrade with Statesman

solution finishes upgrading BR_1 and releases the high-priority lock at time D, the TE solution successfully acquires the low-priority lock again at time E, and then moves traffic back to BR_1.

In this example, neither management solutions needs to be aware of the other since conflict resolution and necessary coordination are automatically enabled by Statesman. We use locking as the conflict resolution strategy in the inter-datacenter case. So the TE solution can move tunnels away from devices being upgraded before the upgrade process starts, rather than after (which would be the case with the conflict resolution based on last-writer-wins). In the intra-datacenter case, we do not use tunnel-based TE, and neighboring routers can immediately reroute traffic when a device is brought down for upgrade without warning.

2.7.4 Handling Operational Failures

In Statesman, a management solution outputs a PS instead of directly interacting with the network devices. Once the PS is accepted into the TS, Statesman takes

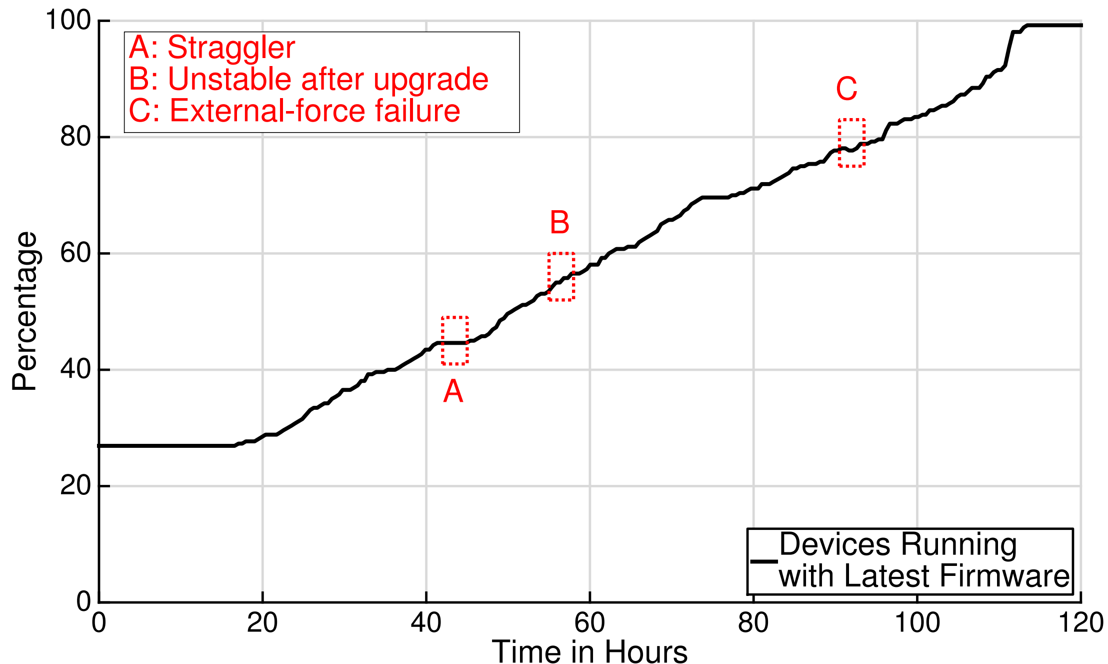

Figure 2.11: Time Series of Firmware Upgrade at Scale

the responsibility of (eventually) moving the network state to the TS. This simplifies failure handling in the management solution.

In this example, we show how the device-upgrade solution rolls out a new firmware to roughly 250 devices in several datacenters in two stages. In the first stage, 25% of the devices are upgraded to test the new firmware. Once we are confident with the new firmware, the remaining 75% of the devices are upgraded in the second stage. Figure 2.11 illustrates the upgrade process, where the percentage of devices with new firmware gradually grows to 100% in about 100 hours. Here, the device-upgrade solution runs conservatively and upgrades one device at a time.

During this process, Statesman automatically overcomes a variety of failures, which are highlighted in Figure 2.11. In box A, there is a straggling device which takes 4 hours to upgrade; note the flat line in the figure. During the 4 hours, Statesman repeatedly tries to upgrade until it succeeds. This straggling happens because the device has a full memory and cannot download the new firmware image. After some of the device memory is freed up, the upgrade proceeds. In box B, a few devices

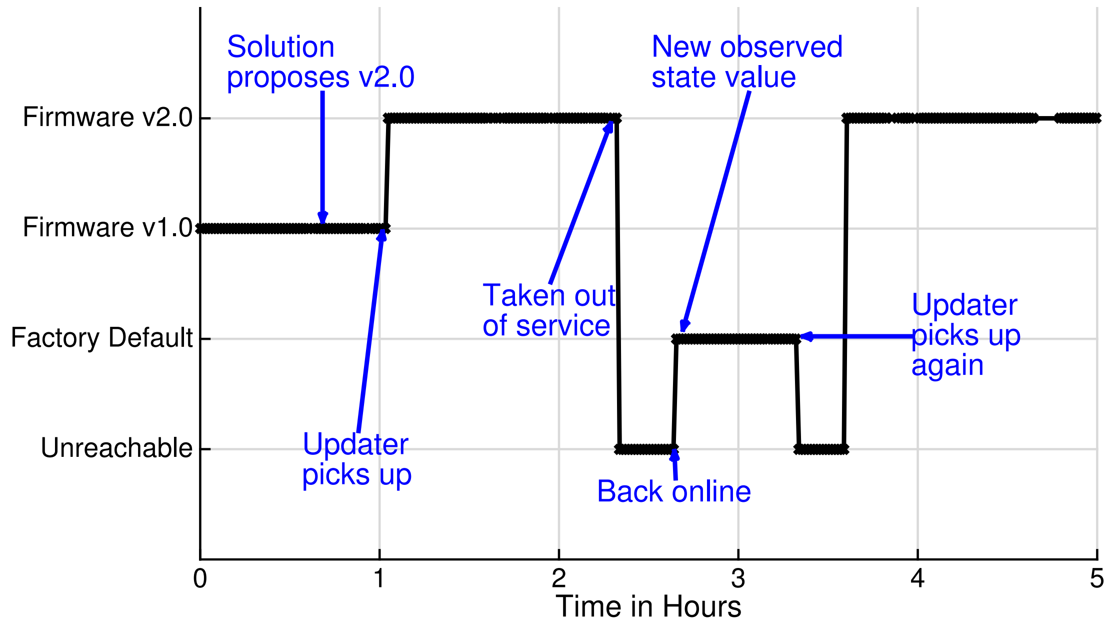

Figure 2.12: External-Force Failure in Device Upgrade

become unstable after the upgrade, e.g., frequently flipping between on and off. This appears as a stair-shape line in Figure 2.11. Statesman automatically retries until the devices are stable with the new firmware.

Box C shows a failure case which involves human interventions. After a device is upgraded, the operators take it out of service, and manually reset it to the factory-default firmware before returning it to production. This action causes the time-series line to slightly drop in Box C, and we zoom in the process in Figure 2.12. Once the device is back to production, Statesman finds that the observed *DeviceFirmwareVersion* of one device is the factory-default one. Since the firmware of the device has been set to a new value in the TS, Statesman automatically redoes the firmware upgrade without any involvement from the device-upgrade solution.

2.8 System Evaluation

In this section, we quantify the latency and coverage of Statesman as well as the performance of checking, reading and writing network states.

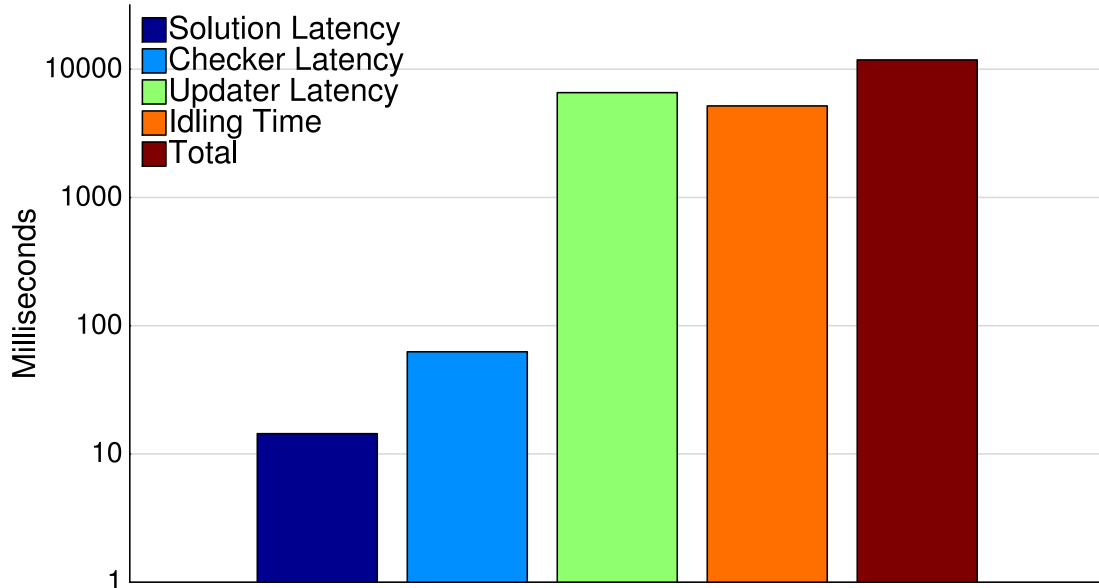

Figure 2.13: End-to-end Latency Breakdown

End-to-end latency: Figure 2.13 shows Statesman's end-to-end latency measured by running the failure-mitigation solution on a subset of our datacenter network. We manually introduce packet drops on a link, and let the failure-mitigation solution discover the problem and then shut down the link. The end-to-end latency is measured from when the solution reads the high packet drop rate of the link to when the faulty link is actually deactivated in the network. We break down the end-to-end latency into four portions:

- Solution latency: from when the solution reads the high packet drop rate to when it writes a PS for shutting down the faulty link.
- Checker latency: from when the checker reads the PS to when it finishes checking and writes a new TS.
- Updater latency: from when the updater reads the new TS to when it finishes shutting down the faulty link.
- Idling time: cumulative time when no one is running. It exists because the solution, checker and updater run asynchronously (every 5 seconds in this experiment).

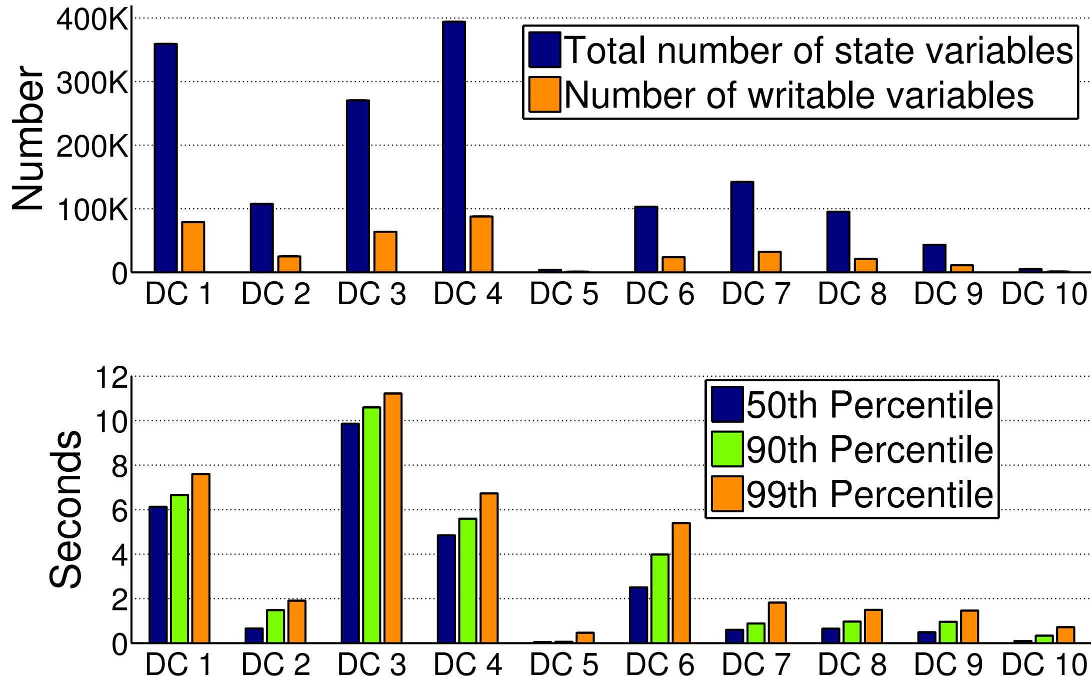

Figure 2.14: Network State Scale & Checker Performance

Figure 2.13 shows that the solution and checker latencies are negligibly small, accounting for only 0.13% and 0.28% of the end-to-end latency on average. The main bottleneck is the time for the updater to apply the link-shutdown commands to the device, which accounts for 57.7% of the end-to-end latency on average.

Coverage: Figure 2.14 shows the deployment scale and the checker latency across the ten datacenters where Statesman is deployed. The number of state variables in each datacenter indicates its size. The largest datacenter ($datacenter_4$) has 394K variables and 7 out of 10 datacenters have over 100K variables. The total number of variables managed by Statesman is currently 1.5 million and it continues to grow as Statesman expands to more datacenters. Since only 23.4% of the variables are writable on average, the size of TS is correspondingly smaller than the OS.

Checker latency: Figure 2.14 also shows that one round of checking takes 0.5 to 7.6 seconds in most datacenters. In the most complex datacenter ($datacenter_3$), the 99th-percentile of checking latency is 11.2 seconds.

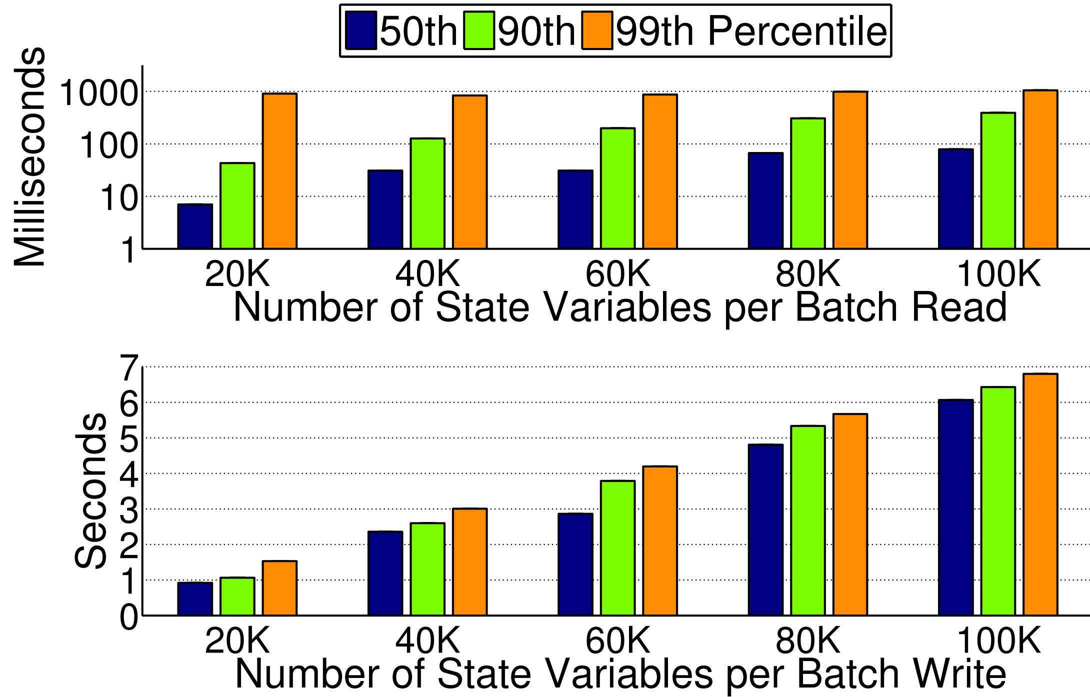

Figure 2.15: Read-write Micro-benchmark Performance

Read-write performance: We stress-test the read-write performance of Statesman by randomly reading and writing varying numbers of state variables. Figure 2.15 shows that the 99th-percentile latency of reading 20K to 100K variables always stays under a second. Since the solutions rarely read more than 100K variables at a time, Statesman's read speed is fast enough. Figure 2.15 also shows that the write latency of Statesman grows linearly with the number of variables, and the 99th-percentile latency of writing 100K variables is under 7 seconds. Since our largest datacenter has fewer than 100K variables in the TS, it takes less than 10 seconds to write the entire TS. In practice, the state-variable writers (i.e., management solutions, monitors, and checkers) rarely write more than 10K variables at a time. Hence, the write latency is usually around 500ms in our normal operations.

2.9 Related Work

Statesman descends from a long line of prior works on software-defined networking [44, 46, 64, 74, 75, 97]. These works enable centralized control of traffic flow by directly manipulating the forwarding states of devices. In contrast, Statesman supports a wider range of network management functions (e.g., device upgrade, link failure mitigation, elastic scaling, etc.).

Similar to Statesman, Onix [90, 91] and Hercules [88] provide a shared network-state platform for all management solutions. But these systems neither resolve conflicts among management solutions, in particular those caused by state variable dependency, nor enforce network-wide invariants.

Pyretic [99], PANE [66], and Maple [128] are recent proposals on making multiple management solutions coexist. Their target is again limited to traffic management solutions. Their composition strategies are specific to such solutions, and are not generalized for the broader class of management solutions that we target. Mesos [79] schedules competing management solutions using the cluster-resource abstraction, which is quite different from our network-state abstraction (e.g., no cross-variable dependency).

Corybantic [98] proposes a different way of resolving conflicts by letting each management solution evaluate others' proposals. As noted previously, such tight coordination, while sometimes beneficial, imposes enormous complexity on the design and testing of the management solutions.

Another approach of hosting multiple management solutions is to partition the network into multiple isolated virtual slices as described Kopone, *et al.* [89] and Sherwood, *et al.* [117]. Compared to the virtual topology model, our network-state model is more fine-grained and flexible. It allows multiple management solutions to manage different levels of states (e.g., power, configuration, and routing) on the same device.

There are also earlier works on invariant verification [85, 86, 95] for the network's forwarding state. In the future, we may incorporate some of these invariant checking algorithms into Statesman.

2.10 Conclusion

In this chapter, we propose the Statesman system. Statesman builds a common platform to run multiple loosely-coupled network management solutions, while preserving network safety and performance. It safely composes uncoordinated and sometimes conflicting management solutions using three distinct views of network state, inspired by version control systems, and a model of dependencies among different parts of network state. Statesman is currently running in ten Microsoft Azure datacenters, along with three diverse applications.

Chapter 3

Hone: Combining End Host and Network for Traffic Management

3.1 Introduction

The cloud providers run a wide variety of applications that generate large amount of traffic. These applications have a complex relationship with the underlying datacenter network, including varying traffic patterns [83], suboptimal TCP parameters [134], traffic bursts that temporarily congest network links (i.e., TCP incast) [131], and elephant flows that overload certain links [39]. To optimize application performance, cloud providers build various traffic management solutions, such as performance monitoring, server load balancing [107], rate limiting [114], and traffic engineering [81, 82].

However, today's traffic-management solutions are constrained by an artificial division between the end hosts and the network. Most existing solutions only rely on network devices. Without access to the application and transport layers, network devices cannot easily associate traffic statistics with applications. For instance, it is difficult to infer the root causes of application performance problems or infer the backlog of application traffic in the sender's socket buffer, when just analyzing

network-level statistics. Furthermore, traffic management solutions are limited by the CPU and memory resources of network devices.

Compared to network devices, end hosts have better visibility into application behavior, greater computational resources, and more flexibility in adopting new functionality. Increasingly, datacenter traffic management capitalizes on the opportunity to move functionality from network devices to end hosts [59, 80, 84, 109, 116, 118, 130, 134, 137]. However, in these works, the hosts are just used as software network devices. The unique advantages of end hosts in traffic management are not fully harnessed.

In order to realize these advantages, the scope of traffic management should expand further, into the host network stack, to measure and analyze fine-grained traffic information. The host network stack can provide detailed data about how applications interact with the underlying network, while remaining application-agnostic. By combining host and network data, traffic management solutions can understand the applications better in order to improve application performance and efficiency of network resource usage.

Bringing the host network stack into traffic management, we face multiple challenges: First, today's end hosts have numerous interfaces for performing network-related functions. Cloud providers use heterogeneous tools to collect TCP logs and kernel network statistics from end hosts (e.g., Windows ETW [31] and Web10G [29]) and they have multiple tools for controlling network behavior (e.g., Linux `tc` [13], `iptables` [18], and Open vSwitch [22]). To simplify traffic management, we need to provide a *uniform* interface for cloud providers to collect measurement data from end hosts and network devices.

Furthermore, cloud providers do not settle on a fixed set of data to measure in advance. Multiple traffic management solutions need to run at the same time, and they need to measure different data from the hosts and devices. Additionally, new

solutions will adopt new measurement data as the application mix and the network design evolve. However, the overhead of measuring all data blindly is prohibitive, especially when the detailed statistics of host network stacks are included in traffic management. Thus, we need a *flexible* way to selectively collect measurements, tailored to the demands of the management solutions.

A final challenge is the efficiency and scalability of traffic management. Although end hosts provide us with detailed statistics, the sheer volume of data poses a scalability challenge for the real-time analysis of the measured data. At the same time, the computational resources of hosts provide an opportunity to execute the analysis logic locally on the same hosts that collect the measurements. To utilize the hosts' resources, the analysis logic of management solutions should be *partitioned* into two parts: one that runs locally with monitoring on the hosts, and the other that runs on a controller with a global view of all the hosts and network devices.

In this chapter, we present a scalable and programmable platform for joint HOst-NEtwork (Hone) traffic management. As shown in Figure 3.1, the architecture includes a logically centralized controller, Hone agents running on each host, and a module interacting with network devices, following the recent trend of Software Defined Networking (SDN). Hone performs monitoring and analysis on streams of measurement data in a fashion of functional reactive programming (FRP) [61, 67, 103]. A management solution can easily define the measurement of various data on hosts and network with minimum collection overhead. The data-analysis logic can be divided into local parts that execute together with measurement on hosts, and global parts running on the controller. To be more specific, two key technical contributions underlie Hone:

Lazy measurement of various data on hosts and network: Hone first abstracts the heterogeneous data on hosts and network devices into a uniform representation of database-like tables. Using our query language on top of the uniform

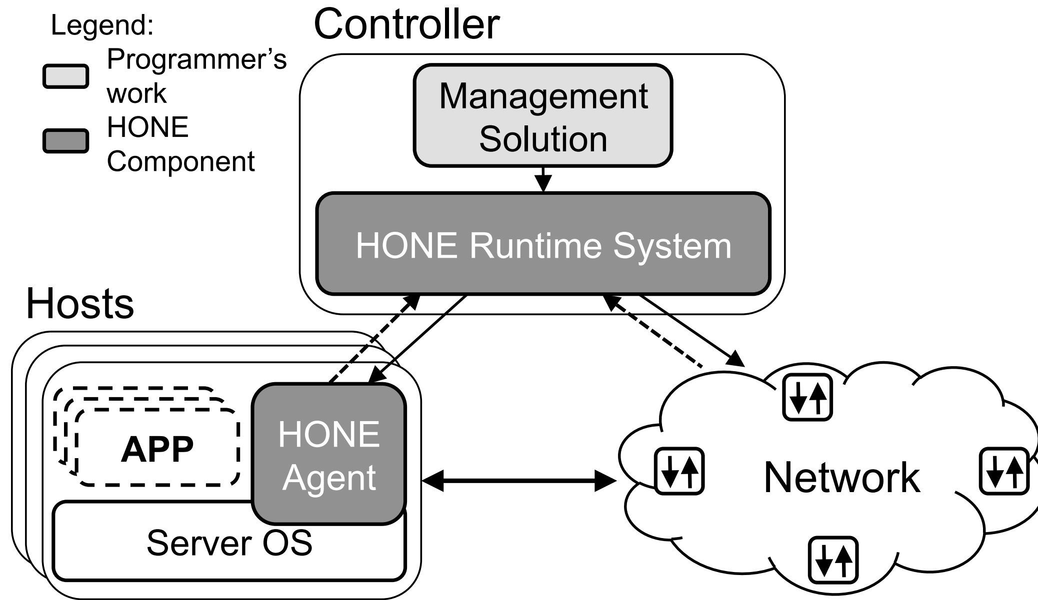

Figure 3.1: Overview of the Hone System

interface, network operators can easily specify what data to measure from various sources. Collecting many statistics for many connections at arbitrary time granularity would be too expensive to support directly. Instead, Hone enables lazy materialization of the measurement data, i.e., the controller and the host agents analyze the queries to collect only the necessary statistics for multiple management solutions at the appropriate frequencies.

Partitioning of analysis logic for local execution on hosts: We have designed a set of data-parallel FRP operators for programming the data-analysis logic. With these operators, programmers can easily link the measurement queries with the analysis logic in a streaming fashion. Our design of the operators further enables partitioning of the analysis logic between the host agents and the controller. To further boost scalability, we have also designed operators for programmers to hierarchically aggregate/process analysis results among multiple hosts.

Hone combines and extends techniques in FRP and distributed systems in a unique way to offer a programmable and scalable platform for joint host-network traffic management in SDN. To demonstrate the effectiveness of our design, we have built a pro-

totype of Hone. Our prototype host agent collects socket activities by intercepting Linux system calls, and measures TCP statistics with Web10G [96, 29]. Our controller interacts with network devices using OpenFlow [97]. The micro-benchmark experiments show that Hone can measure and calculate an application's throughput, and aggregate the results across 128 EC2 machines within a 90th-percentile latency of 58ms.

To further demonstrate the power of the Hone programming framework, we build a collection of canonical management solutions, such as flow scheduling [39, 55], distributed rate limiting [110, 114], network diagnosis [134], etc. These examples demonstrate the expressiveness of our language, as well as the scalability of our data collection and analysis techniques.

Hone focuses on different contexts and problems than prior works in streaming database [42, 54] and MapReduce [57, 58, 108]. Compared to streaming databases which operate on existing data, Hone focuses on how to lazily monitor a minimum necessary set of statistics on hosts, and dynamically generate streams of measurement data for later analysis. MapReduce's key-value programming model fits well with naturally parallelizable data with multiple keys, and it uses data shuffling by keys for reduce operations. In contrast, Hone focuses on the measurement data which are inherently associated with the end hosts that collected them. Hone partitions and places the analysis logic together with monitoring to consume data locally and reduce data-transmission overhead.

3.2 Hone Programming Framework

Hone's programming framework is designed to enable a wide range of traffic management solutions. Traffic management is usually oriented around a three-stage "control

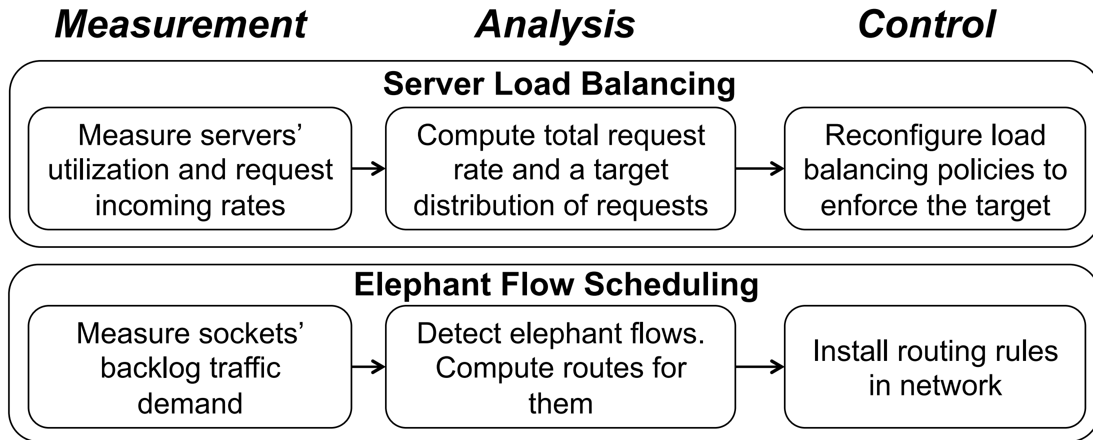

Figure 3.2: Three Stages of Traffic Management

loop" of measurement, data analysis, and control. Figure 3.2 presents two representative management solutions that serve as running examples throughout the chapter:

Server load balancing: The first management solution distributes incoming requests across multiple servers. After measuring the request rate and the workload (e.g., CPU, memory, and bandwidth usage) at each host, the management solution estimates the total request rate, computes a new target division of requests over the hosts, and configures network devices accordingly.

Elephant flow scheduling: The second management solution is inspired by how Hedera [39] and Mahout [55] schedule large flows. After measuring the backlog in the socket buffer for each TCP connection, the management solution identifies the elephant flows and directs them over paths that minimize network congestion.

3.2.1 Measurement: Query on Global Tables

Hone's data model unifies the representation of statistics across a range of formats, locations, types of devices, and modes of access. The Hone controller offers a simple abstraction of a central set of database tables. Programmers can launch sophisticated queries, and rely on Hone to distribute the monitoring to the devices, materialize the necessary tables, transform the data to fit the schema, perform local computations

Table Name	Row for each	Columns
Connections	Connection	App, TCP/UDP five-tuple, TCP-stack statistics.
Applications	Process	Host ID, PID, app's name, CPU/memory usage.
Machines	Host	Host ID, total CPU usage, total memory usage, IP.
Links	Link	IDs/ports of two ends, capacity.
SwitchStats	Switch interface	Switch ID, port, timestamp, per-port counters.

Table 3.1: Global Tables Supported in Hone Prototype

and data reduction, and aggregate the data. The data model reduces a complex and error-prone distributed programming solution to a set of simple, tabular queries that can usually be crafted in just tens of lines of code.

The Hone data model is organized around the protocol layers and the available data sources. Table 3.1 shows the tables that our current prototype supports. On the hosts, Hone collects socket logs and TCP connection statistics, in order to capture the relationship between applications and the network stack while remaining application-agnostic. On the network devices, Hone collects the topology, the routing configurations, and per-port counters using OpenFlow. However, we can easily extend our prototype to support more interfaces (e.g., NetFlow, SNMP) by adding new tables, along with implementations for collecting the data.

Hone offers programmers a familiar, SQL-like query language for collecting the data, as summarized in Table 3.2 [1]. The query language gives programmers a way to state declaratively *what* data to measure, rather than *how*. More sophisticated analysis, transformation, filtering, and aggregation of the data take place in the analysis phase. To illustrate how to create a management solution on Hone, consider the three example queries needed for elephant-flow scheduling:

Backlog in socket buffer: This query generates the data for computing the backlog in the socket buffers:

[1] In the syntax, the star operator (*) glues together the various query components.

```
def ElephantQuery():
  return (
    Select([SrcIp, DstIp, SrcPort, DstPort, BytesWritten, BytesSent]) *
    From(Connections) *
    Every(Seconds 1) )
```

The query produces a stream of tables, with one table every second. In each table, each row corresponds to a single connection and contains the endpoint IP addresses and port numbers, as well as the amount of data written into the socket buffer and sent into the network. Later, the analysis phase can use the per-connection `BytesWritten` and `BytesSent` to compute the backlog in the socket buffer to detect elephant flows.

Active links and their capacities: This query generates a stream of tables with all unidirectional links in the network:

```
def LinkQuery():
  return(
    Select([BeginDevice, EndDevice, Capacity]) *
    From(Links) *
    Every(Seconds 1) )
```

Connection-level traffic by host pair: This query collects the data for computing the traffic matrix:

```
def TrafficMatrixQuery():
  return(
    Select([SrcIp, DstIp, BytesSent, Timestamp]) *
    From(Connections) *
    Groupby([SrcIp,DstIp]) *
    Every(Seconds 1) )
```

The query uses the `Groupby` operator to convert each table (at each second) into a *list* of tables, each containing information about all connections for a single pair of end-points. Later, the analysis phase can sum the `BytesSent` across all connections

<table>
<tr><td>Query</td><td>:=</td><td>Select(Stats) *
From(Table) *
Where(Criteria) *
Groupby(Stat) *
Every(Interval)</td></tr>
<tr><td>Table</td><td>:=</td><td>Connections | Applications
| Machines | Links | SwitchStats</td></tr>
<tr><td>Stats</td><td>:=</td><td>Columns of Table</td></tr>
<tr><td>Interval</td><td>:=</td><td>Integer in Seconds or Milliseconds</td></tr>
<tr><td>Criteria</td><td>:=</td><td>Stat Sign value</td></tr>
<tr><td>Sign</td><td>:=</td><td>$> \mid < \mid \geq \mid \leq \mid = \mid \neq$</td></tr>
</table>

Table 3.2: Measurement Query Language Syntax

in each table in the list, and compute the difference from one time period to the next to produce the traffic matrix.

Together, these queries provide the information needed for the elephant-flow solution. They also illustrate the variety of different statistics that Hone can collect from both hosts and network devices—all within a simple, unified programming framework. Under the hood, the Hone controller analyzes these queries to merge overlapping parts. Then the host agents or the network module will lazily collect only the queried statistics at appropriate frequencies to minimize the measurement overhead.

3.2.2 Analysis: Data-Parallel Operators

Hone enables programmers to analyze data across multiple hosts, without worrying about the low-level details of communicating with the hosts or tracking their failures. Hone's FRP-based data-parallel operators allow programmers to say *what* analysis to perform, rather than *how*. Programmers can associate their own functions with the operators to apply these functions across sets of hosts, as if the streams of tabular measurement data were all available at the controller. Yet, Hone gives the programmers a way to express whether their functions can be (safely) applied in parallel across

data from different hosts, to enable the runtime system to reduce the bandwidth and processing load on the controller by executing these functions at the hosts. Hone's data-parallel operators include the following:

- `MapSet(`f`)`: Apply function f to every element of a stream in the set of streams, producing a new set of streams.
- `FilterSet(`f`)`: Create a new set of streams that omits stream elements e for which $f(e)$ is false.
- `ReduceSet(`f`,`i`)`: "Fold" function f across each element for each stream in the set, using i as an initializer. In other words, generate a new set of streams where $f(\ldots f(f(i, e_1), e_2)\ldots, e_n)$ is the nth element of each stream when e_1, e_2, ..., e_n were the first n elements of the original stream.
- `MergeHosts()`: Merge a set of streams on the hosts into one single global stream. (Currently in Hone, the collection of network devices already generate a single global stream of measurement data, given that our prototype integrates with one SDN controller to access data from network devices.)

`MapSet`, `FilterSet`, and `ReduceSet` are parallel operators. Using them indicates that the analysis functions associated with the operators can run in a distributed fashion. It offers Hone knowledge of how to partition the analysis logic into the local part that can run in parallel on each host and the global part that sits on the controller. Hone also enables analysis on a single global stream with corresponding operators, such as `MapStream`, `FilterStream`, and `ReduceStream`. To combine queries and analysis into a single program, the programmer simply associates his functions with the operators, and "pipes" the result from one query or operation to the next (using the $>>$ operator).

Consider again the elephant-flow scheduling solution, which has three main parts in the analysis stage:

Identifying elephant flows: Following the approach suggested by Curtis *et al.* [55], the function `IsElephant` defines elephant flows as the connections with a socket backlog (i.e., the difference between bytes `bw` written by the application and the bytes `bs` acknowledged by the recipient) in excess of 100KB:

```
def IsElephant(row):
  [sip,dip,sp,dp,bw,bs] = row
  return (bw-bs > 100)

def DetectElephant(table):
  return (FilterList(IsElephant, table))

EStream = ElephantQuery() >>
          MapSet(DetectElephant) >>
          MergeHosts()
```

`DetectElephant` uses `FilterList` (the same as `filter` in Python) to apply `IsElephant` to select only the rows of the connection table that satisfy this condition. Finally, `DetectElephant` is applied to the outputs of `ElephantQuery`, and the results are merged across all hosts to produce a single stream `EStream` of elephant flows at the controller.

Computing the traffic matrix: The next analysis part computes the traffic matrix, starting from aggregating the per-connection traffic volumes by source-destination pair, and then computing the difference across consecutive time intervals:

```
TMStream = TrafficMatrixQuery() >>
           MapSet(MapList(SumBytesSent) >>
           ReduceSet(CalcThroughput, {}) >>
           MergeHosts() >>
           MapStream(AggTM)
```

The query produces a stream of lists of tables, where each table contains the per-connection traffic volumes for a single source-destination pair at a point in time.

`MapList` (i.e., the built-in `map` in Python) allows us to apply a custom function `SumBytesSent` that aggregates the traffic volumes across connections in the same table, and `MapSet` applies this function over time. The result is a stream of tables, which each contains the cumulative traffic volumes for every source-destination pair at a point in time. Next, the `ReduceSet` applies a custom function `CalcThroughput` to compute the differences in the total bytes sent from one time to the next. The last two lines of the analysis merge the streams from different hosts and apply a custom function `AggTM` to create a global traffic matrix for each time period at the controller.

Constructing the topology: The last part of our analysis builds a network topology from the link tables produced by `LinkQuery`, which is abstracted as a single data stream collected from the network:

```
TopoStream = LinkQuery() >>
             MapStream(BuildTopo)
```

The auxiliary `BuildTopo` function (not shown) converts a single table of links into a graph data structure useful for computing paths between two hosts. The `MapStream` operator applies `BuildTopo` to the stream of link tables to generate a stream of graph data structures.

3.2.3 Control: Uniform and Dynamic Policy

In controlling end hosts and network devices, the cloud providers have to use various interfaces. For example, network operators use `tc`, `iptables`, or Open vSwitch on hosts to manage traffic, and they use SNMP or OpenFlow to manage the network devices. For the purpose of managing traffic, these different control interfaces can be unified because they share the same pattern of generating control policies: for a group of connections satisfying predicate, define what actions to take. Therefore, Hone offers cloud providers a uniform way of specifying control policies as *predicate*

Policy	:=	[Rule]+
Rule	:=	*if* Predicate *then* Action
Predicate	:=	Field = *value* | Predicate and Predicate | Predicate or Predicate
Field	:=	AppName | SrcHost | DstHost | Headers
Headers	:=	SrcIP | DstIP | SrcPort | DstPort | $\cdots$
Action	:=	rate-limit *value* | forward-on-path *path*

Table 3.3: Control Policy in Hone Prototype

+ *action* clauses, and Hone takes care of choosing the right control implementations, e.g., we implement *rate limiting* using `tc` and `iptables` in the host agent.

The predicate can be network identifiers (e.g., IP addresses, port numbers, etc.). But this would force the programmer to map his higher-level policies into lower-level identifiers, and identify changes in which connections satisfy the higher-level policies. Instead, we allow programmers to identify connections of interest based on higher-level attributes, and Hone automatically tracks which traffic satisfies these attributes as connections come and go. Our predicates are more general than network-based rule-matching mechanisms in the sense that we can match connections by applications with the help of hosts. Table 3.3 shows the syntax of control policies, each of which our current prototype supports.

Continuing the elephant-flow solution, we define a function `Schedule` that takes inputs of the detected elephant flows, the network topology, and the current traffic matrix. It assigns a routing path for each elephant flow with a greedy *Global First Fit* [39] strategy, and creates a Hone policy for forwarding the flow along the picked path. Other non-elephant flows are randomly assigned to an available path. The outputs of policies by `Schedule` will be piped into `RegisterPolicy` to register them with Hone for execution.

```
def Schedule(elephant, topo, traffic):
  routes = FindRoutesForHostPair(topo)
```

```
  policies = []
  for four_tuples in elephant:
    path = GreedilyFindAvailablePath(four_tuples, routes, traffic)
    predicate = four_tuples
    action = forward-on-path path
    policies.append([predicate, action])
  return policies
```

3.2.4 All Three Stages Together

Combining the measurement, analysis, and control phases, the complete program merges the data streams, feeds the data to the `Schedule` function, and registers the output of policies. With this concrete example of an elephant-flow detection and scheduling solution, we have demonstrated the simple and straightforward way of designing traffic management solutions in Hone programming framework.

```
def ElephantFlowDetectionScheduling():
  MergeStreams([EStream, TopoStream, TMStream]) >>
  MapStream(Schedule) >>
  RegisterPolicy()
```

3.3 Efficient and Scalable Execution

Monitoring and controlling many connections of applications on many hosts could easily overwhelm a centralized controller. Hone overcomes the scalability challenge in four ways. First, a distributed directory service dynamically tracks the mapping of management solutions to hosts, applications, and connections. Second, the Hone agents lazily materialize virtual tables based on the current queries. Third, the controller automatically partitions each management solution into global and local portions, and distributes the local part over the host agents. Fourth, the hosts automat-

ically form a tree to aggregate measurement data based on user-defined aggregation functions to limit the bandwidth and computational overhead on the controller.

3.3.1 Distributed Directory Service

Hone determines which hosts should run each management solution, based on which applications and connections match the queries and control policies. Hone has a directory service that tracks changes in the active hosts, applications, and connections. To ensure scalability, the directory has a two-tiered structure where the first tier (tracking the relatively stable set of active hosts and applications) runs on the controller, and the second tier (tracking the large and dynamic collection of connections) runs locally on each host. This allows the controller to decide which hosts to inform about a query or control policy, while relying on each local agent to determine which connections to monitor or control.

Tracking hosts and applications: Rather than build the first tier of the directory service as a special-purpose component, we leverage the Hone programming framework to run a standing query:

```
def DirectoryService():
  (Select([HostID, App]) *
   From(Applications) *
   Every(Seconds 1) ) >>
  ReduceSet(GetChangeOfAppAndHealth,[]) >>
  MergeHosts() >>
  MapStream(NotifyRuntime)
```

This query returns the set of active hosts and their applications. `GetChangeOfAppAndHealth` identifies changes in the set of applications running on each host, and the results are aggregated at the controller. The controller uses its connectivity to each host agent as the host's health state, and the host agent uses `ps` to find active applications.

Tracking connections: To track the active connections, each host runs a Linux kernel module we build that intercepts the socket system calls (i.e., `connect`, `accept`, `send`, `receive`, and `close`). Using the kernel module, the Hone agent associates each application with the TCP/UDP connections it opens in an event-driven fashion. This avoids the inevitable delay of poll-based alternatives, such as `lsof` and `/proc`.

3.3.2 Lazily Materialized Tables

Hone gives programmers the abstraction of access to diverse statistics at any time granularity. To minimize measurement overhead, Hone lazily materializes the statistics tables by measuring only certain statistics, for certain connections, at certain times, as needed to satisfy the queries. The Hone controller analyzes the queries from the management solutions, and identifies what queries should run on hosts or network devices. For queries to run on hosts, the host agents merge the collection of overlapping statistics to share among management solutions. The agents collect only the statistics as specified in the queries with appropriate measurement techniques, instead of measuring all statistics in the virtual tables. The network module also merges the collection of shared statistics among queries, and collects the requested statistics from network devices using OpenFlow.

Returning to the elephant-flow solution, the controller analyzes the `ElephantQuery` and decides to run the query on the hosts. Since the query does not constrain the set of hosts and applications, the controller instructs *all* local agents to run the query. Each Hone agent periodically measures the values of `SrcIP`, `DstIP`, `SrcPort`, `DstPort`, and `BytesSent` from the network stack (via Web10G [29]), and collects the `BytesWritten` from the kernel module discussed earlier in §3.3.1. Similarly, Hone queries the network devices for the `LinkQuery` data; in our prototype, we interact with network devices using the OpenFlow protocol. Hone does not collect or record

any unnecessary data. Lazy materialization supports a simple and uniform data model while keeping measurement overhead low.

3.3.3 Host-Controller Partitioning

In addition to selectively collecting traffic statistics, the hosts can significantly reduce the resulting data volume by filtering or aggregating the data. For example, the hosts could identify connections with a small congestion window, sum throughputs over all connections, or find the top k flows by traffic volume.

However, parallelizing an arbitrary controller program would be difficult. Instead, Hone provides a `MergeHosts` operator that explicitly divides a management solution into its local and global parts. Analysis functions before `MergeHosts` run locally on each host, whereas functions after `MergeHosts` run on the controller. Hone hides the details of distributing the computation, communicating with hosts and network devices, and merging the results. Having an explicit `MergeHosts` operator obviates the need for complex code analysis for automatic parallelization.

Hone coordinates the parallel execution of management solutions across a large group of hosts[2]. We first carry out industry-standard clock synchronization with NTP [30] on all hosts and the controller. Then the Hone runtime stamps each management solution with its creation time t_c. The host agent dynamically adjusts when to start executing the solution to time $t_c + nT + \epsilon$, where n is an integer, ϵ is set to 10ms, and T is the period of the management solution (as specified by the `Every` statement). Furthermore, the host agent labels the local execution results with a logical sequence number (i.e., n), in order to tolerate the clock drifts among hosts. The controller buffers and merges the data bearing the same sequence number into a

[2]The Hone controller ships the source code of the local portion of management solutions to the host agent. Since Hone programs are written in Python, the agent can execute them with its local Python interpreter, and thus avoids the difficulties of making the programs compatible with diverse environments on the hosts.

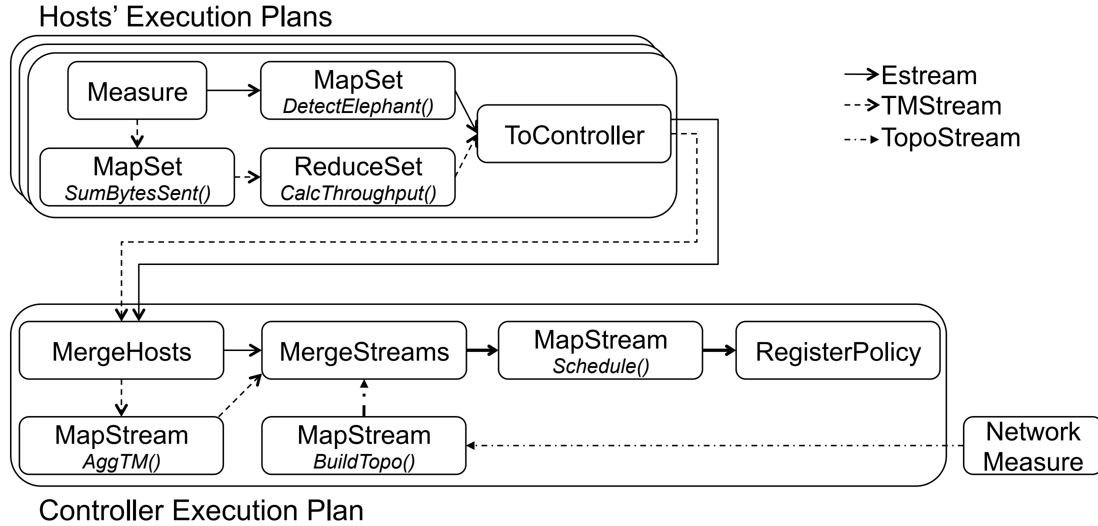

Figure 3.3: Partitioned Execution Plan of Elephant-Flow Solution

single collection, releasing data to the global portion of management solution when either receiving from all expected hosts or timing out after T.

Using our elephant-flow-scheduling solution, Figure 3.3 shows the partitioned execution plan of the management program. Recall that we merge `EStream`, `TMStream`, and `TopoStream` to construct the program. The measurement queries are interpreted as parallel *Measure* operations on the host agents, and the query of switch statistics from the network module. Hone executes the `EStream` and `TMStream` parts on each host in parallel (to detect elephant flows and calculate throughputs, respectively), and streams these local results to the controller (i.e., *ToController*). The merged local results of `TMStream` pass through a throughput aggregation function (*AggTM*), and finally merge together with the flow-detection data and the topology data from `TopoStream` to feed the *Schedule* function.

3.3.4 Hierarchical Data Aggregation

Rather than transmit (filtered and aggregated) data directly to the controller, the hosts construct a hierarchy to combine the results using user-specified functions.

Hone automatically constructs a k-ary tree rooted at the controller[3] and applies a `TreeMerge` operator at each level. All hosts running the solution are leaves of the tree. For each group of b hosts, Hone chooses one to act as their parent in the tree. These parents are grouped again to recursively build the tree towards the controller. User-defined functions associated with `TreeMerge` are applied to all non-leaf nodes of the tree to aggregate data from their children. Hone is unique among research efforts on tree-based aggregation [126, 133], since prior works focus on aggregating data with *a priori* knowledge of the data structure, and don't allow users to specify their own aggregation functions.

Many aggregation functions used in traffic management are both commutative and associative; such functions can be applied hierarchically without compromising correctness. For example, determining the top k values for heavy-hitter analysis is amenable to either direct processing across all data or to breaking the data into subsets for intermediate analysis and combining the results downstream. Calculating the total throughput of connections across all hosts can also be calculated in a distributed manner, as the arithmetic sum is also a commutative and associative function.

Making the user-defined aggregation functions to be both associative and commutative ensures that Hone can apply them correctly in a hierarchical manner. Using `TreeMerge`, Hone assumes that the associated functions have the required properties, avoiding the semantics analysis. `TreeMerge` is similar to `MergeHosts` in the sense that they both combine local data streams from multiple hosts into one data stream on the controller, and intermediate hosts similarly buffer data until they receive data from all their children or a timeout occurs. But with `TreeMerge`, Hone also applies a user-defined aggregation function, while `MergeHosts` simply merges all hosts' data at the controller without intermediate reduction.

[3]The runtime uses information from the directory service to discover and organize hosts.

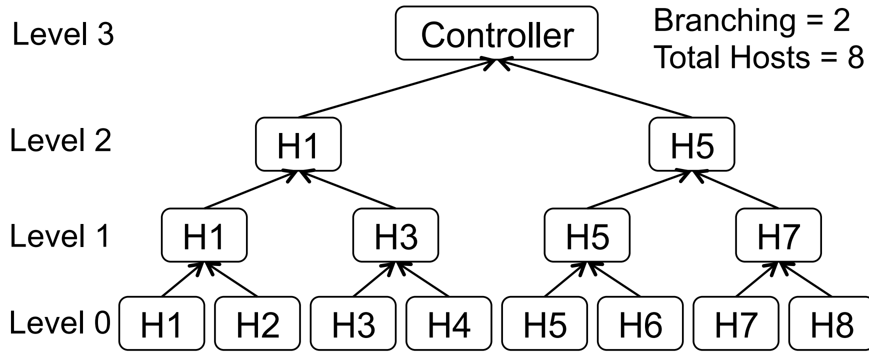

Figure 3.4: Aggregation Tree: 8 Hosts with Branching of 2

The algorithm of constructing the aggregation tree is an interesting extensible part of Hone. We can group hosts based on their network locality, or we can dynamically monitor the resource usage on hosts to pick the one with most available resource to act as the intermediate aggregator. In our prototype, we leave those interesting algorithms to future works, but offer a basic one of incrementally building the tree by when hosts join the Hone system. Subject to the branching factor b, the newly joined leaf greedily finds a node in one level up with less than b children, and links with the node if found. If not found, the leaf promotes itself to one level up, and repeats the search. When the new node reaches the highest level and still cannot find a place, the controller node moves up one level, which increases the height of the aggregation tree. Figure 3.4 illustrates an aggregation tree under the basic algorithm when 8 hosts have joined and b is 2.

3.4 Performance Evaluation

In this section, we present micro-benchmarks on our Hone prototype to evaluate measurement overhead, the execution latency of management solutions, and the scalability; §3.5 will demonstrate the expressiveness and ease-of-use of Hone using several canonical traffic management solutions.

We implement the Hone prototype in combination of Python and C. The Hone controller provides the programming framework and runtime system, which partitions the management solutions, instructs the host agents for local execution, forms the aggregation hierarchy, and merges the data from hosts for the global portion of program execution. The host agent schedules the installed management solutions to run periodically, executes the local part of the program, and streams the serialized data to the controller or intermediate aggregators. We implement the network part of the prototype as a custom module in Floodlight [9] to query switch statistics and install routing rules.

Our evaluation of the prototype focuses on the following questions about our design decisions in §3.2 and §3.3.

1. How efficient is the host-based measurement in Hone?
2. How efficiently does Hone execute entire management solutions?
3. How much overhead does lazy materialization save?
4. How effectively does the controller merge data from multiple end hosts using the hierarchical aggregation?

We run the Hone prototype and carry out the experiments on Amazon EC2. All instances have 30GB memory and 8 virtual cores of 3.25 Compute Units each[4].

3.4.1 Performance of Host-Based Measurement

The Hone host agent collects TCP connection statistics using the Web10G [29] kernel module. We evaluate the measurement overhead in terms of time, CPU, and memory usage as we vary the number of connections running on the host. To isolate the measurement overhead, we run a simple management solution that queries a few randomly-chosen statistics of all connections running on the host every one second

[4]One EC2 Compute Unit provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or Xeon processor.

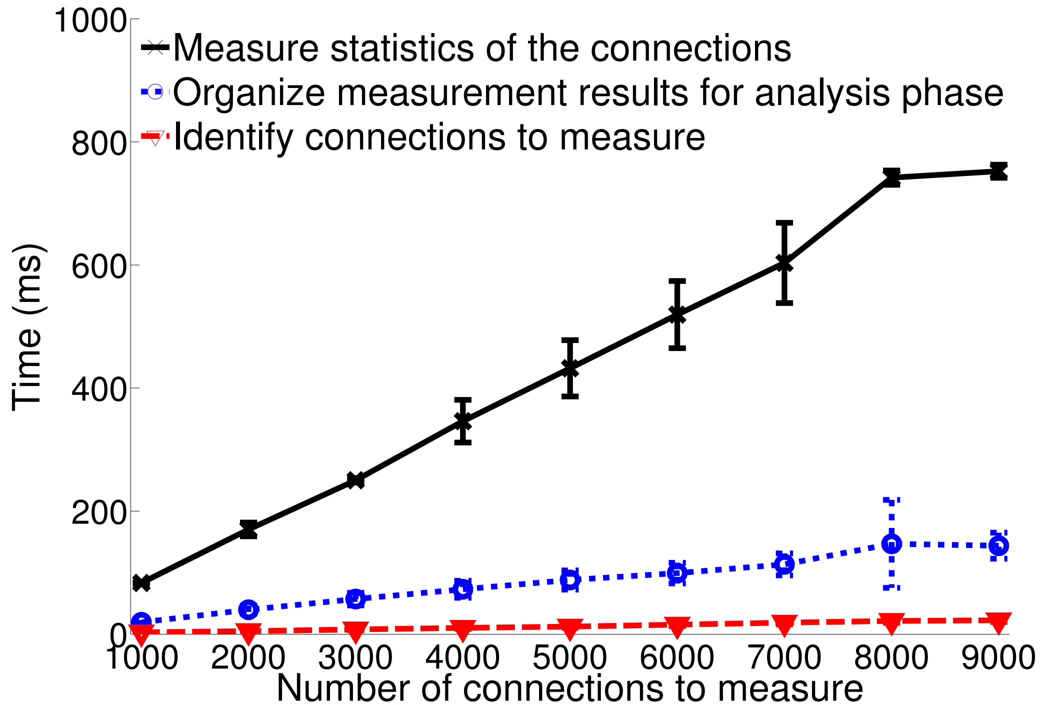

Figure 3.5: Overhead of Collecting Connection Statistics

(we choose the four tuples, bytes of sent data, and the congestion window size). Our experiment consists of three EC2 instances: one for the controller, and two running the Hone agent.

To collect the statistics, the host agent must first identify what connections to measure. Then the agent queries the kernel via Web10G to retrieve the statistics. Finally, the agent organizes the statistics in the schema specified by the query and feeds the result to the management program. In Figure 3.5, we break down the latency in each portion. For each fixed number of connections, we run the management solution for five minutes (i.e., about 300 iterations), and plot the average and standard deviation of time spent in each portion.

Figure 3.5 shows that the agent performs well, measuring 5000 connections in an average of 532.6ms. The Web10G measurement takes the biggest portion–432.1ms, and the latency is linear in the number of active connections. The time spent in identifying connections to measure is relatively flat, since the agent tracks the relevant

connections in an event-driven fashion via the kernel module of intercepting socket calls. The time spent in organizing the statistics rises slowly as the agent must go through more connections to format the results into the query's schema. The results set lower limit for the periods of management solutions that need measurement of different numbers of connections. The CPU and memory usage of the agent remain stable throughout the experiments, requiring an average of 4.55% CPU of one core and 1.08% memory of the EC2 instance.

3.4.2 Performance of Management Solutions

Next, we evaluate the end-to-end performance of several management solutions. To be more specific, we evaluate the latency of finishing one *round* of a solution: from the agent scheduling a solution to run, measuring the corresponding statistics, finishing the local analysis, sending the results to the controller, the controller receiving the data, till the controller finishing the remaining parts of the management program. We run three different kinds of management solutions which have a mix of leverages of hosts, network devices, and the controller in Hone, in order to show the flexibility of Hone adapting to different traffic management solutions. All experiments in this subsection run on an 8-host-10-switch fat-tree topology [38]. The switches are emulated by running Open vSwitch on an EC2 instance.

- ***Task1*** calculates the throughputs of all `iperf` connections on each host, sums them up, and aggregates the global `iperf` throughput at the controller. This solution performs most of the analysis at the host agents, leaving very few work for the controller. Every host launches 100 `iperf` connections to another randomly chosen host.
- ***Task2*** queries the topology and statistics from the network, and uses the per-port counters on the network devices to calculate the current link utilization. This solution uses the network module in Hone a lot to measure data, and runs com-

putation work on the controller. *Task2* is performed under the same setting of running `iperf` as *Task1*.

- ***Task3*** collects measurement data from the hosts to detect connections with a small congestion window (i.e., which perform badly). It also queries the network to determine the forwarding path for each host pair. The solution then diagnoses the shared links among those problematic flows as possible causes of the bad network performance. *Task3* is a joint host-network job, which runs its computation across hosts, network, and the controller. *Task3* is still under the same setting, but we manually add rules on two links to drop 50% of packets for all flows traversing the links, emulating a lossy network.

Figure 3.6 illustrates the cumulative distribution function (CDF) of the latency for finishing one round of execution, as we run 300 iterations for each solution. We further break down the latency into three parts: the execution time on the agent or the network, the data-transmission time from the host agent or network module to the controller, and the execution time on the controller. In Figure 3.7, we plot the average latency and standard deviation for each part of the three solutions. *Task1* finishes one round with a 90th-percentile latency of 27.8ms, in which the agent takes an average of 17.8ms for measurement and throughput calculation, the data transmission from 8 hosts to the controller takes another 7.7ms, and the controller takes the rest. Having a different pattern with *Task1*, *Task2*'s 140.0ms 90th-percentile latency is consisted of 87.5ms of querying the network devices via Floodlight and 8.9ms of computation on the controller (the transmission time is near zero since Floodlight is running on the controller machine). *Task3*'s latency increases as it combines the data from both hosts and the network, and its CDF also has two stairs due to different responsiveness of the host agents and the network module.

Table 3.4 summarizes the average CPU and memory usage on the host agent and the controller when running the solution. The CPU percentage is for one core of

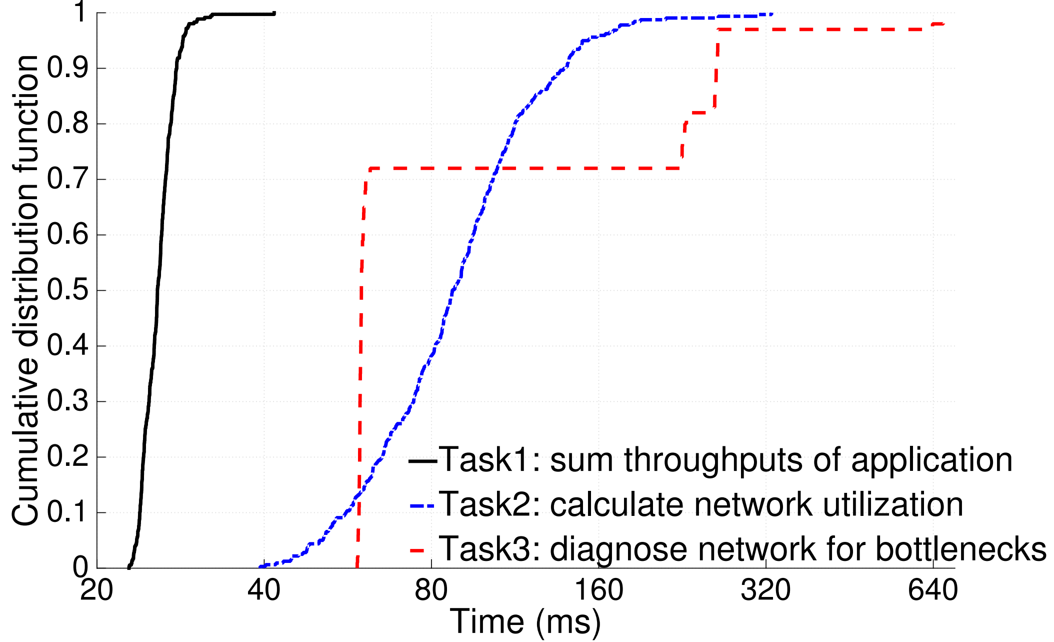

Figure 3.6: Latency of One Round of Execution of Management Solutions

	CPU Agent	Memory Agent	CPU Controller	Memory Controller
Task1	3.71%	0.94%	0.67%	0.10%
Task2	N/A	N/A	0.76%	1.13%
Task3	7.84%	1.64%	1.03%	0.11%

Table 3.4: Average CPU and Memory Usage of Execution

8 cores of our testbed machines. The results show that Hone's resource usage are bound to the running management solutions: *Tasks3* is the most complex one with flow detection/rate calculation on the hosts, and having the controller join host and network data.

3.4.3 Effects of Lazy Materialization

Hone lazily materializes the contents of the statistics tables. We evaluate how much overhead the feature can save for measurement efficiency in Hone.

We set up two applications (A and B) with one thousand active connections each on a host. We run multiple management solutions with different queries over

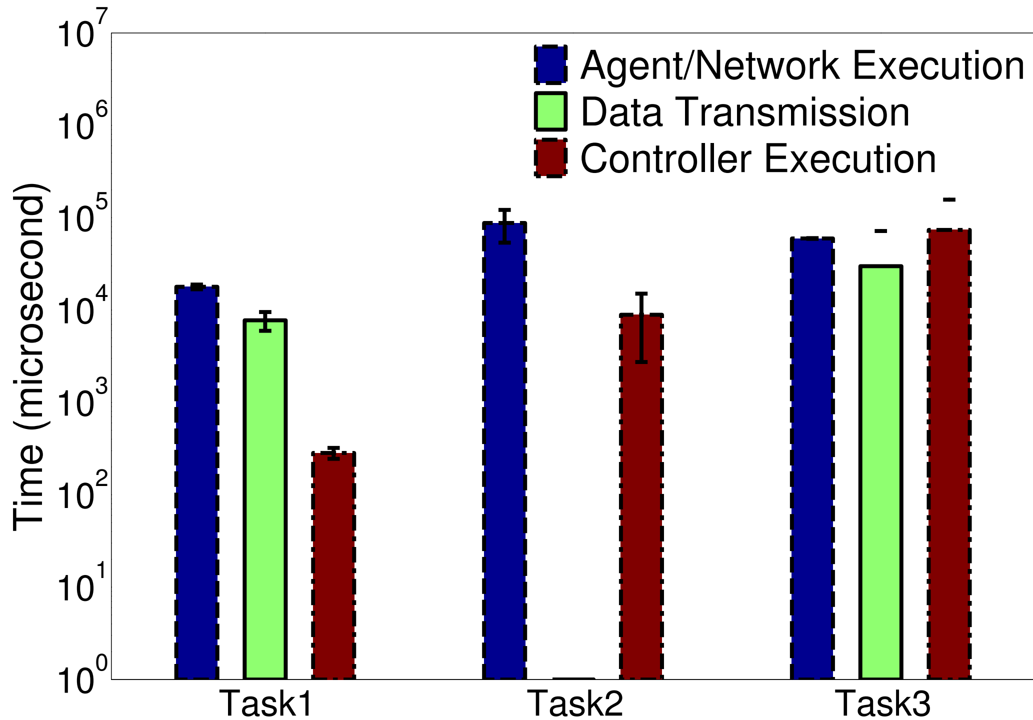

Figure 3.7: Breakdown of Execution Latency

the statistics to evaluate the measurement overhead in terms of latency. Figure 3.8 illustrates the average and standard deviation of the latencies for different queries. The first program queries all 122 TCP-stack statistics available in Web10G of all two thousands connections, and all applications' CPU and memory usage. The following ones query various statistics of `Connections` or `Applications` tables with details shown on Figure 3.8.

The lazy materialization of the tables lowers the measurement overhead by either measuring a subset of tables (*Query1* vs. others), rows (number of connections in *Query1* vs. *Query2* and *Query3*), and columns (number of statistics in *Query2* vs. *Query3*). The high overhead of *Query4* is due to the implementation of CPU measurement, which is, for each process, one of the ten worker threads on the agent keeps running for 50ms to get a valid CPU usage.

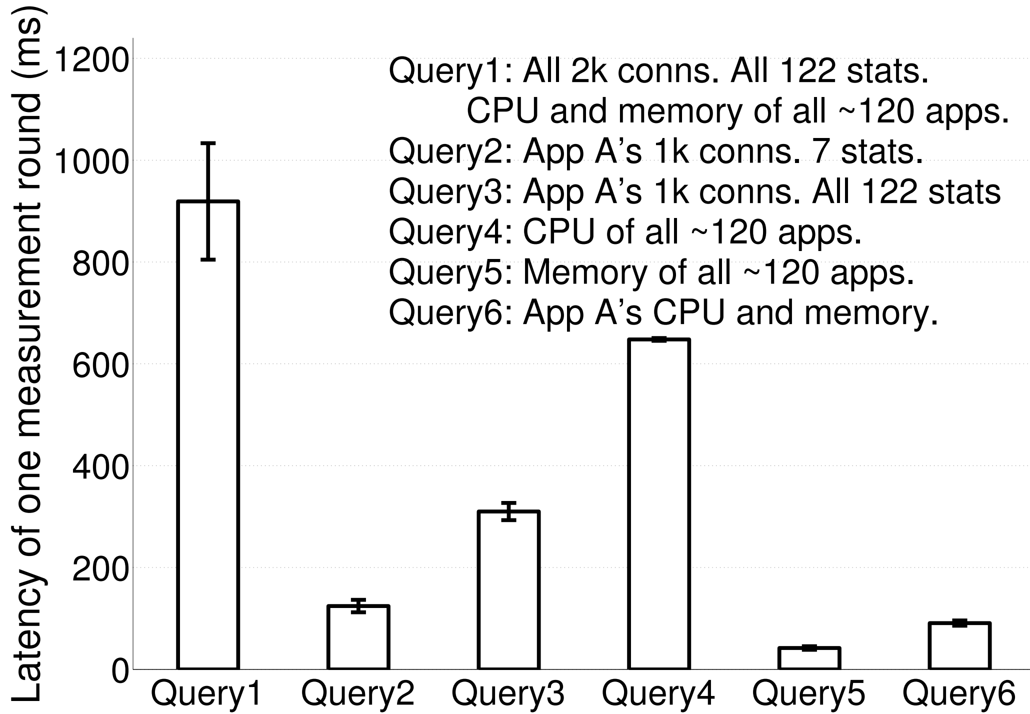

Figure 3.8: Effects of Lazy Materialization

3.4.4 Evaluation of Scalability in Hone

We will evaluate the scalability of Hone from two perspectives. First, when Hone controller partitions the management program into local and global parts of execution, the controller will handle the details of merging the local results processed in the same time period from multiple hosts, before releasing the merged result to the global part of execution. Although the host clocks are synchronized via NTP as mentioned in §3.3.3, the clocks still drift slightly over time. resulting in a buffering delay at the controller. Now we will evaluate how well the buffering works in terms of the time difference between when the controller receives the first piece of data and when the controller receives all the data bearing the same sequence number.

To focus on the merging performance, we use the *Task1* in §3.4.2. All hosts will directly send their local results to the controller without any hierarchical aggregation. Each run of the experiment lasts 7 minutes, containing about 400 iterations. We repeat the experiment, varying the number of hosts from 16 to 128.

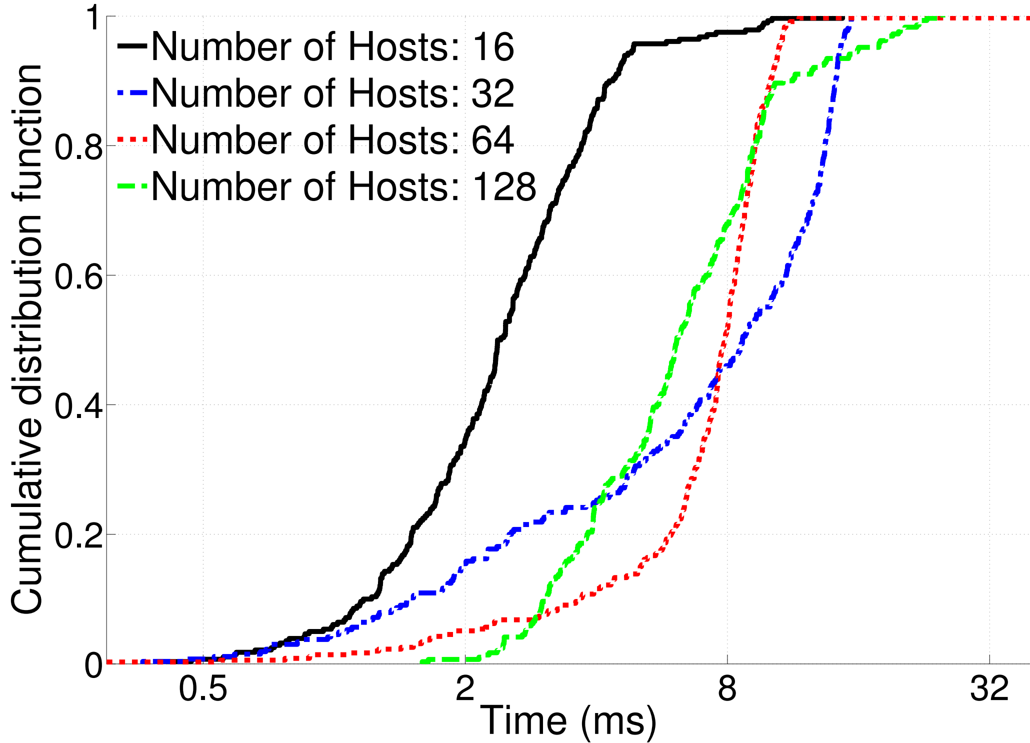

Figure 3.9: Buffering Delay of Merging Data from Hosts on Controller

Figure 3.9 shows the CDFs of the latencies for these experiments. The 90th-percentile of the controller's buffering delay is 4.3ms, 14.2ms, 9.9ms, and 10.7ms for 16, 32, 64, and 128 hosts respectively. The results show that the synchronization mechanism on host agents work well in coordinating their local execution of a management solution, and the controller's buffering delay is not a problem in supporting traffic management solutions whose execution periods are typically in seconds.

After evaluating how the controller merges distributed collection of data, we would evaluate another important feature of Hone for scalability–the hierarchical aggregation among the hosts. We continue using the same management solution of summing the application's throughputs across hosts. However, we switch to using the `TreeMerge` operator to apply the aggregation function. In this way, the solution will be executed by Hone through a k-ary tree consisted of the hosts.

Number of Hosts	CPU Agent	Memory Agent	CPU Controller	Memory Controller
16	4.19%	0.96%	1.09%	0.05%
32	4.93%	0.96%	1.27%	0.05%
64	5.26%	0.97%	1.31%	0.06%
128	4.80%	0.97%	2.36%	0.07%

Table 3.5: Average CPU/Memory Usage with Hierarchical Aggregation

In this experiment, we fix the branching factor k of the hierarchy to 4. We repeat the experiment with 16, 32, 64, and 128 hosts, in which case the height of the aggregation tree is 2, 3, 3, and 4 respectively. Figure 3.10 shows the CDFs of the latencies of one round of execution, which captures the time difference from the earliest agent starting its local part to the controller finishing the global part. The 90th-percentile execution latency increases from 32.2ms, 30.5ms, 37.1ms, to 58.1ms. Table 3.5 shows the average CPU and memory usage on the controller and the host agent. The host agent's CPU and memory usage come from the agent that multiplexes as local-data generator and the intermediate aggregators in all levels of the k-ary tree. It shows the maximum overhead that the host agent incurs when running in a hierarchy.

From the results above, we can conclude that Hone's own operations pose little overhead to the execution of management solutions. The performance of management solutions running in Hone will be mainly bound by their own program complexities, and the amount of data they need to process or transmit.

3.5 Case Studies

We have shown the micro-benchmark evaluation of Hone to demonstrate its efficiency and scalability. Now we will illustrate the expressiveness and ease-of-use of Hone by showing how we build a diversity of traffic management solutions in data centers.

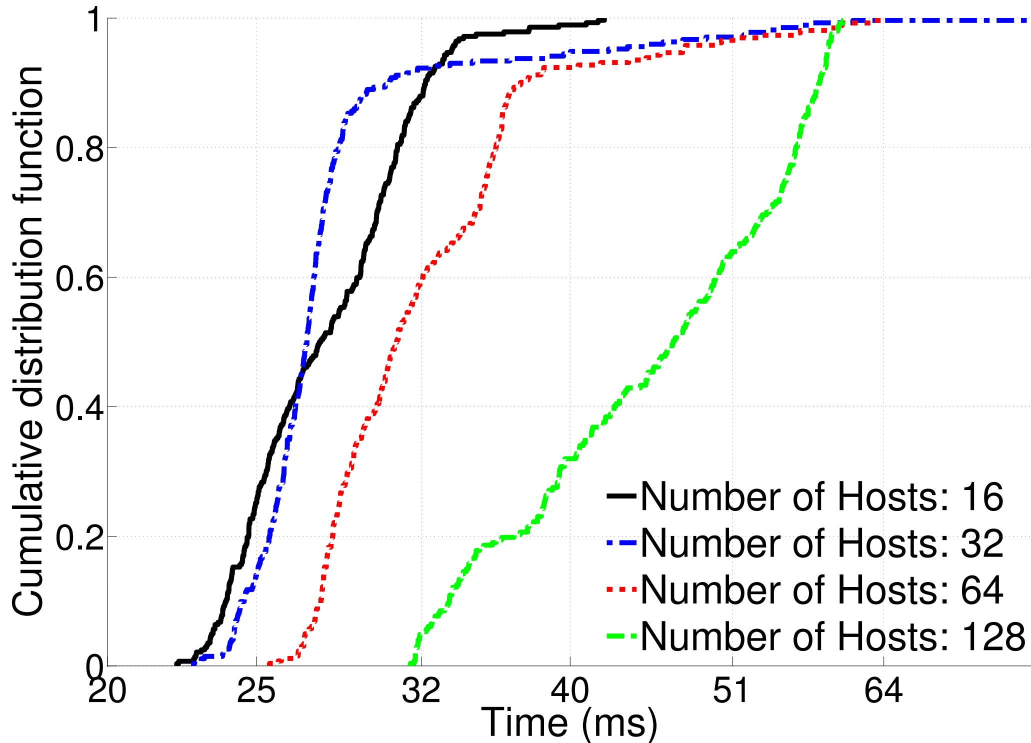

Figure 3.10: End-to-end Execution Latency with Hierarchical Aggregation

Table 3.6 lists all the management solutions that we have built, ranging from conventional management operations in data centers (e.g., calculating link utilizations) to recent proposals (e.g., network performance diagnosis [134]). Those conventional traffic management solutions can actually serve as basic building blocks for more complex management solutions. The network operators can compose the code of those Hone programs to construct their own. Hone is an open-source project, and code for the management programs are also available at `http://hone.cs.princeton.edu/examples`.

In the following subsections, we pick two management solutions as case studies to illustrate more details, and evaluate the Hone-based solutions.

Management Task	Lines of Code
Summing application's throughputs	70
Monitoring CPU and memory usage	24
Collecting connection TCP statistics	19
Calculating traffic matrix	85
Calculating link utilizations	48
Discovering network topology	51
Network performance diagnosis	56
Hone's directory service	31
Elephant flow scheduling	140
Distributed rate limiting	74

Table 3.6: Hone-based Traffic Management Solutions

3.5.1 Elephant Flow Scheduling

In data centers, it is important to detect elephant flows with high traffic demands and properly route them to minimize network congestion. With Hone, we can easily build such a management solution to schedule the elephant flows. This example takes the scheduling strategies and the elephant-flow detection threshold from Hedera [39] and Mahout [55]. We implement Hedera's *Global-first-fit* routing strategy with 140 lines of code in Hone. The code of the management solution has been already shown in previous sections as an example.

We deploy Hone on EC2 instances to emulate a datacenter network with a 8-host-10-switch fat-tree topology (the switches are instances running Open vSwitch). We repeat an all-to-all data shuffle of 500MB (i.e., a 28GB shuffle) for 10 times. The Hone-based solution finishes the data shuffle with an average of 82.7s, compared to 103.1s of using ECMP. The improvement over shuffle time is consistent with Hedera's result.

3.5.2 Distributed Rate Limiting

Distributed rate limiting in datacenters is used to control the aggregate network bandwidth used by an application, which runs on multiple hosts. It can help the application's owner to control the total cost under a pay-as-you-go billing model.

Prior works [110, 114] proposed mechanisms to make distributed rate limiters collaborate as a single, aggregate global limiter inside the network. Hone enables distributed rate limiting from the host side, which would introduce less overhead as the hosts have more computational power than the network devices, and better visibility into the traffic demand of applications.

In Hone, the network operators do not need to worry about the complexity of collecting throughputs from multiple hosts in a synchronized way. Instead, they just need to write a simple program that sums up throughputs of an application's connections on each host, aggregates the throughputs across hosts, and then calculates their rate-limiting policies accordingly. The code written in Hone are shown below:

```
def DistributedRateLimiting():
  (Select([App, SrcIp, DstIp,
           BytesSent, Timestamp]) *
   From(Connections) *
   Where(App == X) *
   Every(Seconds 1) ) >>
  ReduceSet(CalculateThroughput, {}) >>
  MapSet(LocalAgg) >>
  ReduceSet(MovingAverage, initValue) >>
  MergeHosts() >>
  MapStream(GenerateRateLimitPolicy) >>
  RegisterPolicy()
```

We run the solution to limit the aggregate throughput of application X to 100Mbps. The application X is set to send traffic on each host for 80 seconds with a default rate of 50Mbps. We launch X on 5 hosts, one by one every 10 seconds.

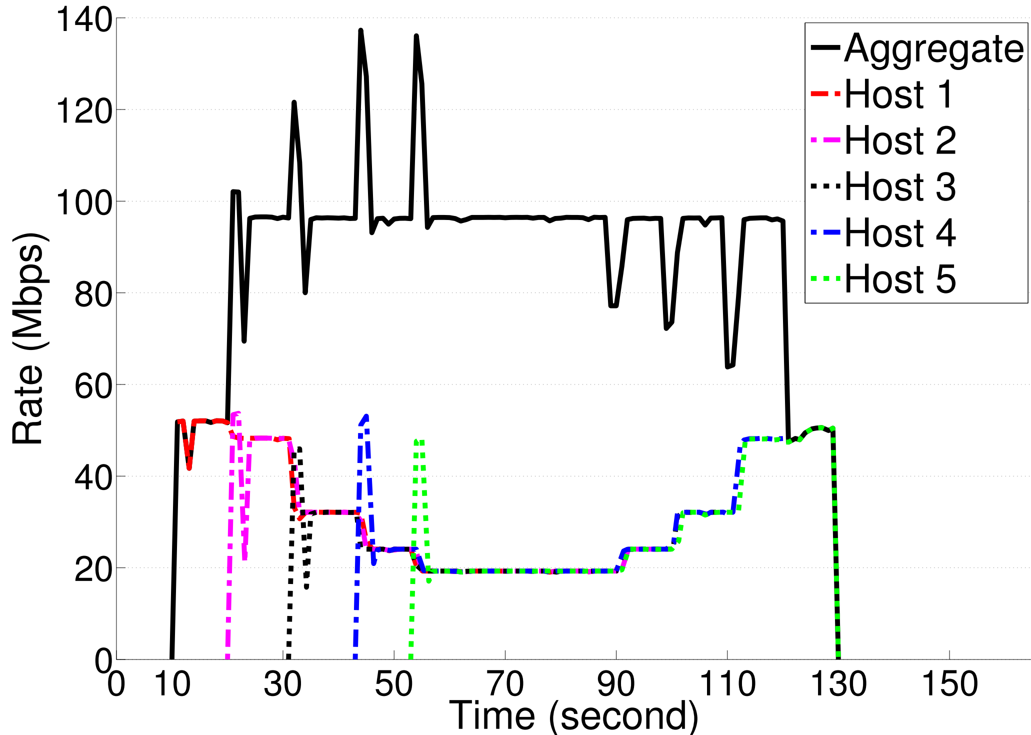

Figure 3.11: Time Series of Application Throughput

Figure 3.11 shows the time series of the aggregate and individual traffic rates of X. The management solution succeeds in limiting the total rate of X running on a distributed set of hosts to 100Mbps. Note that it takes one round of execution for the management solution to discover new traffic and update the rate-limiting policies. That is why there are several 1-second spikes when X starts on new hosts.

3.6 Related Work

Recent projects have sought to incorporate end hosts into network management [59, 84]. But these solutions view the hosts only as software switches or as trusted execution environments for the network. Cooke *et al.* [53] use hosts to collect socket activities for a better understanding of network. However, their solution cannot selectively collect statistics, and the statistics at the transport layer are not covered. OpenTCP [71] dynamically adapts the configuration of TCP on the end hosts based

on traffic conditions. Lee *et al.* [92] propose joint optimization across the application and network layers for better application throughput. In contrast, Hone supports diverse traffic management solutions, and thus provides more measurement, data analysis, and control functionalities across hosts and network devices.

There have also been industry efforts in simplifying cloud management, such as various commercial tools from vendors [4, 24, 32]. They aim at enabling better visualization and infrastructure control at hosts and switches in the cloud. Hone is complementary to these systems by focusing more on monitoring and analysis in traffic management solutions and providing programmable interfaces for these solutions.

Prior work also adopts the stream abstraction for network traffic analysis [42, 54]. But they mainly focus on extending the SQL language, while we use functional language constructs to define traffic management mechanisms more easily. Further, some of these works [42, 106] focus on a specific problem (e.g., intrusion detection) when designing their programming language, while Hone aims for a more generic programming interface for traffic management.

Finally, there are recent works proposing network programming languages [67, 99, 102, 116, 135, 136]. Their programming abstraction is the raw packets or traffic counters on a single network device. Hone mainly moves the programmability to the end hosts, provides an extensible platform for various types of measurement, and spans both the hosts and the network devices.

3.7 Conclusion

Hone is a programmable and scalable platform for joint host-network traffic management. Hone expands the scope of traffic management to the host network stacks, in order to harness the detailed network-related statistics and the computational resources on the hosts. We design an integrated data model for diverse fine-grained

measurement from hosts and network. The programming framework further offers data-parallel streaming operators to define measurement and analysis logic of traffic management solutions. The system can selectively measure the data as needed by the management solutions, and it divides the analysis logic to execute locally on hosts in real time. Thus the Hone system is efficient and scalable for traffic management with integration of host stacks. Micro-benchmarks and experiments with real management solutions demonstrate the performance and expressiveness of our system. We believe that Hone can become an invaluable platform for the research community and cloud providers.

Chapter 4

Sprite: Bridging Enterprise and ISP for Inbound Traffic Control

4.1 Introduction

Many edge networks—like enterprises, university campuses, and broadband access networks—connect to multiple Internet Service Providers (ISPs) for higher reliability, better performance, and lower cost.

Over the past fifteen years, many research projects [36, 37, 72] and commercial products [5, 6, 10, 14, 20, 25] have shown how multihomed networks can divide their *outbound* traffic over multiple ISPs to optimize peformance, load, and cost. However, relatively few works have explored how to perform *inbound* traffic engineering effectively.

Yet, inbound traffic engineering has never been more important due to the growing role of modern applications like video streaming and cloud services. Video streaming has highly asymmetric traffic demands, with the vast majority of the traffic entering the edge network and only small requests and acknowledgments traveling in the other direction. For example, our measurements show that the Princeton University

campus receives an average of *eight times* more traffic than it sends. Receiving this traffic via the right ISP is thus crucial for offering good performance at a reasonable cost. Moreover, video traffic increasingly encounters bottlenecks in the middle of the Internet [51], where some ISPs do not devote enough bandwidth for high-quality video streaming, leading video services to decrease the video quality to reduce the bandwidth requirements. Switching the traffic to a different incoming ISP along a better end-to-end path could increase the video quality.

Many enterprises have outsourced key business applications to the cloud (*e.g.*, Amazon Web Services, Microsoft Azure, and Google App Engine). These enterprises now receive much of their own business traffic over the Internet through their ISPs, rather than purely internally. However, without more control over the flow of traffic, enterprises can experience bad end-to-end performance for these business-critical applications. To ensure good performance, enterprises could connect to the cloud provider over dedicated Virtual Private Network (VPN) [3, 16], rather than the public Internet, but at a much greater cost. In addition, enterprises often outsource the hosting of their public websites to content distribution networks (*e.g.*, Akamai, CloudFlare, and Limelight Networks). As a result, these enterprises have even less outbound traffic, relative to the volume of incoming traffic. Broadband providers face similar challenges, where most traffic goes *to* their customers, and "apps" experience poor performance when the traffic arrives over a bad end-to-end path.

Unfortunately, inbound traffic engineering (TE) is quite difficult. Indeed, Internet routing is destination-based. As such, the sending Autonomous System (AS) decides where to forward traffic, based on the routes announced by its neighbors. The receiving network has, at best, clumsy and indirect control by manipulating its Border Gateway Protocol (BGP) announcements. For example, the edge network can perform *AS-PATH prepending* to make routes through one ISP look artificially longer than another by adding fake hops. However, AS-PATH prepending is hard to

control—adding one extra (fake) hop to the path can dramatically change the division of traffic, or have no effect at all [94]. Alternatively, the edge network can exert direct control with *selective prefix announcements* to force all traffic to one group of users to arrive via one ISP rather than another. Both techniques are coarse-grained and do not allow an edge network to, for instance, receive all Netflix traffic via one ISP and all Gmail traffic via another.

Edge networks need an inbound TE solution that offers: (i) *direct* control over which traffic enters via each ISP; (ii) *fine-grained* control, by sender, receiver, applications or even individual connections; (iii) *local* control without requiring support from the sending network or the rest of the Internet; (iv) *BGP stability*, so that it should not adapt its BGP announcements to achieve its objectives; and (v) *scalability*, so that the system can handle fine-grained TE objectives over a large number of connections and multiple border routers.

In this chapter, we introduce Sprite (Scalable PRogrammable Inbound Traffic Engineering), a software-defined TE solution. Sprite controls inbound traffic by dividing the edge network's public IP address space across the ISPs, and using source network address translation (SNAT) to map each outbound connection to a specific inbound ISP for the return traffic [36], as discussed in more detail in the next section. Given the nature of Internet routing, an edge network cannot fully control the entire end-to-end path—only which entry point receives the traffic. Still, this gives an edge network enough control to balance load and optimize performance by selecting among a small set of end-to-end paths for each connection.

The key contribution of Sprite is a *scalable* solution for realizing *high-level* TE objectives, using software-defined networking (SDN):

- **Scalable data plane through distributed SNAT:** Sprite distributes the SNAT functionality over many (possibly software) switches, close to the end hosts, rather than implementing SNAT at a single proxy or border router.

- **Scalable control plane through local agents:** A local agent at each switch generates the SNAT rules himself, based on a network policy set by the controller, rather than sending the first packet of each connection to the controller.
- **Dynamic adaptation of network policy based on high-level objective:** The network administrator specifies a high-level traffic-engineering objective based on names (rather than network identifiers) and performance metrics (rather than routing decisions). The controller translates the high-level objective into network policy, and dynamically adjusts the network policy based on measurements of performance metrics.

We present the design and implementation of our Sprite architecture, and evaluate our system "in the wild" using an EC2-based testbed with the help of the PEERING testbed [115, 125].

4.2 Inbound TE Using Source NAT

An edge network can directly control inbound traffic by combining two mechanisms:

Split the IP address block across ISPs: To control the flow of inbound traffic, the edge network's public address space is divided into separate prefixes, and Sprite assigns each prefix to one upstream ISP, similar to the common practice of selective prefix announcements. In the example in Figure 4.1, an edge network with the 1.1.0.0/23 address and two ISPs could announce 1.1.0.0/24 via ISP 1 and 1.1.1.0/24 via ISP 2; for fault tolerance in the case of ISP disconnection, the edge network also announces the supernet 1.1.0.0/23 via both ISPs.

Perform SNAT on outbound traffic: To associate a new connection with a particular inbound ISP, the edge network maps the source IP address of outgoing request traffic to an address in the appropriate prefix. In the example of Figure 4.1, the edge network numbers its client machines using private addresses (e.g., in the

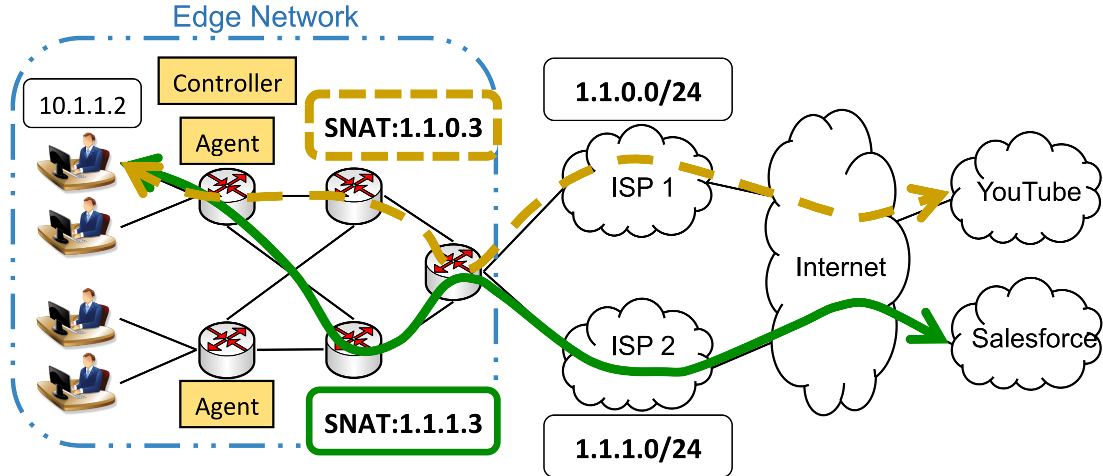

Figure 4.1: Example of How Sprite Works

10.0.0.0/8 address block), and maps outgoing connections to a source IP address in either 1.1.0.0/24 (if the destination corresponds to a YouTube server) or 1.1.1.0/24 (if the destination corresponds to a Salesforce.com server).

While the outbound traffic might leave the edge network through either upstream ISP, these two mechanisms ensure that the response traffic arrives via the selected ISP. If the edge network runs any public services, those hosts should have public IP addresses routable via either ISP.

Using SNAT for inbound traffic engineering is not a new idea. Previous work [36, Sec IV.C] briefly discusses how to realize NAT-based inbound route control at a Web proxy using `iptables`. The main challenges we address in this paper are (i) automatically translating fine-grained TE objectives to low-level rules and (ii) distributing both the control-plane and data-plane functionality for better scalability.

4.3 Scalable Sprite Architecture

Sprite achieves scalability by distributing the data-plane rules (across switches near the end hosts) and control-plane operations (across local agents on or near the switches), as Figure 4.3 shows the architecture of Sprite. The network administrator

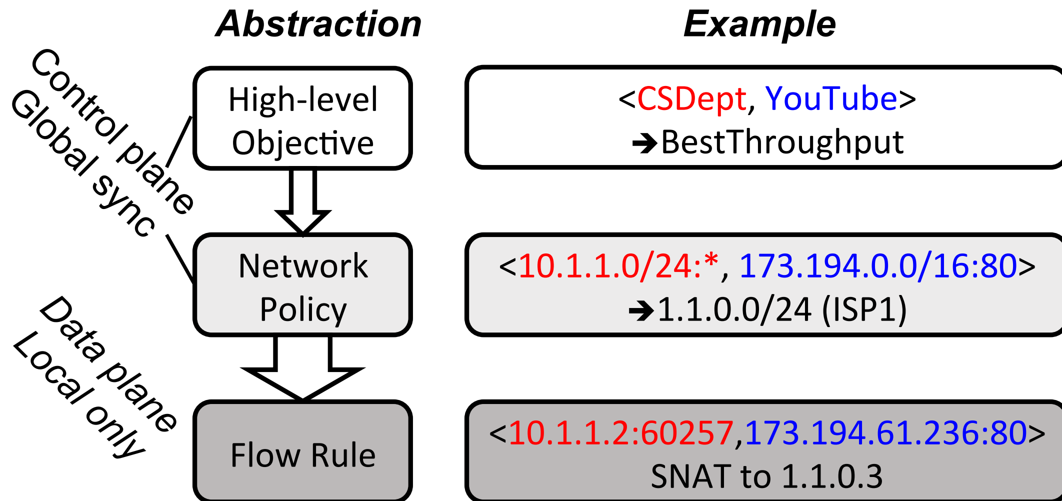

Figure 4.2: Three Levels of Abstraction in Sprite

conveys a high-level traffic-engineering objective to the controller, then the controller generates a set of network policies to distribute to the local agents, and finally the local agents install SNAT rules in the switch data plane, as summarized in Figure 4.2.

4.3.1 Data Plane: Edge Switches Near Hosts

SNAT gives edge networks direct, fine-grained control over inbound traffic, at the expense of scalability. SNAT requires dynamically establishing a data-plane rule for each connection. Performing SNAT at a single proxy or border router would require data-plane state in proportion to the number of active connections, as well as control-plane operations on every connection set-up. The presence of multiple border routers introduces further complexity, since traffic for the same connection may enter and leave via different locations.

Instead, Sprite performs SNAT on a distributed collection of switches, near the end hosts. These switches can also collect passive traffic measurements (e.g., byte and packet counts per rule) that can help drive traffic-engineering decisions. These edge switches could be virtual switches running on the end hosts, access switches connected directly to a group of end hosts, or a gateway switch connecting a department to the

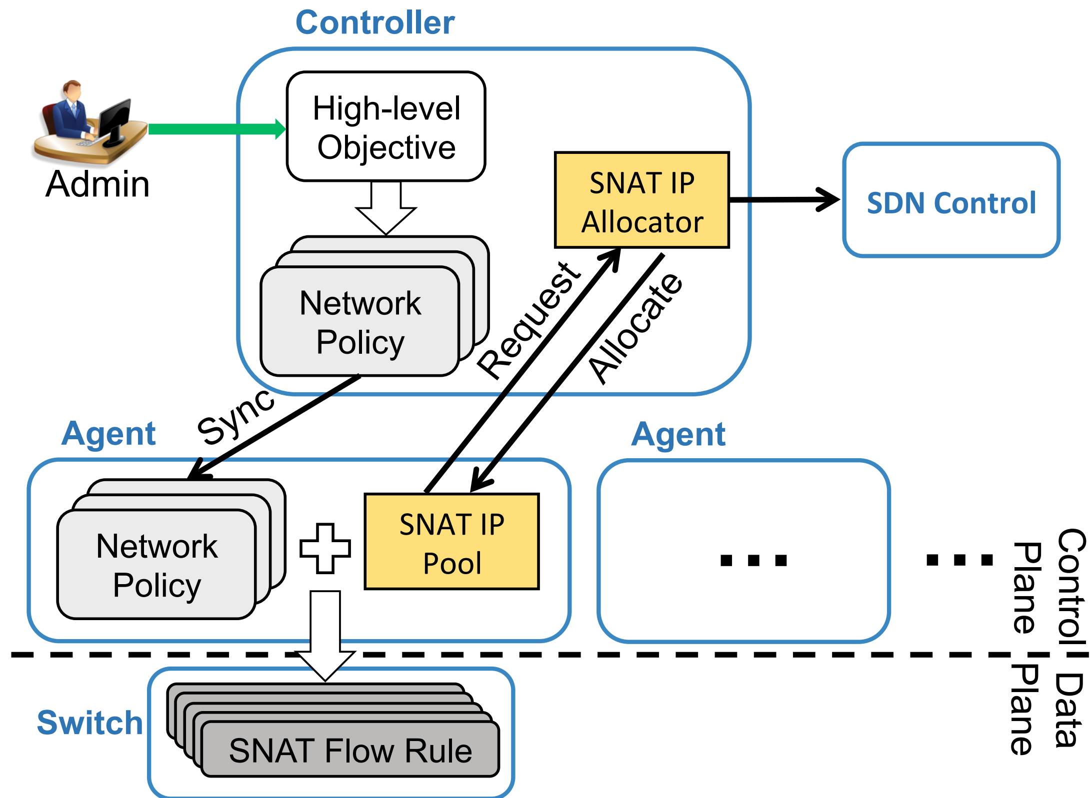

Figure 4.3: Sprite System Components

rest of the enterprise network. Compared to the border routers, each switch handles much fewer active connections and a much lower arrival rate of new connections, enabling the use of commodity switches (with small rule tables) or software switches.

The edge switches also need to receive the return traffic destined to its associated end hosts. Each edge switch maps outbound traffic from internal private addresses to a small set of public IP addresses assigned by the controller to the local agent. As such, the return traffic is easily routed to the right edge switch based on the destination IP prefix. If the edge network runs a legacy routing protocol (e.g., OSPF or IS-IS), the controller configures the injection of these prefixes into the routing protocol. If the edge network consists of SDN switches, the controller can install coarse-grained rules in the border router and interior switches to forward traffic to the right edge switch. By carefully assigning contiguous prefixes to nearby edge switches, the controller could further aggregate these prefixes in the rest of the edge network.

4.3.2 Control Plane: Local Agents Near Switches

Rather than involving the controller in installing each SNAT rule, a local agent on (or near) each switch performs this simple task. Based on a high-level traffic-engineering objective, the controller computes a network policy that maps network identifiers (e.g., IP addresses and port ranges) to the appropriate inbound ISP. Then, the controller subdivides the unique set of source IP addresses and port ranges across the local agents, so each local agent can generate flow rules on its own as new connections arrive. As a result, Sprite does not send *any* data packets to the controller, and ensures that all traffic follows an efficient path between the end hosts and the border routers. The local agent can also collect and aggregate measurement data from the switch's rule counter and via active probing, and share the results with the controller to inform future decisions.

The network policy generated by the Sprite controller only specifies source and destination IP prefixes, TCP/UDP port ranges, and which inbound ISP to use. Each local agent uses the network policy to automatically create SNAT rules for each new flow. Each edge switch has a default rule that sends all outbound traffic (i.e., packets with an internal source IP address) to the associated local agent. Upon receiving the packet, the local agent consults the current network policy to identify the suitable set of public addresses and port numbers, selects a single unused address/port pair, and installs the NAT rules for both directions of the traffic.

The controller needs to assign each local agent a sufficiently large pool of addresses and port numbers. When new flow starts, the agent can immediately use one free source IP from its own pool without requesting the controller and pausing the new flow. Yet the limited public address space shall not be wasted. Thus, the allocation algorithm of Sprite follows two rules: *1)* the number of IPs and ports allocated for an agent shall slightly exceed the actual number of connections traversing the agent; and

2) when network policy changes, we shall limit the changes of ownership of source IPs to minimize the impact to ongoing connections.

The controller uses measurement data collected from the local agent to track statistics on the number of simultaneously active connections. When running low on available address/port pairs, the agent contacts the controller to request additional identifiers, e.g., when the actual number of SNATed connections exceeds 90% of the upper limit of currently allocated IPs and port ranges. Similarly, the controller can reclaim unused ranges of addresses from one local agent and assign them to another as needed. With reasonable "headroom" to over-allocate address/port pairs to each agent, the controller can limit the churn in the assignments.

4.4 Dynamic Policy Adaptation

Sprite enables network administrators to express a wide range of TE objectives using high-level names of remote services and groups of users, as well as performance metrics. The controller automatically translates the TE objective into a set of network policies, and adapts in real time to traffic and performance measurements.

4.4.1 High-level Traffic Engineering Objectives

Rather than specifying TE objectives on IP addresses and port numbers, Sprite allows administrators to use high-level names to identify services (e.g., YouTube) and groups of users (e.g., CSDept). The administrators can let Sprite dynamically map the connections of the users/services onto the ISPs by providing an evaluation function. The function takes many metrics (e.g., the ISP capacity, the connections' performance, etc.) as inputs, and returns a score for how the ISP behaves. For example,

$$\text{USER}(\textit{BioDept}) \text{ AND SERVICE}(\textit{Salesforce.com}) \rightarrow$$
$$\text{DYNAMIC}(\textit{LatencyCalculationFunction})$$

OBJECTIVE	:=	PREDICATE → ISP_CHOICE
PREDICATE	:=	USER(*user identifier*)
		| SERVICE(*remote service name*)
		| PREDICATE AND/OR PREDICATE
ISP_CHOICE	:=	DYNAMIC(*evaluation function*)
		| STATIC([<*ISP identifier, weight*>, ...])
evaluation function	:=	*User-defined function over ISP capacity, connection statistics, and other metrics*

Table 4.1: Syntax of High-level Objective

specifies that traffic from Salesforce.com to the Biology Department should use the ISP that offers the lowest latency. Alternatively, the administrator can associate connections with particular named users and services with a specific set of ISPs (with weights that specify the portion of inbound traffic volume by ISP). For example,

SERVICE(*YouTube*) → STATIC([<*ISP1,1.0*>, <*ISP2,4.0*>, <*ISP3,9.0*>])

specified that YouTube traffic should enter via ISPs 1, 2, and 3 in a 1:4:9 ratio by traffic volume. We summarize the syntax of the language in Table 4.1.

4.4.2 Computing Network Policy

Sprite collects performance metrics of the network policy and uses inputs from the edge network itself to automatically adapt the set of network policies for a high-level objective. Figure 4.4 illustrates the workflow of network policy adaptation.

Mapping names to identifiers: Sprite maintains two data sources to map the high-level name of a service or a group of users to the corresponding IP addresses. For users, Sprite combines the data from the device registry database of the edge network (linking device MAC addresses to users) and the DHCP records to track the mappings of <user ID, list of owned IPs>. The <user group, list of users> records are provided by the network administrators manually. For external services, Sprite tracks the set of IP addresses hosting them. Like NetAssay [60], Sprite combines three sources of

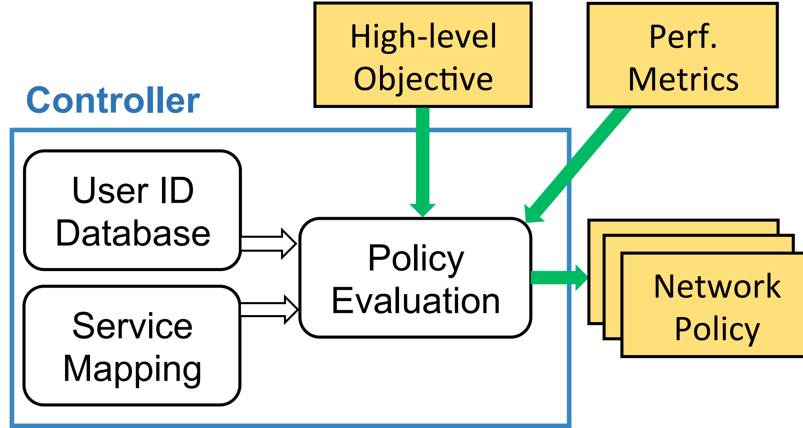

Figure 4.4: Workflow of Network Policy Adaptation

data to automatically map a service's name to the prefixes it uses to send traffic: *1)* the DNS records obtained from the edge network; *2)* the BGP announcements at the border routers; and *3)* traces coming from a few probe machines that emulate user traffic to popular services. Although Sprite cannot guarantee 100% accuracy, it can discover the prefixes for a majority of the service's inbound volume in practice[1].

Satisfying the TE objective: The Sprite controller translates the high-level objective into a set of clauses for the network policy, expressed as <*user prefix: port range, service prefix: port range*> → *inbound ISP*. For each network policy, the Sprite agent collects the performance metrics of each matching connection, from the counters of SNAT rules in the data plane (e.g., throughput) to richer transport-layer statistics (e.g., round-trip time, size of socket buffer, TCP congestion window size) [123]. The controller collects these metrics periodically, and calculates the aggregate performance. Then the data are fed to the evaluation function provided by the administrators to score how each ISP behaves. If the scores of the ISPs are different enough, the controller adapts the network policy by swapping some users from one inbound ISP to another. Sprite always keeps at least one active user on an ISP so that it can

[1]One reason is that the major contributors of inbound traffic (e.g., Netflix and YouTube) are increasingly using their own content delivery networks (CDNs) [11, 19], rather than commercial CDNs. These services' own CDNs usually sit in their own ASes.

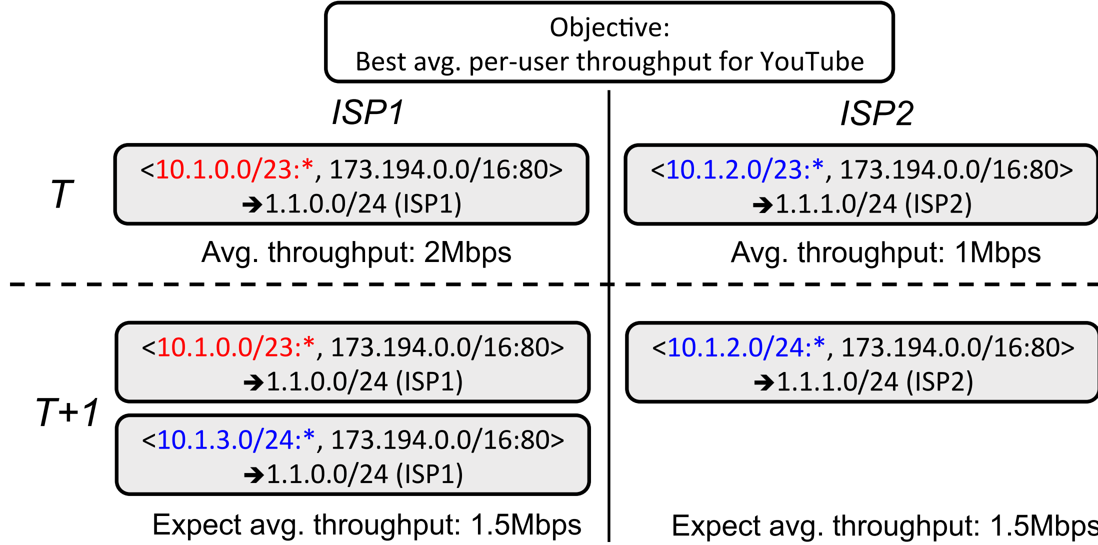

Figure 4.5: Network Policy Adaptation for Dynamic Perf-driven Balancing

always know the actual performance of inbound traffic via an ISP through passive measurement of real traffic.

We now illustrate the process through an example in Figure 4.5. Suppose the objective is to achieve the maximum average throughput for YouTube clients. Users in the edge network are in the 10.1.0.0/22 address block. The Sprite controller initially splits the users into two groups (10.1.0.0/23, 10.1.2.0/23), and allocates their traffic with YouTube to use one of the two ISPs. Figure 4.5 shows the network policies generated in the iteration *T*. Carrying out the network policies, Sprite measures the throughput of each SNATed connection with YouTube, and calculates the average per-user throughput. The average inbound throughput via ISP2 is 1Mbps due to high congestion, while that of ISP1 is 2Mbps. Thus the controller decides to adapt the set of network policies to move some users from ISP2 to ISP1. In the iteration *T+1*, the users in 10.1.2.0/23 are further split into two smaller groups: 10.1.2.0/24 and 10.1.3.0/24. While users in 10.1.2.0/24 stay with ISP2, users in 10.1.3.0/24 have their new connections use ISP1 for their traffic from YouTube. The new set of network policies should alleviate congestion on ISP2 and might increase congestion on ISP1, leading to further adjustments in the future.

4.5 Implementation

In this section, we describe the design and implementation of the Sprite system and how we made it efficient and robust.

4.5.1 Design for Fault Tolerance

Sprite system centers on a distributed datastore (see Figure 4.6[2]), which keeps all the stateful information related to the high-level objective, the network policy, the performance metrics of SNATed connections, and the status of SNAT IP allocation. The controller and all the agents run independently in a stateless fashion. They never directly communicate with each other, and just read or write data through the distributed datastore.

Making the datastore the single stateful place in Sprite greatly improves the system's robustness. Indeed, device failures are common since Sprite employs a distributed set of agents and commodity switches. In this architecture, any controller or agent failure won't affect the operations of other instances or the stability of the whole system. Recovery from failures also becomes a simple task. We can start a fresh instance of controller or agent to catch up states from the datastore to resume the dropped operations. Taking switches offline for maintenance is also tolerable as we can freeze the datastore correspondingly to stop the operation of affected agents.

The architecture also makes the Sprite system very flexible for different deployment environments. For instance, some enterprises may have standard imaging for all machines, and wish to bundle the Sprite agent in the image to run directly on the end host, while others can only place the agent side by side with the gateway routers. The adopters of Sprite can plug in/out or re-implement their own controller or agent

[2]Not shown in Figure 4.6, we use Floodlight controller as our SDN control module, and it only communicates with the controller for insertion and deletion of routing rules.

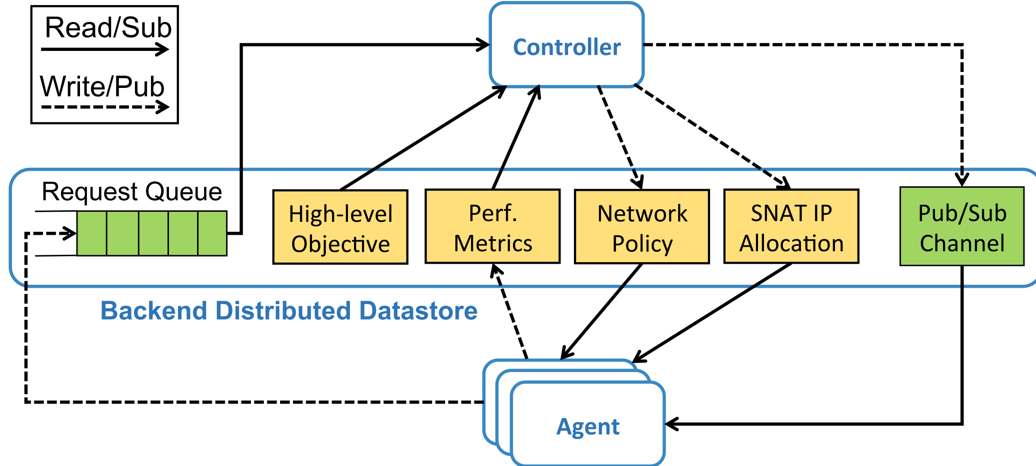

Figure 4.6: System Architecture of Sprite Implementation

to accommodate the deployment constraints, as long as maintaining the read/write interface with the datastore.

The implementation of the distributed datastore depends on our data model. The model of network policy involves the mapping of the four-tuple prefix/port wildcard match and the inbound ISP. The SNAT IP allocation is the mapping among IP, ISP, allocation state, and agent. Using multiple items as the *keys*, the row-oriented, multi-column-index data structure of Cassandra is the best fit. Thus, we use Cassandra as the datastore of Sprite.

4.5.2 How Components Communicate

The controller and agents of Sprite interact via the datastore in a pull-based fashion. However, the pull-based approach slows Sprite in two places. Firstly, when the controller adapts the network policies, a pull-based agent may take up to its run period to pick up the new set of network policies. This significantly drags the convergence speed of carrying out the new policy throughout the edge network, thus slowing the convergence of the policy adaptation. A second issue with the pull-based approach happens in the allocation process of SNAT IPs. When agents request the allocation of new source IPs and port ranges, new connections of users may be halted at the

agent. A long wait time would trigger the connection to drop, thus affecting the user's performance.

We need to add push-based communication method to balance the robustness and performance of Sprite. Thus, we add two communication points in the datastore for push-based signaling between controller and agents: a publish/subscribe channel and a message queue, as shown in Figure 4.6. The signaling works as shown below:

- **Network policy adaptation:** When the controller writes new network policies or new SNAT IP allocation into the datastore, the controller publishes notification via the pub/sub channel. As all agents subscribe to the channel upon startup, the notification triggers them to refresh the data from the datastore, thus catching up with the new policy or allocation state quickly. Also, whenever a new controller instance starts, it will publish notification upon finishing bootstrap, in case that the old controller instance has failed after writing into the datastore, yet before publishing the notification.
- **SNAT IP allocation:** The message queue keeps the agents' allocation requests at its tail, and the controller only removes the head once it successfully handles the request and updates the datastore. In this way, the message queue guarantees that each request is handled at least once. Thus users' connections are less likely to be stuck at the agents due to lack of source IPs. The effects of possibly handing one request more than once are offset by the reclamation of the controller. This mechanism also tolerates agent failures that a rebooted agent instance can read the allocation results from the datastore without re-submitting a request.

4.5.3 Routing Control for Returning Packets

When we design to scale up the control plane of Sprite, we decide not to synchronize the SNAT states of active connections. These states are kept only locally at each agent/switch. As a result, the returning packets destined for the SNATed IP must

arrive at the agent which handles the translation in the first place, in order to reverse the translation correctly.

Assuming an OpenFlow-enabled network in our implementation, Sprite installs routing rules to direct the returning packets to the right agents, i.e., once a source IP/port range is allocated to an agent, the controller installs OpenFlow rules to match the source IP/port range along the switches from the border router to the agent.

Rather than simply installing one rule per allocated source IP in switches, we try to consolidate the routing rules into matching a bigger prefix block to collapse many rules into one. Our current algorithm works in this way: we construct the shorted-path tree rooted at the border router with all agents as the leaves. When allocating a source IP to an agent, we pick the one that is bit-wise closest to the IPs allocated to the agents having the longest shared paths. We leave the improvement of the current algorithm to future efforts.

4.5.4 BGP Stability

Sprite splits the edge network's address space to announce separately via different ISPs. An alternative would be using the peering IPs with ISPs for SNAT. Compared to the alternative, our splitting technique risks inflating global routing tables. However, we argue that the technique offers more benefits than drawbacks.

Our approach ensures global reachability, while benefiting from the robustness of BGP. The separately announced IP blocks belong to the edge network. The neighboring ISPs must advertise them to further upstream ISPs, while the peering IPs are typically private. The edge networks also enjoy the automatic failover brought by BGP, since Sprite announces the supernet to all ISPs. In case of ISP-level disconnection, inbound traffic can move to other ISPs automatically, instead of being lost if using the peering IPs.

Sprite can also have higher capacity for SNATing connections. For example, the announcement of a /24 block gives Sprite an upper limit of about 14 million connections ($256 \times (65535 - 10000)$) to SNAT. Using the peering IPs only yields a 55-thousand capacity ($65535 - 10000$), which is way not enough for a large-size edge network.

Finally, the splitting technique is already widely used in practice. Adopting this approach in Sprite requires the least cooperation from the upstream ISPs, thus being the most deployable option.

4.6 Evaluation

We collected traffic data from the campus network of Princeton University to understand the traffic patterns of multi-homed enterprise networks. We then evaluate Sprite with a pilot deployment on an EC2-based testbed to demonstrate how Sprite achieves TE objectives.

4.6.1 Princeton Campus Network Data

The campus network of Princeton University is a multihomed site with three upstream ISPs (Cogent, Windstream, and Magpie). The ISPs are contracted to provide 3Gbps, 2Gbps, and 1Gbps respectively. In recent years, the campus network receives rapidly growing inbound traffic mainly from video streaming services, and the university is consolidating the departmental computation services into one university-wide service hosted in a remote, newly built datacenter.

We want to study how the traffic pattern changes under the trends and how the upstream ISPs are utilized for the inbound traffic. We have collected Netflow data on the border router of the campus [3]. The Netflow data spreads two weeks long in

[3] For privacy concern, we have anonymized the Netflow data at the time of collection. For every IP address that belongs to Princeton University, we create a unique yet random map to an address

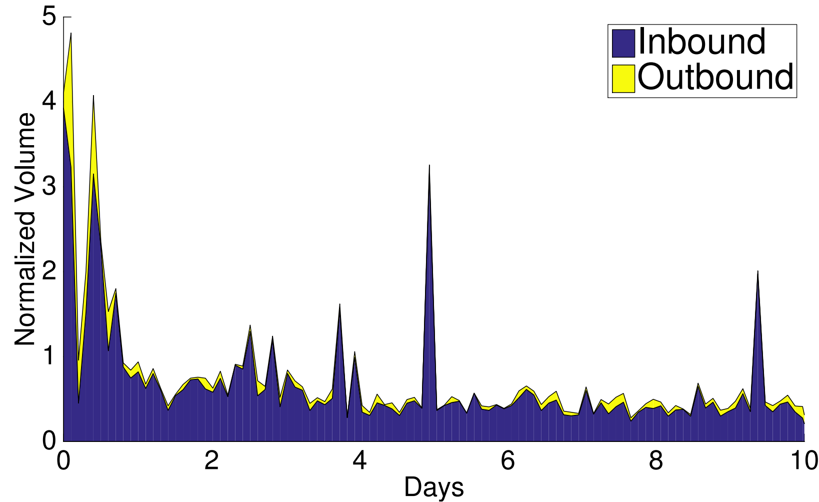

Figure 4.7: Stacked Chart of Inbound and Outbound Traffic Volume

	Windstream	Magpie	Cogent
Percentage	32.2%	8.0%	59.8%

Table 4.2: Total Inbound Volume Distribution among ISPs

December 2014. Each Netflow record identifies a single connection with many traffic statistics (e.g., number of packets/bytes). Matching with the physical configuration of the border router, we can study which ISPs carries each connection.

Figure 4.7 is the stacked area chart showing the volume of inbound and outbound traffic over time [4]. The inbound traffic is always dominant, averaging 89.7% of the total volume. Further delving into the inbound traffic pattern, we show the stacked chart of the inbound traffic carried by the three ISPs in Figure 4.8. We also aggregate the total volume via each ISP of the collection window, and show the traffic proportion in Table 4.2. The results show that the campus network currently uses ISP Cogent and ISP Windstream as the main carriers, and splits the traffic roughly 2:1 regardless of users or services. Remember that the contracted bandwidth of Cogent, Windstream, and Magpie is 3:2:1. It means Windstream and Magpie are usually underutilized.

in the 10.0.0.0/8 block, and modify the Netflow records correspondingly. We leave non-Princeton IP addresses intact.

[4]We have normalized the traffic volume of the campus network for privacy concerns.

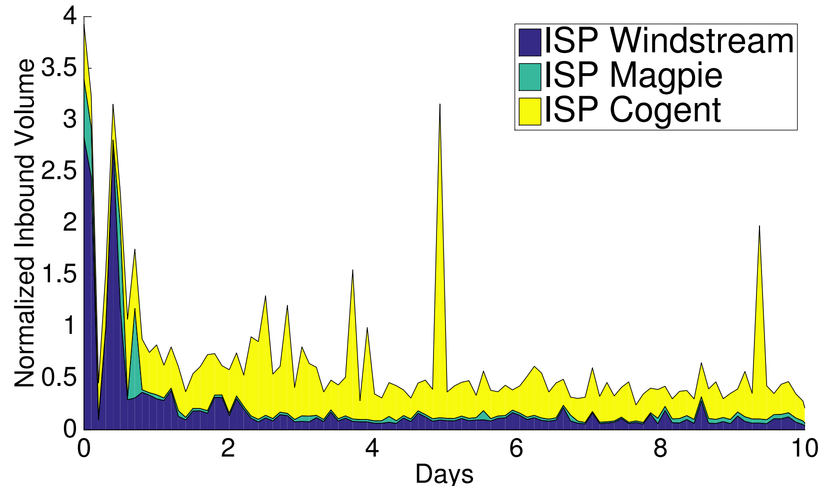

Figure 4.8: Stacked Chart of Inbound Traffic via Three ISPs

4.6.2 Multi-ISP Deployment Setup

We build a testbed in AWS Virtual Private Cloud (VPC) to emulate an enterprise network with multiple upstream ISPs, with the help of the PEERING testbed. The PEERING testbed is a multi-university collaboration platform which allows us to use each participating university as an ISP. Our VPC testbed connects with two PEERING sites to emulate a two-ISP enterprise network.

Figure 4.9 shows the testbed setup. In the AWS VPC, we launch one machine (i.e., an AWS EC2 instance) to function as the border router. The border-router instance runs Quagga software router to establish BGP sessions with the PEERING sites in Georgia Tech and Clemson University. For each PEERING site, we have one /24 globally routable block to use.

Behind the border-router instance, we launch many EC2 instances to function as the "user" machines. These user-machine instances connect with the border-router instance via regular VPN tunnels to create a star topology. On each user-machine instance, we run the Sprite agent and OpenVSwitch. The Sprite agents uses `iptables` and OpenVSwitch to monitor and SNAT the connections. We will launch applications (e.g., YouTube) from the user-machine instances to emulate the traffic.

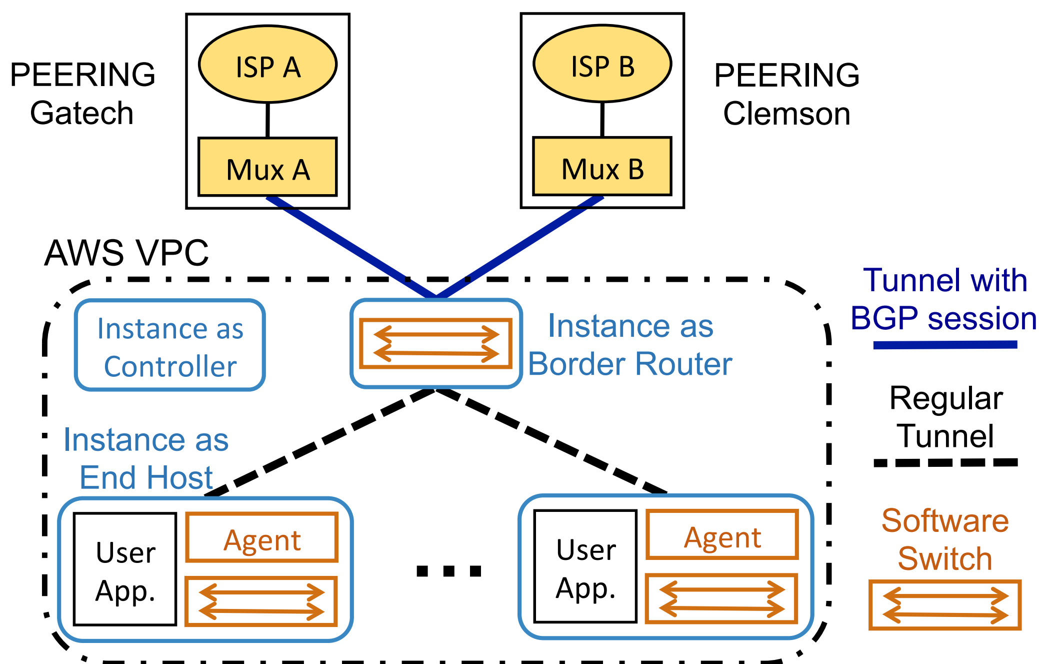

Figure 4.9: Setup of the Multihomed Testbed on AWS VPC

4.6.3 Inbound-ISP Performance Variance

ISPs perform differently when delivering the same service to the edge networks, e.g., YouTube and Netflix. The performance difference among ISPs can be caused by various reasons [51]. An example is the recent dispute between Netflix/Cogent and Verizon. The video quality of Netflix is bad when delivered by Verizon, due to the limited capacity of the peering links between Verizon and Netflix. In contrast, Cogent does not have the quality issue as its peering links have higher capacity.

Using Sprite, we can prove that different ISPs provide different quality towards the same service by specifying an objective of equally splitting the users to use one of the two ISPs. On all user machines, we launch YouTube for a 2-hour-long movie, and we explicitly set the users to stream the movie from the same YouTube access point. In the process, we measure the video quality of the video every 1 minute on every machine. Figure 4.10 shows the histogram of all these quality measurement points to examine the characteristics of the two ISPs for streaming YouTube. The Gatech PEERING site consistently delivers video of higher quality than the Clemson site.

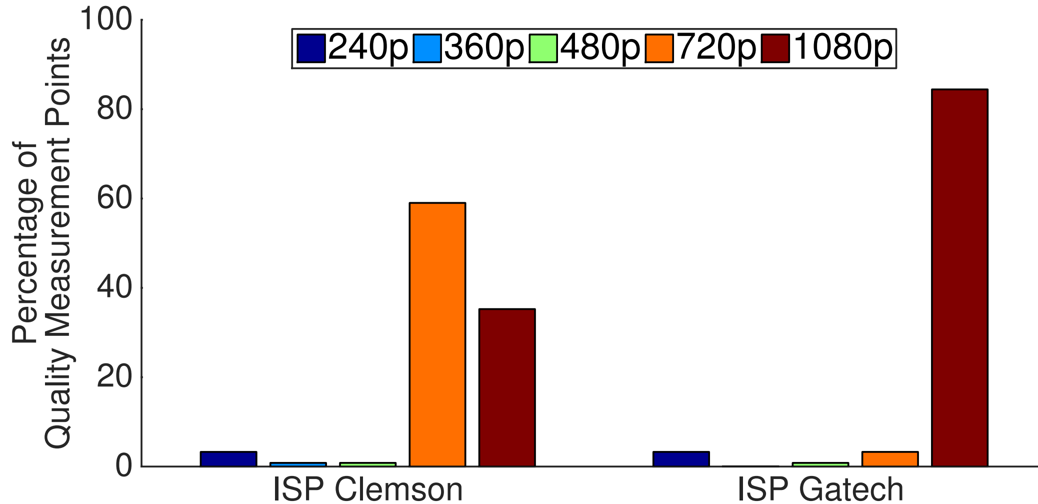

Figure 4.10: Histogram of Video Quality via Two ISPs

4.6.4 Effects of Dynamic Balancing

Sprite can dynamically move traffic among ISPs to achieve the TE objective specified by the administrators. We provide an objective to achieve best average per-user throughput for YouTube traffic, and evaluate how Sprite adapts the network policies for such an objective. The objective is expressed as:

$$\text{SERVICE}(YouTube) \rightarrow \text{BEST}(AvgIndividualThroughput)$$

The experiment runs on the VPC-based testbed. We launch YouTube on 10 user machines. We want to examine how the traffic of users moves from one ISP to another over the time, and whether Sprite can keep the average per-user throughput roughly the same (within 5% margin) between the two ISPs. To evaluate how Sprite reacts, we manually limit the capacity of the tunnel with the Gatech PEERING site to emulate high congestion on the link. Figure 4.11 shows the time series of the average per-user throughput of accessing YouTube on these two ISPs. The average throughput of two ISPs are always kept in line.

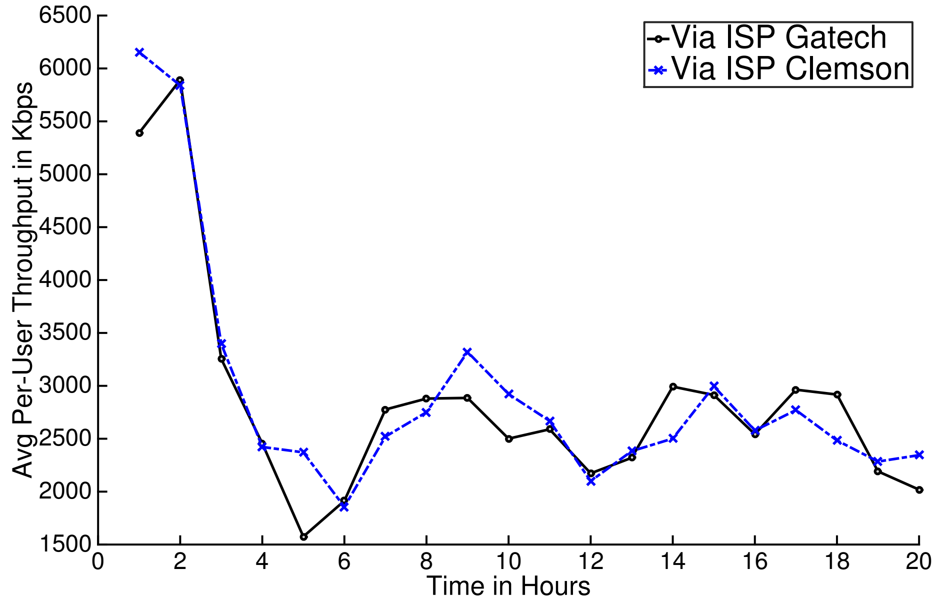

Figure 4.11: Time Series of Average Per-User Throughput of YouTube

4.7 Related Work

Many works have considered aspects of the problem we address, without providing a complete solution for direct, fine-grained, incrementally-deployable inbound TE.

BGP-based approaches: Studying the impact of tuning BGP configuration to an AS's incoming traffic has a long and rich history spanning over a decade [47, 50, 63, 69, 111, 112, 113, 129], including numerous proprietary solutions [6, 12, 21]. All these solutions suffer from at least three problems. First, they are *non-deterministic*. They can only indirectly influence remote decisions but cannot control them, forcing operators to rely on trial-and-error. Second, they are too coarse-grained as they only work at the level of a destination IP prefix. Third, they often increase the amount of Internet-wide resources (e.g., routing table size, convergence, churn) required to route traffic, for the benefit of a single AS. In contrast, Sprite provides direct and fine-grained control (at the level of a user or service) without increasing Internet resources.

Clean-slate approaches: Given the inherent problems with BGP, many works have looked at re-architecting the Internet to enable better control over the forwarding paths. Those works can be classified as *network-based* [62, 65, 119], which modify the way the routers select paths, and *host-based* approaches which do the opposite [40, 56, 100, 105]. While these solutions can offer a principled solution to the problem of inbound traffic engineering, they all suffer from incremental deployment challenges. In contrast, any individual AS can deploy Sprite on its own, right now, and reap the benefits of fine-grained inbound traffic engineering.

4.8 Conclusion

In this chapter, we study how to control the inbound traffic of cloud services for the edge networks with multiple upstream ISPs. Our proposal, called Sprite, enables edge networks to have direct and fine-grained control of their inbound traffic with a scalable system solely residing inside the edge networks. Sprite also provides simple and high-level interfaces to easily express traffic engineering objectives, and Sprite dynamically adapts the objectives into low-level policies to enforce throughout the edge networks. We have tested Sprite with live Internet experiments on the PEERING testbed, and we plan to conduct more extensive experiments with possible deployment on the campus network of Princeton University.

Chapter 5

Conclusion

The ongoing trend of adopting cloud computing raises the requirements for the quality of the end-to-end networks. Building proper network management solutions is the key factor in improving the efficiency and reliability of networks. This dissertation focuses on solving two main problems of current network management: *1)* The management systems of network components are disjoint, e.g., servers, routing on network devices, device hardware configurations, etc. As the responsibilities of managing various components fall on the shoulders of cloud service providers, the separation becomes bottleneck in building better management solutions; *2)* Network management heavily relies on the vendor-specific interfaces with devices. It not only binds management solutions to hardware features, but also becomes overcomplicated as datacenters grow in scale with commodity devices from multiple vendors.

This dissertation takes a practical approach to carefully balance the research exploration in solving the two problems and the engineering efforts in impacting commercial cloud services. Closely working with major cloud providers, we identify real-world opportunities for integrating different management components with proper high-level abstraction. We then design and build safe, efficient, and scalable integrated management systems, deploying them in datacenters of cloud providers and enterprise

networks that use cloud-based applications. In this chapter, we first summarize the contributions of this dissertation in §5.1. We then briefly discuss some open issues and future directions on our works in §5.2, and conclude in §5.3.

5.1 Summary of Contributions

This dissertation identified three areas of network management in need of integrating different components, and presented corresponding abstraction design and system solutions.

We first built a management platform for cloud providers to consolidate traffic and infrastructure management in datacenters. In this platform, named Statesman, we designed a network-state abstraction to provide a uniform data model for interacting with various aspects of network devices. Offering three distinct views of network state as the workflow pipeline, Statesman could run many traffic and infrastructure management solutions simultaneously, resolving their conflicts and preventing network-wide failures in datacenters. We deployed Statesman in Microsoft Azure worldwide, making it a foundation layer of Azure networking. We also published the work in ACM SIGCOMM 2014.

Second, we identified the opportunity for bringing end hosts into datacenter traffic management. Our solution, named Hone, integrated end hosts and network devices with a uniform data model, and empowered traffic management solutions to utilize the rich application-traffic statistics in the end hosts. Adopted by Verizon Business Cloud, Hone improved the performance of cloud-based applications by improving the quality of connections between customers and Verizon's datacenters. The work was published in Springer Journal of Network and Systems Management, volume 23, 2015.

Finally, we bridged edge networks and their upstream ISPs to provide the edge networks with direct and fine-grained control of their inbound traffic from cloud appli-

cations. Our Sprite system provided simple and high-level interface to easily express traffic engineering objectives, and Sprite executed the objectives with an efficient and scalable system. We tested Sprite with live Internet experiments on the PEERING testbed, and the work was published in ACM SIGCOMM Symposium on SDN Research 2015.

Collectively, the contributions in this dissertation provide system solutions for managing networks along the end-to-end path of cloud computing services. These works have explored how to integrate various disjoint management components to simplify and enhance network management solutions.

5.2 Open Issues and Future Works

The works presented in this dissertation raised a number of open questions that deserve future investigation.

5.2.1 Combining Statesman and Hone in Datacenters

Statesman consolidates traffic and infrastructure management on network devices, and Hone joins end hosts with the routing control on network devices. We believe it is a promising direction to integrate the measurement and control functions of end hosts (as provided by Hone) into the framework of Statesman. In this way, servers and network devices could be managed on a single platform by cloud providers. Yet there are still several challenges: *1)* how to adapt the network-state abstraction to capture the rich data and functionalities of servers; *2)* how to expand the dependency model to correctly capture the relationship between server-side and network-side states; and *3)* how to correctly capture the server availability requirements in the safety invariants checked by Statesman. How to solve these challenges merits further investigation.

5.2.2 Supporting Transactional Semantics in Statesman

The current conflict-resolution mechanism in Statesman does not provide any guarantees as to how the proposed network changes from management solutions are accepted or denied. One could imagine building transactional semantics on top of Statesman. One possible direction to explore is to provide *grouping* semantics that some of the proposed network changes are grouped together for being accepted or denied as a whole. This could guarantee that Statesman either executes all grouped changes together or none at all. Another possible direction is to provide *condition* semantics that specifies the conditions when a proposed change shall be accepted, e.g., a proposal of moving traffic onto device A shall only be accepted if device A is healthy. We currently do not support these advanced mechanisms in Statesman, because the current simple mechanism is sufficient for our operational management solutions. Identifying what transactional semantics are actually necessary and building them into Statesman is a promising venue for future research.

5.2.3 Hone for Multi-tenant Cloud Environment

Hone collects the fine-grained traffic statistics from inside the end hosts, assuming that the cloud providers have access to the hosts' operating systems. In a multi-tenant public cloud, tenants may not want the cloud providers to access the guest OS of the virtual machines. A viable alternative would be to collect measurement data from the hypervisor and infer the transport-layer statistics of the applications in the virtual machines. This direction is currently under exploration [70], and can complement Hone to support more types of cloud environments.

5.3 Concluding Remarks

This dissertation has *1)* presented a new datacenter network management platform that simplifies both traffic and infrastructure management and allows many management solutions to run with no conflicts and network-wide failures; *2)* designed and built a traffic management system for cloud providers to utilize the measurement and control functions of both end hosts and network devices; *3)* developed a scalable system for edge networks to directly control which ISPs shall carry their inbound traffic from cloud applications.

At a high level, the works presented in this dissertation are motivated by practical challenges in network operation of cloud computing services, and we solved the challenges by leveraging evolving technology in our field (e.g., SDN) and knowledge from other fields (e.g., software engineering, distributed storage system, etc.). We believe that, in the networking research area, it will remain an effective research approach to keep close with industry practices, identify and abstract their challenges as research problems, and apply emerging technologies to solve the problems.

Bibliography

[1] Amazon Web Services. http://aws.amazon.com/.

[2] Amazon Web Services Elastic Load Balancing. http://aws.amazon.com/elasticloadbalancing/.

[3] AWS Virtual Private Gateway. http://aws.amazon.com/vpc/.

[4] Boundary. http://www.boundary.com/.

[5] Cisco Optimized Edge Routing (OER). http://www.cisco.com/en/US/tech/tk1335/tsd_technology_support_sub-protocol_home.html.

[6] Cisco Systems Performance Routing (PfR). http://www.cisco.com/c/en/us/products/ios-nx-os-software/performance-routing-pfr/index.html.

[7] Cisco Visual Networking Index Forecast 2013-2018. http://www.cisco.com/c/en/us/solutions/collateral/service-provider/ip-ngn-ip-next-generation-network/white_paper_c11-481360.pdf.

[8] Details of the February 22nd 2013 Windows Azure Storage Disruption. http://azure.microsoft.com/blog/2013/03/01/details-of-the-february-22nd-2013-windows-azure-storage-disruption/.

[9] Floodlight OpenFlow Controller. http://floodlight.openflowhub.org/.

[10] Google chooses RouteScience Internet technology. http://www.computerweekly.com/news/2240046663/Google-chooses-RouteScience-Internet-technology.

[11] Google Global Caching. http://peering.google.com/about/ggc.html.

[12] Internap. Managed Internet Route Optimizer (MIRO). http://www.internap.com/network-services/ip-services/miro/.

[13] Linux Advanced Routing & Traffic Control. http://www.lartc.org/.

[14] Managed Internet Route Optimizer (MIRO). http://www.internap.com/network-services/ip-services/miro/.

[15] Microsoft Azure. http://azure.microsoft.com/.

[16] Microsoft Azure ExpressRoute. http://azure.microsoft.com/en-us/services/expressroute/.

[17] Mobility and Networking Researchers Making a Big Impact in the Cloud. http://research.microsoft.com/en-us/news/features/sigcomm14-081814.aspx.

[18] Netfilter.org. http://www.netfilter.org/.

[19] Netflix Open Connect. http://openconnect.itp.netflix.com/.

[20] netVmg's Flow Control Platform (FCP) puts you in the driver's seat. http://www.davidwriter.com/netvmgw/.

[21] Noction. Intelligent Routing Platform. http://www.noction.com/intelligent_routing_platform.

[22] Open vSwitch. http://openvswitch.org/.

[23] OpenStack. http://www.openstack.org/.

[24] OpenTSDB Project. http://www.opentsdb.net/.

[25] Sockeye's GlobalRoute 2.0 for managed routing services. http://www.networkcomputing.com/networking/sockeyes-globalroute-20-for-managed-routing-services/d/d-id/1204992?

[26] State of the Network: Project with Microsoft Manages Competing Demands. https://www.cs.princeton.edu/news/article/state-network-project-microsoft-manages-competing-demands.

[27] Summary of the December 24, 2012 Amazon ELB Service Event in the US-East Region. http://aws.amazon.com/message/680587/.

[28] Today's Outage for Several Google Services. http://googleblog.blogspot.com/2014/01/todays-outage-for-several-google.html.

[29] Web10G Project. http://web10g.org/.

[30] NTP: The Network Time Protocol. http://www.ntp.org/, 2003.

[31] Event Tracing for Windows. http://support.microsoft.com/kb/2593157, 2011.

[32] VMWare vCenter Suite. http://www.vmware.com/products/datacenter-virtualization/vcenter-operations-management/overview.html, 2013.

[33] Intel AGMT Dated 09-09-09. Hosting Virtual Networks on Multicore Platforms.

[34] NSF / Rutgers University 4847 (Prime CNS 1247764). EARS: SAVANT - High Performance Dynamic Spectrum Access via Inter Network Collaboration.

[35] PRIME DARPA N66001-11-2-4206 UIUC 2012-00310-02. DARPA Cloud Computing.

[36] Aditya Akella, Bruce Maggs, Srinivasan Seshan, and Anees Shaikh. On the Performance Benefits of Multihoming Route Control. *IEEE/ACM Transactions on Networking*, 16(1):91–104, February 2008.

[37] Aditya Akella, Bruce Maggs, Srinivasan Seshan, Anees Shaikh, and Ramesh Sitaraman. A Measurement-based Analysis of Multihoming. In *ACM SIGCOMM*, 2003.

[38] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A Scalable, Commodity Data Center Network Architecture. In *ACM SIGCOMM*, 2008.

[39] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *USENIX NSDI*, San Jose, California, April 2010.

[40] R. J. Atkinson and S. N. Bhatti. Identifier-Locator Network Protocol (ILNP) Architectural Description. RFC 6740, Nov 2012.

[41] Theophilus Benson, Aditya Akella, and David A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *ACM IMC*, 2010.

[42] Kevin Borders, Jonathan Springer, and Matthew Burnside. Chimera: A Declarative Language for Streaming Network Traffic Analysis. In *USENIX Security*, 2012.

[43] Sergey Brin and Lawrence Page. The Anatomy of a Large-scale Hypertextual Web Search Engine. In *International Conference on World Wide Web*, 1998.

[44] Matthew Caesar, Donald Caldwell, Nick Feamster, Jennifer Rexford, Aman Shaikh, and Jacobus van der Merwe. Design and Implementation of a Routing Control Platform. In *USENIX NSDI*, May 2005.

[45] Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking Control of the Enterprise. In *ACM SIGCOMM*, 2007.

[46] Martin 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*, July 2006.

[47] Rocky KC Chang and Michael Lo. Inbound Traffic Engineering for Multihomed ASs using AS Path Prepending. *IEEE Network*, 19(2):18–25, 2005.

[48] Chao-Chih Chen, Peng Sun, Lihua Yuan, David A. Maltz, Chen-Nee Chuah, and Prasant Mohapatra. SWiM: Switch Manager For Data Center Networks. *IEEE Internet Computing*, April 2014.

[49] Kai Chen, Chuanxiong Guo, Haitao Wu, Jing Yuan, Zhenqian Feng, Yan Chen, Songwu Lu, and Wenfei Wu. Generic and Automatic Address Configuration for Data Center Networks. In *ACM SIGCOMM*, August 2010.

[50] Luca Cittadini, Wolfgang Muhlbauer, Steve Uhlig, Randy Bush, Pierre Francois, and Olaf Maennel. Evolution of Internet Address Space Deaggregation: Myths and Reality. *Journal on Selected Areas in Communications*, 28(8):1238–1249, 2010.

[51] D. Clark, S. Bauer, K. Claffy, A. Dhamdhere, B. Huffaker, W. Lehr, and M. Luckie. Measurement and Analysis of Internet Interconnection and Congestion. In *Telecommunications Policy Research Conference (TPRC)*, September 2014.

[52] NSF CNS-1162112. NeTS: Medium: Collaborative Research: Optimizing Network Support for Cloud Services: From Short-Term Measurements to Long-Term Planning.

[53] Evan Cooke, Richard Mortier, Austin Donnelly, Paul Barham, and Rebecca Isaacs. Reclaiming network-wide visibility using ubiquitous end system monitors. In *USENIX ATC*, 2006.

[54] Chuck Cranor, Theodore Johnson, Oliver Spataschek, and Vladislav Shkapenyuk. Gigascope: A Stream Database for Network Applications. In *ACM SIGMOD*, 2003.

[55] Andrew Curtis, Wonho Kim, and Praveen Yalagandula. Mahout: Low-Overhead Datacenter Traffic Management using End-Host-Based Elephant Detection. In *IEEE INFOCOM*, 2011.

[56] Cédric De Launois, Olivier Bonaventure, and Marc Lobelle. The NAROS Approach for IPv6 Multihoming with Traffic Engineering. In *Quality for All*, pages 112–121. Springer, 2003.

[57] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *USENIX OSDI*, 2004.

[58] Jeffrey Dean and Sanjay Ghemawat. MapReduce: A Flexible Data Processing Tool. *Commun. ACM*, 53(1):72–77, January 2010.

[59] Colin Dixon, Hardeep Uppal, Vjekoslav Brajkovic, Dane Brandon, Thomas Anderson, and Arvind Krishnamurthy. ETTM: A Scalable Fault Tolerant Network Manager. In *USENIX NSDI*, 2011.

[60] Sean Donovan and Nick Feamster. Intentional Network Monitoring: Finding the Needle Without Capturing the Haystack. In *ACM HotNets*, 2014.

[61] Conal Elliott and Paul Hudak. Functional Reactive Animation. In *ACM SIGPLAN International Conference on Functional Programming*, 1997.

[62] D. Farinacci, V. Fuller, D. Meyer, and D. Lewis. The Locator/ID Separation Protocol (LISP). IETF Request for Comments 6830, January 2013.

[63] Nick Feamster, Jay Borkenhagen, and Jennifer Rexford. Guidelines for Interdomain Traffic Engineering. *ACM SIGCOMM Computer Communication Review*, 33(5):19–30, 2003.

[64] Nick Feamster, Jennifer Rexford, and Ellen Zegura. The Road to SDN. *ACM Queue*, 11(12):20:20–20:40, December 2013.

[65] Anja Feldmann, Luca Cittadini, Wolfgang Mühlbauer, Randy Bush, and Olaf Maennel. HAIR: Hierarchical Architecture for Internet Routing. In *Workshop on Re-architecting the Internet*, pages 43–48. ACM, 2009.

[66] Andrew Ferguson, Arjun Guha, Chen Liang, Rodrigo Fonseca, and Shriram Krishnamurthi. Participatory Networking: An API for Application Control of SDNs. In *ACM SIGCOMM*, August 2013.

[67] 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*, 2011.

[68] Rohan Gandhi, Hongqiang Harry Liu, Y. Charlie Hu, Guohan Lu, Jitendra Padhye, Lihua Yuan, and Ming Zhang. Duet: Cloud Scale Load Balancing with Hardware and Software. In *ACM SIGCOMM*, 2014.

[69] Ruomei Gao, Constantinos Dovrolis, and Ellen W Zegura. Interdomain Ingress Traffic Engineering through Optimized AS-path Prepending. In *Networking Technologies, Services, and Protocols; Performance of Computer and Communication Networks; Mobile and Wireless Communications Systems*, pages 647–658. Springer, 2005.

[70] Mojgan Ghasemi, Theophilus Benson, and Jennifer Rexford. RINC: Real-Time Inference-based Network Diagnosis in the Cloud. Technical Report TR-975-14, Princeton University, 2015.

[71] Monia Ghobadi, Soheil Hassas Yeganeh, and Yashar Ganjali. Rethinking End-to-End Congestion Control in Software-Defined Networks. In *ACM HotNets*, October 2012.

[72] David K. Goldenberg, Lili Qiu, Haiyong Xie, Yang Richard Yang, and Yin Zhang. Optimizing Cost and Performance for Multihoming. In *ACM SIGCOMM*, 2004.

[73] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. VL2: A Scalable and Flexible Data Center Network. In *ACM SIGCOMM*, Barcelona, Spain, 2009.

[74] 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 SIGCOMM Computer Communication Review*, 35(5):41–54, October 2005.

[75] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. NOX: Towards an Operating System for Networks. *ACM SIGCOMM Computer Communication Review*, 38(3):105–110, July 2008.

[76] Chuanxiong Guo, Guohan Lu, Dan Li, Haitao Wu, Xuan Zhang, Yunfeng Shi, Chen Tian, Yongguang Zhang, and Songwu Lu. BCube: A High Performance, Server-centric Network Architecture for Modular Data Centers. In *ACM SIGCOMM*, 2009.

[77] Chuanxiong Guo, Haitao Wu, Kun Tan, Lei Shi, Yongguang Zhang, and Songwu Lu. Dcell: A Scalable and Fault-tolerant Network Structure for Data Centers. In *ACM SIGCOMM*, 2008.

[78] Brandon Heller, Srini Seetharaman, Priya Mahadevan, Yiannis Yiakoumis, Puneet Sharma, Sujata Banerjee, and Nick McKeown. ElasticTree: Saving Energy in Data Center Networks. In *USENIX NSDI*, April 2010.

[79] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *USENIX NSDI*, March 2011.

[80] Chi-Yao Hong, Matthew Caesar, and P. Brighten Godfrey. Finishing Flows Quickly with Preemptive Scheduling. In *ACM SIGCOMM*, 2012.

[81] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving High Utilization with Software-driven WAN. In *ACM SIGCOMM*, August 2013.

[82] 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*, August 2013.

[83] Srikanth Kandula, Sudipta Sengupta, Albert Greenberg, Parveen Patel, and Ronnie Chaiken. The Nature of Datacenter Traffic: Measurements & Analysis. In *ACM IMC*, 2009.

[84] Thomas Karagiannis, Richard Mortier, and Antony Rowstron. Network Exception Handlers: Host-network Control in Enterprise Networks. In *ACM SIGCOMM*, 2008.

[85] Peyman Kazemian, George Varghese, and Nick McKeown. Header Space Analysis: Static Checking for Networks. In *USENIX NSDI*, April 2012.

[86] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. VeriFlow: Verifying Network-wide Invariants in Real Time. In *USENIX NSDI*, April 2013.

[87] Changhoon Kim, Matthew Caesar, and Jennifer Rexford. Floodless in Seattle: A Scalable Ethernet Architecture for Large Enterprises. In *ACM SIGCOMM*, 2008.

[88] Wonho Kim and P. Sharma. Hercules: Integrated Control Framework for Datacenter Traffic Management. In *IEEE Network Operations and Management Symposium*, April 2012.

[89] Teemu Koponen, Keith Amidon, Peter Balland, Martin Casado, Anupam Chanda, Bryan Fulton, Igor Ganichev, Jesse Gross, 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 *USENIX NSDI*, April 2014.

[90] Teemu Koponen, Martin 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 *USENIX OSDI*, Vancouver, BC, Canada, October 2010.

[91] Bob Lantz, Brian O'Connor, Jonathan Hart, Pankaj Berde, Pavlin Radoslavov, Masayoshi Kobayashi, Toshio Koide, Yuta Higuchi, Matteo Gerola, William Snow, and Guru Parulkar. ONOS: Towards an Open, Distributed SDN OS. In *ACM SIGCOMM HotSDN Workshop*, August 2014.

[92] Young Lee, Greg Bernstein, Ning So, Tae Yeon Kim, Kohei Shiomoto, and Oscar Gonzalez de Dios. Research Proposal for Cross Stratum Optimization (CSO) between Data Centers and Networks. `http://tools.ietf.org/html/draft-lee-cross-stratum-optimization-datacenter-00`, March 2011.

[93] Hongqiang Harry Liu, Xin Wu, Ming Zhang, Lihua Yuan, Roger Wattenhofer, and David Maltz. zUpdate: Updating Data Center Networks with Zero Loss. In *ACM SIGCOMM*, August 2013.

[94] Samantha Lo and Rocky K. C. Chang. Measuring the Effects of Route Prepending for Stub Autonomous Systems. In *IEEE ICC Workshop on Traffic Engineering in Next Generation IP Networks*, June 2007.

[95] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. Debugging the Data Plane with Anteater. In *ACM SIGCOMM*, August 2011.

[96] Matt Mathis, John Heffner, and Raghu Raghunarayan. RFC 4898: TCP Extended Statistics MIB. `http://www.ietf.org/rfc/rfc4898.txt`, May 2007.

[97] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, March 2008.

[98] Jeff Mogul, Alvin AuYoung, Sujata Banerjee, Jeongkeun Lee, Jayaram Mudigonda, Lucian Popa, Puneet Sharma, and Yoshio Turner. Corybantic: Towards Modular Composition of SDN Control Programs. In *ACM HotNets*, November 2013.

[99] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing Software-defined Networks. In *USENIX NSDI*, April 2013.

[100] R. Moskowitz, P. Nikander, P. Jokela, and T. Henderson. Host Identity Protocol, April 2008. RFC 5201.

[101] Prime ONR N00014-12-1-757. Networks Opposing Botnets (NoBot).

[102] Tim Nelson, Arjun Guha, Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. A Balance of Power: Expressive, Analyzable Controller Programming. In *ACM SIGCOMM HotSDN*, 2013.

[103] Henrik Nilsson, Antony Courtney, and John Peterson. Functional Reactive Programming, Continued. In *ACM SIGPLAN Workshop on Haskell*, 2002.

[104] Radhika Niranjan Mysore, Andreas Pamboris, Nathan Farrington, Nelson Huang, Pardis Miri, Sivasankar Radhakrishnan, Vikram Subramanya, and Amin Vahdat. PortLand: A Scalable Fault-tolerant Layer 2 Data Center Network Fabric. In *ACM SIGCOMM*, 2009.

[105] Erik Nordmark and Marcelo Bagnulo. Shim6: Level 3 Multihoming Shim Protocol for IPv6. IETF Request for Comments 5533, June 2009.

[106] Xinming Ou, Sudhakar Govindavajhala, and Andrew W. Appel. MulVAL: A Logic-based Network Security Analyzer. In *USENIX Security*, 2005.

[107] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A. Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, Changhoon Kim, and Naveen Karri. Ananta: Cloud Scale Load Balancing. In *ACM SIGCOMM*, August 2013.

[108] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A Comparison of Approaches to Large-scale Data Analysis. In *ACM SIGMOD*, 2009.

[109] Ben Pfaff, Justin Pettit, Keith Amidon, Martin Casado, Teemu Koponen, and Scott Shenker. Extending networking into the virtualization layer. In *ACM HotNets*, October 2009.

[110] Lucian Popa, Gautam Kumar, Mosharaf Chowdhury, Arvind Krishnamurthy, Sylvia Ratnasamy, and Ion Stoica. FairCloud: Sharing the Network in Cloud Computing. In *ACM SIGCOMM*, 2012.

[111] Bruno Quoitin and Olivier Bonaventure. A Cooperative Approach to Interdomain Traffic Engineering. In *Next Generation Internet Networks*, pages 450–457. IEEE, 2005.

[112] Bruno Quoitin, Cristel Pelsser, Louis Swinnen, Ouvier Bonaventure, and Steve Uhlig. Interdomain Traffic Engineering with BGP. *IEEE Communications Magazine*, 41(5):122–128, 2003.

[113] Bruno Quoitin, Sébastien Tandel, Steve Uhlig, and Olivier Bonaventure. Interdomain Traffic Engineering with Redistribution Communities. *Computer Communications*, 27(4):355–363, 2004.

[114] Barath Raghavan, Kashi Vishwanath, Sriram Ramabhadran, Kenneth Yocum, and Alex C. Snoeren. Cloud Control with Distributed Rate Limiting. In *ACM SIGCOMM*, 2007.

[115] Brandon Schlinker, Kyriakos Zarifis, Italo Cunha, Nick Feamster, and Ethan Katz-Bassett. PEERING: An AS for Us. In *ACM HotNets*, 2014.

[116] Justine Sherry, Daniel C. Kim, Seshadri S. Mahalingam, Amy Tang, Steve Wang, and Sylvia Ratnasamy. Netcalls: End Host Function Calls to Network Traffic Processing Services. Technical Report UCB/EECS-2012-175, U.C. Berkeley, 2012.

[117] Rob Sherwood, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martin Casado, Nick McKeown, and Guru Parulkar. Can the Production Network Be the Testbed? In *USENIX OSDI*, October 2010.

[118] Alan Shieh, Srikanth Kandula, Albert Greenberg, Changhoon Kim, and Bikas Saha. Sharing the Data Center Network. In *USENIX NSDI*, 2011.

[119] Lakshminarayanan Subramanian, Matthew Caesar, Cheng Tien Ee, Mark Handley, Morley Mao, Scott Shenker, and Ion Stoica. HLP: A Next Generation Inter-domain Routing Protocol. In *ACM SIGCOMM*, August 2005.

[120] Peng Sun, Ratul Mahajan, Jennifer Rexford, Lihua Yuan, Ming Zhang, and Ahsan Arefin. A Network-state Management Service. In *ACM SIGCOMM*, August 2014.

[121] Peng Sun, Laurent Vanbever, and Jennifer Rexford. Scalable Programmable Inbound Traffic Engineering. In *ACM SIGCOMM SOSR*, 2015.

[122] Peng Sun, Minlan Yu, Michael J. Freedman, and Jennifer Rexford. Identifying Performance Bottlenecks in CDNs Through TCP-level Monitoring. In *ACM SIGCOMM Workshop on Measurements Up the Stack*, 2011.

[123] Peng Sun, Minlan Yu, MichaelJ. Freedman, Jennifer Rexford, and David Walker. HONE: Joint Host-Network Traffic Management in Software-Defined Networks. *Journal of Network and Systems Management*, 23(2):374–399, 2015.

[124] Doug Terry. Replicated Data Consistency Explained Through Baseball. *Communications of the ACM*, 56(12):82–89, December 2013.

[125] Vytautas Valancius, Nick Feamster, Jennifer Rexford, and Akihiro Nakao. Wide-area Route Control for Distributed Services. In *USENIX ATC*, 2010.

[126] Robbert van Renesse and Adrian Bozdog. Willow: DHT, Aggregation, and Publish/Subscribe in One Protocol. In *IPTPS*, 2004.

[127] Laurent Vanbever, Stefano Vissicchio, Cristel Pelsser, Pierre Francois, and Olivier Bonaventure. Seamless Network-wide IGP Migrations. In *ACM SIGCOMM*, August 2011.

[128] Andreas Voellmy, Junchang Wang, Y Richard Yang, Bryan Ford, and Paul Hudak. Maple: Simplifying SDN Programming Using Algorithmic Policies. In *ACM SIGCOMM*, August 2013.

[129] Feng Wang and Lixin Gao. On Inferring and Characterizing Internet Routing Policies. In *Internet Measurement Conference*, pages 15–26. ACM, 2003.

[130] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. Better Never Than Late: Meeting Deadlines in Datacenter Networks. In *ACM SIGCOMM*, 2011.

[131] Haitao Wu, Zhenqian Feng, Chuanxiong Guo, and Yongguang Zhang. ICTCP: Incast Congestion Control for TCP in Data Center Networks. In *ACM CoNEXT*, 2010.

[132] Xin Wu, Daniel Turner, Chao-Chih Chen, David A. Maltz, Xiaowei Yang, Lihua Yuan, and Ming Zhang. NetPilot: Automating Datacenter Network Failure Mitigation. In *ACM SIGCOMM*, August 2012.

[133] Praveen Yalagandula and Mike Dahlin. A Scalable Distributed Information Management System. In *ACM SIGCOMM*, 2004.

[134] Minlan Yu, Albert Greenberg, Dave Maltz, Jennifer Rexford, Lihua Yuan, Srikanth Kandula, and Changhoon Kim. Profiling Network Performance for Multi-tier Data Center Applications. In *USENIX NSDI*, 2011.

[135] Minlan Yu, Lavanya Jose, and Rui Miao. Software Defined Traffic Measurement with OpenSketch. In *USENIX NSDI*, 2013.

[136] Lihua Yuan, Chen-Nee Chuah, and Prasant Mohapatra. ProgME: Towards Programmable Network Measurement. In *ACM SIGCOMM*, 2007.

[137] David Zats, Tathagata Das, Prashanth Mohan, Dhruba Borthakur, and Randy Katz. DeTail: Reducing the Flow Completion Time Tail in Datacenter Networks. In *ACM SIGCOMM*, 2012.