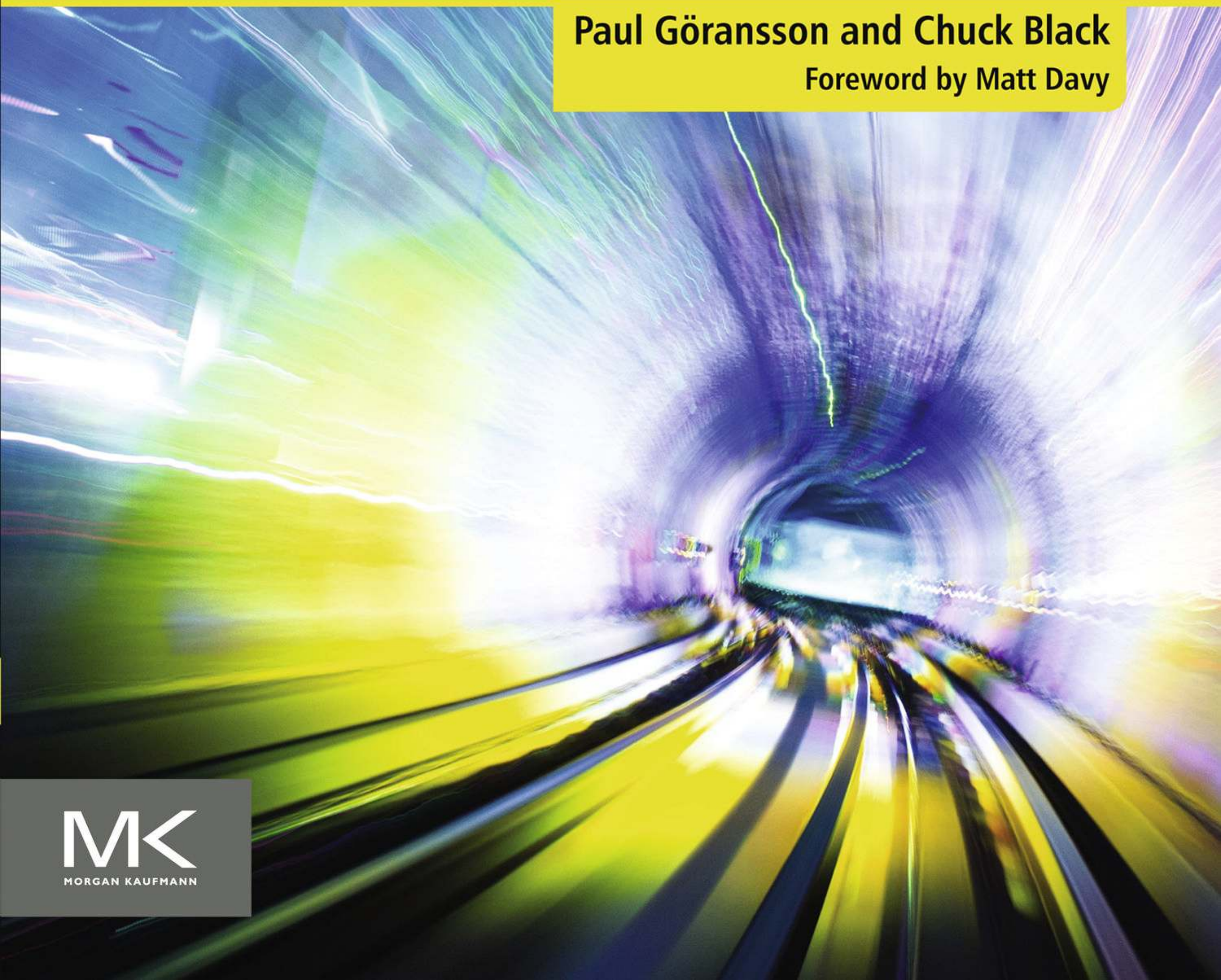
Software Defined Networks

A Comprehensive Approach

Paul Göransson and Chuck Black

Foreword by Matt Davy



MORGAN KAUFMANN

Software Defined Networks

Software Defined Networks
A Comprehensive Approach

Paul Göransson

Chuck Black

ELSEVIER

AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO
Morgan Kaufmann is an imprint of Elsevier

MK MORGAN KAUFMANN

Acquiring Editor: Steve Elliot
Editorial Project Manager: Kaitlin Herbert
Project Manager: Punithavathy Govindaradjane
Designer: Mark Rogers

Morgan Kaufmann is an imprint of Elsevier
225 Wyman Street, Waltham, MA 02451, USA

Library of Congress Cataloging-in-Publication Data
Application submitted

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library

ISBN: 978-0-12-416675-2

Printed and bound in the United States of America
14 15 16 17 18 10 9 8 7 6 5 4 3 2 1

This book is dedicated to our families.

In praise of *Software Networks: A Comprehensive Approach*

"SDN is here and Görannson's book will fill a glaring need in the technical education community. It is a new game in town. This useful reference will help get our students up to speed and ready to play in this world of immense new possibilities."

—Brian Capouch, Assistant Professor of Computer Science, Saint Joseph's College

"This book is THE collection of the state-of-the-art in SDN. Göransson and Black have compiled all of the current literature into an easy-to-read text, preparing the reader to be both current in SDN technology and aware of the impact that SDN will have today and tomorrow. This is a must-have on every network technologist's shelf."

—Scott Valcourt, Director of Strategic Technology, University of New Hampshire

"Göransson and Black have truly written a comprehensive book on SDN! This is the book we've all been waiting for. Göransson and Black provide an excellent background on SDN, how OpenFlow plays an important role in SDNs and how the close alignment with the open source movement has allowed this technology to progress at lightning speed over the past few years. Anyone who reads this book will walk away with a detailed understanding of SDN, OpenFlow, the SDN Ecosystem and even the business ramifications of implementing SDN. Personally, I can't wait to use this book with my students in my SDN course."

—Robert M. Cannistra, NYS Cloud Computing Center - SDN Innovation Lab School of Computer Science and Mathematics, Marist College

Contents

This page is intentionally left blank

List of Figures

List of Tables

This page is intentionally left blank

Foreword

What is Software Defined Networking, and why has it created so much hype in the networking industry? This seems like a simple question to answer, yet I spent nearly three years traveling the world answering that very question for everyone from undergraduate engineering students to chief technology officers of major networking vendors.

In July 2010, when I gave my first public talk on OpenFlow, the protocol at the heart of SDN, virtually no one had heard of OpenFlow or Software Defined Networking, even among the university research networking community. By May 2011, just 10 months later, OpenFlow was the talk of Interop Las Vegas, with OpenFlow prototypes on display from 15 vendors; the Open Networking Foundation had been formed to promote and standardize SDN, and every networking publication was running stories on the new technology. That was truly an amazing time in which to be involved with SDN!

During 2010 and 2011, as the network architect for Indiana University, I worked with most of the networking vendors to explain how OpenFlow and SDN could bring value to large enterprises and service provider networks. I had numerous meetings, phone calls, and lively discussions over beers with both engineers and executives from established players such as Cisco, Juniper, Hewlett-Packard, IBM, Dell, and Brocade as well as numerous startups. My team engaged in collaborations and early pilots with both Nicira and Big Switch Networks, the earliest and most talked about SDN startups. I was even fortunate enough to meet with developers from Google for briefings on the much talked about Google OpenFlow network nearly two years before Google first spoke publicly about the project.

During that time period, I explained SDN and why it was different and valuable to literally thousands of people, from students to network engineers to C-level executives and even some university presidents. Some people immediately "got it" after just a few minutes of explanation, often before I could get to Slide 3 in the presentation. With some people it took a full-day class before the light bulb went on. Almost universally, the people who were quickest to understand SDN's potential to impact the industry were those who had been in networking long enough to remember a time before the Internet, who had actually built networking products and who understood the business of networking.

This is why I was so excited when Paul and Chuck approached me about their plans to write this book. They both have tremendous experience in the networking industry, dating back longer than they probably care to admit. They have built networking solutions through multiple generations of technology. Having led two successful startups, Paul is also intimately familiar with the business of the networking industry. This depth and breadth of experience are absolutely invaluable when explaining SDN, positioning SDN in the context of the last 30-plus years of computer networking, and forecasting the potential impacts on networking in the coming years.

Perhaps the best proof point was a conversation I had with Chuck after he wrote his first network access control (NAC) solution based on an SDN controller. Chuck had spent several years at HP as the architect for that company's NAC solution and knew firsthand how challenging it was to build a NAC solution based on "legacy" technologies such as Simple Network Management Protocol (SNMP) and Remote Authentication Dial-In User Service (RADIUS). After he had a working NAC solution built on an open-source SDN controller in a timeframe measured in days, not months, there was no doubt Chuck was sold on the value of SDN!

There is no shortage of information on the Internet today about SDN, in the form of blog posts, podcasts, tweets, and the like. There is even a self-published book on the OpenFlow protocol.

However, for someone who has not been directly involved with SDN for the past couple years and wants to get up to speed, there has been no comprehensive source of information on SDN until the authors created this book.

The book is extremely approachable for anyone interested in networking and SDN, including undergraduate and graduate students, networking professionals, and IT managers. It is very much self-contained and does not require extensive knowledge of networking in order to provide value to the reader. It provides the reader with valuable context by presenting a brief history of networking and a description of the technologies and industry landscape leading up to the creation of OpenFlow and SDN. If you are a network architect or an IT manager trying to compare multiple solutions that claim to be based on SDN but which appear to be based on very different technology, the authors provide a solid basis for understanding and evaluating the various competing approaches of SDN in the market.

Over the past three years, I have spent more time than I care to recall answering the simple question, *What is SDN and why is it valuable?* I am regularly contacted via email and am approached at conferences by people telling me they want to learn more about SDN and asking me what they should read. Until now, I have had to point them to a long list of podcasts, whitepapers, blog posts, and videos of conference presentations that provide a very fragmented, disjointed, and often biased perspective on SDN. With this book, there is finally a comprehensive resource on SDN that I can feel good about recommending.

Matt Davy
Tallac Networks, Inc.
October 7, 2013

Matt Davy *is a world-renowned expert on Software Defined Networking technology. At Indiana University, he served as executive director of InCNTRE, the SDN Interoperability Lab, Network Research, and Internships and Training. He was the lead architect for an enterprise network with more than 120,000 users, 100,000 Ethernet ports, and 5,000 wireless access points. He has 16 years of experience designing and operating large service provider and enterprise networks.*

Preface

When we initially conceived of the idea of writing this book, we were motivated in part by the lack of a single good reference for a comprehensive overview of SDN. Although both of us as authors were involved professionally with SDN technologies, even we found information related to SDN to be largely unavailable in any single comprehensive source. We realized that for the very large numbers of professionals who were *not* directly working with SDN but who needed to learn about it, this was a big problem. Thus, our broad-brush goal in writing this book is to describe the setting that gave rise to SDN, to outline the defining characteristics that distinguish it from competing technologies, and to explain the numerous significant commercial impacts that this nascent technology is already having.

One of the challenges in writing an early book about such a rapidly evolving technology is that it is a moving target. During the months of development of this text, things have changed at such a rapid rate that we can only strive for this to be an accurate and comprehensive snapshot of the current state of SDN technology. It will surely continue to evolve in the years following publication. We have selected to use the phrase *A Comprehensive Approach* as part of our title. There are many competing ideas in use today, the creators of which want to jump on the SDN bandwagon. Whatever aspect or type of SDN technology our reader may be required to deal with, we at least hope that he or she will be able to place it into the broader SDN context through reading this book. For this purpose, we try to discuss a variety of definitions of SDN. We hope that no reader takes offense that we were not dogmatic in our application of the definition of SDN in this work.

Individuals who are interested in learning about Software Defined Networks or having a general interest in any of the following topics:

- Networking
- Switching
- Software Defined Networks
- OpenFlow
- OpenStack
- Network virtualization

will find this book useful.

Software Defined Networking is a broad field that is rapidly expanding. Although we have attempted to be as comprehensive as possible, the interested reader may need to pursue certain technical topics via the references provided. We assume that the reader has no special knowledge other than a basic understanding of computer concepts. Some experience in computer programming and computer networking will be helpful for understanding the material presented. The book contains a large number of figures and diagrams to explain and illustrate networking concepts that are being defined or discussed. These graphics help the reader to continue straight through the text without feeling the need to reach for other references.

Suggestions and Corrections

Although we have tried to be as careful as possible, the book may still contain some errors, and certain topics may have been omitted that readers feel are especially relevant for inclusion. In anticipation of possible future printings, we would like to correct any mistakes and incorporate as many suggestions as possible. Please send comments via email to:

chuck_black@tallac.com

About the Authors

Dr. Paul Göransson is a serial entrepreneur who has led two boot-strap startup companies through successful acquisitions by industry giants: Qosnetics by Hewlett-Packard (1999) and Meetinghouse by Cisco (2006). Paul held senior management positions with Agilent Technology's Advanced Networks Division and Cisco's Wireless Networking Business Unit. As a founder and chairperson of the Elbrys Networks board of directors, Paul currently leads corporate strategy and directs Elbrys's intellectual property portfolio. Paul received a B.A. in psychology from Brandeis University, an M.S. in computer engineering from Boston University, and a Ph.D. in computer science from the University of New Hampshire. Paul has been an avid marathoner, mountaineer, and triathlete and continues to be an active scuba diver and outdoor enthusiast. He has completed six Ironman triathlons and numerous ultramarathons and is a National Association of Underwater Instructors (NAUI) divemaster. He has lived, studied, and worked for extensive periods in France, Algeria, Venezuela, Mexico, and Sweden. Paul co-authored the book *Roaming Securely in 802.11 Networks* as well as numerous articles in technical journals related to computer networking. He is often an invited speaker at technical conferences. Paul owns and, in his spare time, manages a 130-acre beef and hay farm in southern Maine.

Chuck Black has over 31 years of experience in the field of computer networking, working in research and development labs for Hewlett-Packard for most of that time before becoming co-founder of Tallac Networks, a Software Defined Networking startup. Most recently he has been the innovator and creator of multiple networking products for HP in the area of network access control and security. Prior to this work, he developed products in the field of network management for HP's software organization. In the early days of local area networking, he was author of some of the first network topology discovery applications in the industry. Chuck holds a B.S. and M.S. in computer science from California Polytechnic State University, San Luis Obispo.

Acknowledgments

Many thanks to our families for their tremendous support throughout the years, and especially while we were writing this book.

This book would not have been possible without the unwavering support of Bill Johnson, Matt Davy, and Dr. Paul Congdon, co-founders of Tallac Networks. Their deep technical understanding of SDN, their painstaking reviews of the manuscript drafts, and their many direct contributions to the text were invaluable.

We thank Dr. Niels Sluijs, Internet communication and service engineer at Sicse in the Netherlands, for his refreshingly unbiased view and comments on the manuscript.

Our great appreciation is extended to Nopadon Juneam of Chiang Mai University, whose help was invaluable in rendering the final versions of many of the figures in this book.

Special thanks to Steve Elliot and Kaitlin Herbert at Elsevier for their encouragement and support during this project.

We are also grateful to Ali Ezzet of Tallac Networks for his careful reading of draft chapters of the manuscript. His technical expertise caught a number of bugs and greatly helped improve the manuscript. We extend our appreciation to Anthony Delli Colli of Elbrys Networks, who provided important input and advice on the business and marketing aspects presented in this work. These contributions to the book were indispensable.

A special thanks to Helen Göransson for her meticulous reviews of the many drafts of the manuscript. This work has certainly been made more readable by her efforts.

The authors gratefully acknowledge Tallac Networks for the use of selected illustrations that appear throughout the book.

Paul Göransson
Chuck Black

This page is intentionally left blank

CHAPTER

Introduction

1

It is not often that an author of a technology text gets to read about his subject matter in a major story in a current issue of a leading news magazine. The tempest surrounding *Software Defined Networking* (SDN) is indeed intense enough to make mainstream news [1]. The modern computer network has evolved into a complex beast that is challenging to manage and that strugges to scale to the requirements of some of today's environments. SDN represents a new approach that attempts to address these weaknesses of the current paradigm. SDN is a fundamentally novel way to program the switches utilized in modern data networks. SDN's move to a highly scalable and centralized network control architecture is better suited to the extremely large networks prevalent in today's megascale data centers. Rather than trying to crowbar application-specific forwarding into legacy architectures that are ill-suited to the task, SDN is designed from the outset to perform fine-grained traffic-forwarding decisions. Interest in SDN goes far beyond the research and engineering communities intrigued by this new Internet switching technology. If SDN's technological promise is realized, it will represent nothing short of a tectonic shift in the networking industry, as long-term industry incumbents may be unseated and costs to consumers may plummet. Surely, though, along with this anticipation comes a degree of over-hype, and it is important that we understand not only the potentials of this new networking model but also its limitations. In this work we endeavor to provide a technical explanation of how SDN works, an overview of those networking applications for which it is well suited and those for which it is not, a tutorial on building custom applications on top of this technology, and a discussion of the many ramifications of SDN on the networking business itself.

This introductory chapter provides background on the fundamental concepts underlying current state-of-the-art Internet switches, where data plane, control plane, and management plane will be defined and discussed. These concepts are key to understanding how SDN implements these core functions in a substantially different manner than the traditional switch architecture. We also present how forwarding decisions are made in current implementations and the limited flexibility this offers network administrators to tune the network to varying conditions. At a high level, we provide examples of how more flexible forwarding decisions could greatly enhance the business versatility of existing switches. We illustrate how breaking the control plane out of the switch itself into a separate, open-platform controller can provide this greater flexibility. We conclude by drawing parallels between the way the Linux operating system has enjoyed rapid growth by leveraging the open-source development community and how the same efficiencies can be applied to the control plane on Internet switches.

We next look at some basic packet-switching terminology that will be used throughout the text. Following that we provide a brief history of the field of packet switching and its evolution.

1.1 Basic Packet-Switching Terminology

This section defines much of the basic packet-switching terminology used throughout the book. Our convention is to italicize a new term on its first use. For more specialized concepts that are not defined in this section, they will be defined on their first use. Many packet-switching terms and phrases have several and varied meanings to different groups. Throughout the book we try to use the most accepted definition for terms and phrases. Acronyms are also defined and emphasized on their first use; the book's appendix on acronyms provides an alphabetized list of all acronyms used in this work. An advanced reader may decide to skip over this section. Others might want to skim this material and later look back to refer to specific concepts.

This terminology is an important frame of reference as we explain how SDN differs from traditional packet switching. To some degree, though, SDN does away with some of these historic concepts or changes their meaning in a fundamental way. Throughout this book, we encourage the reader to look back at these definitions and consider when the term's meaning is unchanged in SDN, when SDN requires a nuanced definition, and when a discussion of SDN requires entirely new vocabulary.

A *wide area network* (WAN) is a network that covers a broad geographical area, usually larger than a single metropolitan area.

A *local area network* (LAN) is a network that covers a limited geographical area, usually not more than a few thousand square meters in area.

A *metropolitan area network* (MAN) is a network that fills the gap between LANs and WANs. This term came into use because LANs and WANs were originally distinguished not only by their geographical areas of coverage but also by the transmission technologies and speeds that they used. With the advent of technologies resembling LANs in terms of speed and access control, but with the capability of serving a large portion of a city, the term MAN came into use to distinguish these networks as a new entity distinct from large LANs and small WANs.

A *wireless local area network* (WLAN) is a LAN in which the transmission medium is air. The typical maximum distance between any two devices in a wireless network is on the order of 50 meters. Although it is possible to use transmission media other than air for wireless communication, we will not consider such uses in our use of this term in this work.

The *physical layer* is the lowest layer of the seven-layer *Open Systems Interconnection* (OSI) model of computer networking [10]. It consists of the basic hardware transmission technology to move bits of data on a network.

The *data link layer* is the second lowest layer of the OSI model. This is the layer that provides the capability to transfer data from one device to another on a single network segment. For clarity, here we equate a LAN network segment with a collision domain. A strict definition of a LAN segment is an electrical or optical connection between network devices. For our definition of data link layer, we consider multiple segments linked by repeaters as a single LAN segment. Examples of network segments are a single LAN, such as an Ethernet, or a point-to-point communications link between adjacent nodes in a WAN. The link layer includes: (1) mechanisms to detect sequencing errors or bit errors that may occur during transmission, (2) some mechanism of flow control between the sender and receiver across that network segment, and (3) a multiplexing ability that allows multiple network protocols to use the same communications medium. These three functions are considered part of the *logical link control* (LLC) component of the data link layer. The remaining functions of the data link layer are part of the *Media Access Control* component, described separately below.

Media Access Control (MAC) is the part of the data link layer that controls when a shared medium may be accessed and provides addressing in the case that multiple receivers will receive the data, yet only one should process it. The *MAC layer* is part of the data link layer. For our purposes in this book, we will not distinguish between data link layer and MAC layer.

The *network layer* provides the functions and processes that allow data to be transmitted from sender to receiver across multiple intermediate networks. To transit each intermediate network involves the data link layer processes described above. The network layer is responsible for stitching together those discrete processes such that the data correctly makes its way from the sender to the intended receiver.

Layer one is the same as the physical layer defined above.

Layer two is the same as the data link layer defined above. We will also use the term *L2* synonymously with layer two.

Layer three is the same as the network layer defined above. *L3* will be used interchangeably with layer three in this work.

A *port* is a connection to a single communications medium, including the set of data link layer and physical layer mechanisms necessary to correctly transmit and receive data over that link. This link may be of any feasible media type. We will use the term *interface* interchangeably with *port* throughout this text. Since this book also deals with virtual switches, the definition of port will be extended to include virtual interfaces, which are the endpoints of tunnels.

A *frame* is the unit of data transferred over a layer two network.

A *packet* is the unit of data transferred over a layer three network. Sometimes this term is used more generally to refer to the units of data transferred over either a layer two network (frames) as well, without distinguishing between layers two and three. When the distinction is important, a packet is always the payload of a frame.

A *MAC address* is a unique value that globally identifies a piece of networking equipment. Though these addresses are globally unique, they serve as layer two addresses, identifying a device on a layer two network topology.

An *IP address* is a nominally unique value assigned to each host in a computer network that uses the Internet Protocol for layer three addressing.

An *IPv4 address* is an IP address that is a 32-bit integer value conforming to the rules of Internet Protocol Version 4. This 32-bit integer is frequently represented in *dotted* notation, with each of the 4 bytes comprising the address represented by a decimal number from 0 to 255, separated by periods (e.g., 192.168.1.2).

An *IPv6 address* is an IP address that is a 128-bit integer conforming to the rules of Internet Protocol Version 6, introducing a much larger address space than IPv4.

A *switch* is a device that receives information on one of its ports and transmits that information out one or more of its other ports, directing this information to a specified destination.

A *circuit switch* is a switch whereby contextual information specifying where to forward the data belonging to a circuit (i.e., a connection) is maintained in the switch for a prescribed duration, which may span lapses of time when no data belonging to that connection is being processed. This context is established either by configuration or by some *call setup* or *connection setup* procedure specific to the type of circuit switch.

A *packet switch* is a switch whereby the data comprising the communication between two or more entities is treated as individual packets that each make their way independently through the network toward the destination. Packet switches may be of the connection-oriented or connectionless type.

In the *connection-oriented* model, data transits a network where there is some context information residing in each intermediate switch that allows the switch to forward the data toward its destination. The circuit switch described above is a good example of the connection-oriented paradigm.

In the *connectionless* model, data transits a network and there is sufficient data in each packet such that each intermediate switch can forward the data toward its destination without any *a priori* context having been established about that data.

A *router* is a packet switch used to separate subnets. A *subnet* is a network consisting of a set of hosts that share the same network prefix. A network prefix consists of the most significant bits of the IP address. The prefix may be of varying lengths. Usually all of the hosts on a subnet reside on the same LAN. The term *router* is now often used interchangeably with *layer three switch*. A home wireless access point typically combines the functionality of WiFi, layer two switch, and router into a single box.

To *flood* a packet is to transmit it on all ports of a switch except for the port on which it was received.

To *broadcast* a packet is the same as flooding it.

Line rate refers to the bandwidth of the communications medium connected to a port on a switch. On modern switches this bandwidth is normally measured in *megabits per second* (Mbps) or *gigabits per second* (Gbps). When we say that a switch handles packets at line rate, this means it is capable of handling a continuous stream of packets arriving on that port at that bandwidth.

WiFi is the common name for wireless communications systems that are based on the IEEE 802.11 standard.

1.2 Historical Background

The major communications networks around the world in the first half of the 20th century were the telephone networks. These networks were universally circuit-switched networks. Communication between endpoints involved the establishment of a communications path for that dialogue and the tearing down of that path at the dialogue's conclusion. The path on which the conversation traveled was static during the call. This type of communications is also referred to as connection-oriented. In addition to being based on circuit switching, the world's telephone networks were quite centralized, with large volumes of end users connected to large switching centers. Paul Baran, a Polish immigrant who became a researcher working at Rand Corporation in the United States in the 1960s, argued that in the event of enemy attack, networks like the telephone network were very easy to disrupt [7,9]. The networks had poor survivability characteristics in that the loss of a single switching center could remove phone capability from a large swath of the country. Baran's proposed solution was to transmit the voice signals of phone conversations in packets of data that could travel autonomously through the network, finding their own way to their destination. This concept included the notion that if part of the path being used for a given conversation was destroyed by enemy attack, the communication would *survive* by automatically rerouting over alternative paths to the same destination. He demonstrated that the national voice communications system could still function even if 50% of the forwarding switches were destroyed, greatly reducing the vulnerability characteristics of the centralized, circuit-switching architecture prevalent at the time.

When Baran did his work at Rand, he never could have envisioned the dominance his idea would ultimately achieve in the area of data networking. Although it was certainly not the universally accepted networking paradigm at the time, the history that followed is now part of Internet folklore. Baran's ideas became embodied in the *Department of Defense's* (DOD's) experimental ARPANET network that began to operate in 1969. The ARPANET connected academic research institutions, military departments, and defense contractors. This decentralized, connectionless network grew over the years until bursting on the commercial landscape around 1990 in the form of the Internet, known and loved around the world today.

For decades after the emergence of the ARPANET, networking professionals waged battles over the advantages of connection-based vs. connectionless architectures and centralized vs. distributed architectures. The tides in this struggle seemed to turn one way, only to reverse some years later. The explosive growth of the Internet in the 1990s seemed to provide a conclusion to these arguments, at least as far as computer networking was concerned. The Internet was in ascendance and was unequivocally a distributed, connectionless architecture. Older connection-oriented protocols like X.25 [6] seemed destined to become a thing of the past. Any overly centralized design was considered too vulnerable to attack, whether malicious or a simple act of nature. Even the new kid on the block, *Asynchronous Transfer Mode* (ATM) [6], hyped in the mid-1990s as capable of handling line rates greater than the Internet could ever handle, would eventually be largely displaced by the ever-flexible Internet, which somehow managed to handle line rates in the tens of gigabits-per-second range, once thought possible only by using cell-switching technology such as ATM.

1.3 The Modern Data Center

The Internet came to dominate computer networking to a degree rarely seen in the history of technology and was, at its core, connectionless and distributed. During this period of ascendance, the *World Wide Web* (WWW), an offspring of the Internet, spawned by Sir Tim Berners-Lee of the United Kingdom, gave rise to ever-growing data centers hosting ever more complex and more heavily subscribed web services. These data centers served as environmentally protected warehouses housing large numbers of computers that served as compute and storage servers. The warehouses themselves were protected against environmental entropy as much as possible by being situated in disaster-unlikely geographies with redundant power systems and ultimately with duplicate capacity at disparate geographical locations.

Because of the large numbers of these servers, they were physically arranged into highly organized rows of racks of servers. Over time, the demand for efficiency drove the migration of individual server computers into server *blades* in densely packed racks. Racks of compute-servers were hierarchically organized such that *top-of-rack* (ToR) switches provided the networking within the rack and the inter-rack interface capability. We depict the rigid hierarchy typical in today's data center in Figure 1.1.

During this evolution, the sheer numbers of computers in the data center grew rapidly. Data centers are being built now that will accommodate over 120,000 physical servers [8]. State-of-the-art physical servers can conceivably host 20 *virtual machines* (VMs) per physical server. This means that the internal network in such a data center would interconnect 2,400,000 hosts! These growing numbers of host computers within a data center all communicate with each other via a set of communications protocols and networking devices that were optimized to work over a large, disparate geographical area with unreliable communications links. Obviously, tens of thousands of computers sitting side by side in an environmental cocoon with highly reliable communications links is a different nature of network altogether. It gradually emerged that the goals of survivability in the face of lost communications links or a bombed-out network control center were not highly relevant in this emerging reality of the data

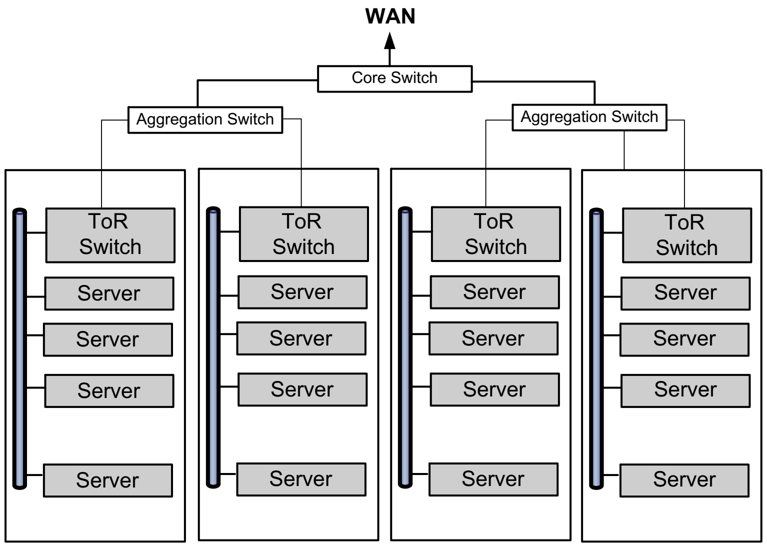


FIGURE 1.1

Typical data center network topology.

center. Indeed, an enormous amount of complexity invented to survive precisely those situations was making untenable the operation of the ever larger and more complex data center.

Beyond the obvious difference in the stability of the network topology, the sheer scale of these mega data centers in terms of nodes and links creates a network management challenge different from those encountered previously. Network management systems designed for carrier public networks or large corporate intranets simply cannot scale to these numbers. A new network management paradigm is needed.

Furthermore, though these data centers exist in order to support interaction with the turbulent world outside their walls, current research [13] predicts that by 2014 80% of the traffic in data centers will be *East-West* traffic. East-West traffic is composed of packets sent by one host in a data center to another host in that same data center. Analogously, *North-South* traffic is traffic entering (leaving) the data center from (to) the outside world. For example, a user's web browser query about the weather might be processed by a web server (North-South) that retrieves data from a storage server in the same data center (East-West) before responding to the user (North-South). The protocols designed to achieve robustness in the geographically dispersed wide-area Internet today require that routers spend more than 30% of their CPU cycles [8] rediscovering and recalculating routes for a network topology in the data center that is highly static and only changed under strict centralized control. This increasing preponderance of East-West traffic does not benefit from the overhead and complexities that have evolved in traditional network switches to provide just the decentralized survivability that Paul Baran so wisely envisioned for the WANs of the past.

The mega data centers discussed in this section differ from prior networks in a number of ways: stability of topology, traffic patterns, and sheer scale. Traditional networking technologies are simply inadequate to scale to the levels being required. SDN, a technology designed explicitly to work well with this new breed of network, represents a fundamental transformation from traditional Internet switching. To understand how SDN does differ, in Sections 1.4 and 1.5 we review how legacy Internet switches work to establish a baseline for comparison with SDN.

Before continuing, we should emphasize that although the modern data center is the premier driver behind the current SDN fervor, by no means is SDN applicable to only the data center. When we review SDN applications in Chapter 10 we will see that SDN technology can bring important innovations in domains that have little to do with the data center, such as mobility.

1.4 Traditional Switch Architecture

We will now look at what the traditional Internet switch looks like from an architectural perspective, and how and why it has evolved to be the complex beast it is today. The various switching functions are traditionally segregated into three separate categories. Since each category may be capable of *horizontal* communication with peer elements in adjacent entities in a topology, and also capable of *vertical* communication with the other categories, it has become common to represent each of these categories as a layer or *plane*. Peer communications occur in the same plane, and cross-category messaging occurs in the third dimension, between planes.

1.4.1 Data, Control and Management Planes

We show the control, management and data planes of the traditional switch in Figure 1.2. The vast majority of packets handled by the switch are only touched by the *data plane*. The data plane consists of the various ports that are used for the reception and transmission of packets and a forwarding table with its associated logic. The data plane assumes responsibility for packet buffering, packet scheduling, header modification and forwarding. If an arriving data packet's header information is found in the forwarding table it may be subject to some header field modifications and then will be forwarded without any intervention of the other two planes. Not all packets can be handled in that way, sometimes simply because their information is not yet entered into the table, or because they belong to a control protocol that must be processed by the *control plane*. The control plane, as shown in Figure 1.2, is involved in many activities. Its principal role is to keep current the information in the forwarding table so that the data plane can independently handle as high a percentage of the traffic as possible. The control plane is responsible for processing a number of different control protocols that may affect the forwarding table, depending on the configuration and type of switch. These control protocols are jointly responsible for managing the active topology of the network. We review some of these control protocols below in Section 1.5. These control protocols are sufficiently complex as to require the use of general purpose microprocessors and accompanying software in the control plane, whereas we will see in Section 1.4.3 that the data plane logic can today be incarnated entirely in silicon.

The third plane depicted in Figure 1.2 is the management plane. Network administrators configure and monitor the switch through this plane, which in turn extracts information from or modifies data in the control and data planes as appropriate. The network administrators use some form of network management system to communicate to the management plane in a switch.

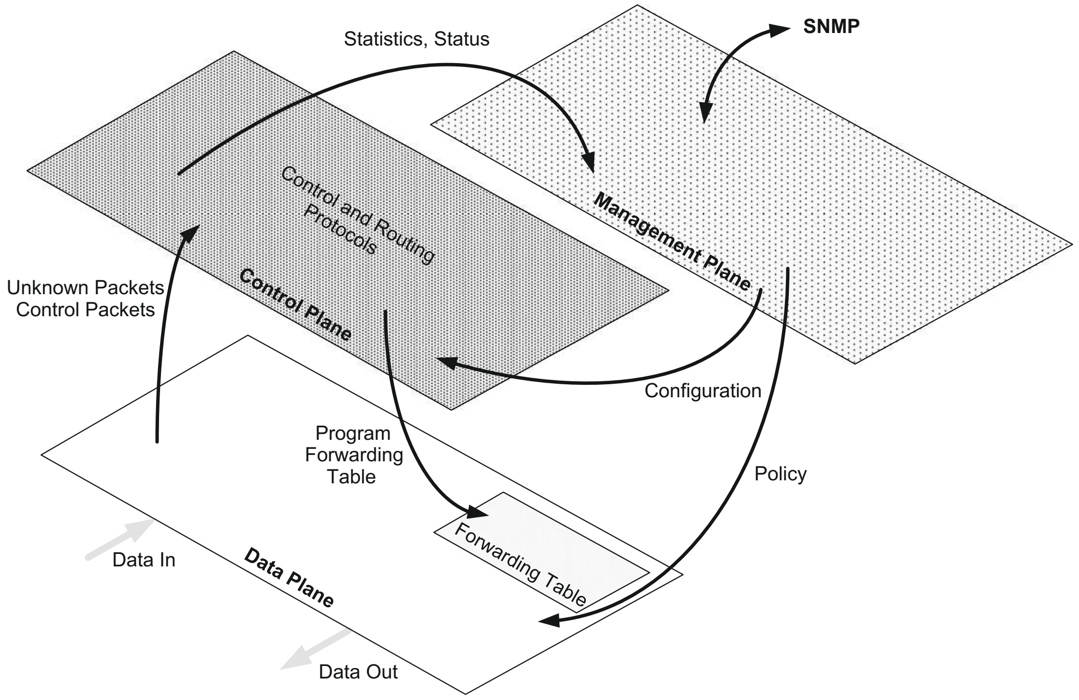


FIGURE 1.2

Roles of the control, management, and data planes.

1.4.2 Software-Based Routing and Bridging

In the late 1980s the commercial networking world was undergoing the cathartic process of the emergence of the Internet as a viable commercial network. New startups such as Cisco Systems and Wellfleet Communications had large teams of engineers building special-purpose, commercial versions of the *routers* that had previously been relegated to use in research and military networks. Although the battle between connection-oriented packet switching, such as X.25 and ATM, and the connectionless Internet architecture would continue to be waged for a number of years, ultimately the connectionless router would dominate the packet-switching world.

Before this time, most routers were just general-purpose Unix computers running software that inspected a packet that arrived on one interface and looked up the destination IP address in some efficiently searched data type such as a hierarchical tree structure. This data structure was called the *routing table*. Based on the entry found during this search, the packet would be transmitted on the indicated outbound interface. Control packets would be shunted to the appropriate control processes on the Unix system rather than being processed through that routing table.

In addition to layer three routers, many companies marketed layer two *bridges*. Bridges create a bridged LAN, which is a topology of interconnected LAN segments. Because multiple competing layer two technologies were prevalent, including Token Ring, *Fiber Distributed Data Interface* (FDDI), and different speeds and physical media versions of the original Ethernet, these bridges also served as a mechanism to interface between disparate layer two technologies. The then-current terminology was to use the terms *bridging* of *frames* for layer two and *routing* of *packets* for layer three. We will see in this book that as the technologies have evolved, these terms have become blended such that a modern networking device capable of handling both layer two and layer three is commonly referred to simply as a *packet switch*.

Demand for increased speed brought about concomitant advances in the performance of these network devices. In particular, rapid evolution occurred both in optical and twisted copper pair physical media. Ten Mbps Ethernet interfaces were commonplace by 1990, and 100 Mbps fiber and ultimately twisted pair were on the horizon. Such increases in media speed made it necessary to continually increase the speed of the router and bridge platforms. At first, such performance increases were achieved by distributing the processing over ever more parallel implementations involving multiple *blades*, each running state-of-the-art microprocessors with independent access to a distributed forwarding table. Ultimately, though, the speed of the interfaces reached a point where performing the header inspection and routing table lookup in software simply could not keep up.

1.4.3 Hardware Lookup of Forwarding Tables

The first major use of hardware acceleration in packet switching was via the use of *application-specific integrated circuits* (ASICs) to perform high-speed hashing functions for table lookups. In the mid-1990s, advances in *content-addressable memory* (CAM) technology made it possible to perform very high-speed lookups using destination address fields to find the output interface for high-speed packet forwarding. The networking application of this technology made its commercial debut in *Ethernet switches*.

At that time, the term *switch* became used to distinguish a hardware lookup-capable layer two bridge from the legacy software lookup devices discussed in Section 1.4.2. The term *router* still referred to a layer three device, and initially these were still based on software-driven address lookup. Routers were still software-based for a number of reasons. First, the packet header modifications required for layer three switching were beyond the capability of the ASICs used at the time. In addition, the address lookup in layer two switching was based on the somewhat more straightforward task of looking up a 48-bit *Media Access Control* (MAC) address. Layer three address lookup is more complicated because the devices look up the *closest* match on a network address, whereas the match may only be on the most significant bits of a network address. Any of the destination addresses matching that network address will be forwarded out the same output interface to the same *next-hop* address. The capabilities of CAMs steadily improved, however, and within a few years there were layer three routers that were capable of hardware lookup of the destination layer three address. At this point, the distinction between router and switch began to blur, and the terms *layer two switch* and *layer three switch* came into common use. Today, where the same device has the capability to simultaneously act as both a layer two and a layer three switch, the use of the simple term *switch* has become commonplace.

1.4.4 Generically Programmable Forwarding Rules

In reality there are many packets that need to be processed by a switch and that need more complicated treatment than simply being forwarded on another interface different from the one on which they arrived. For example, the packet may belong to one of the control protocols covered in Sections 1.5.1 and 1.5.2, in which case it needs to be processed by the control plane of the switch. The brute-force means of handling these exceptions by passing all exception cases to the control plane for processing rapidly became unrealistic as forwarding rates increased. The solution was to push greater intelligence down into the forwarding table such that as much of the packet processing as possible can take place at line rates in the hardware *forwarding engine* itself.

Early routers only had to perform limited packet header field modifications. This included decrementing the *Time-to-Live* (TTL) field and swapping the MAC header to the source-destination MAC headers for the next hop. In the case of multicast support, they had to replicate the packet for transmission out multiple output ports. As the switch's features grew to support advanced capabilities such as *virtual local area networks* (VLANs) and *multiprotocol label switching* (MPLS), more packet header fields needed ever more complex manipulation. To this end, the idea of *programmable rules* came into being, whereby some of this more complex processing could be encoded in rules and carried out directly in the forwarding engines at line rate. Although most control protocol packets still needed to be shunted to the control plane processor for handling, this capability allowed the evolving switches to implement ever more complex protocols and features at ever-increasing line rates. This very programmability of the hardware was one of the early seeds that gave life to the concept of SDN.

Though there are many variations in hardware architecture among modern switches, we attempt to depict an idealized state-of-the-art switch in Figure 1.3. The figure shows the major functional blocks that a packet will transit as it is being forwarded by the switch. In the figure we see that the packet may transit the *packet receive*, *ingress filter*, *packet translation*, *egress filter*, and *packet transmit* functions or be consumed by the *switch OS*. We can see that the forwarding database is influenced by the arrival of certain packets, such as when a new address is learned. We also see that the hardware generates a flow

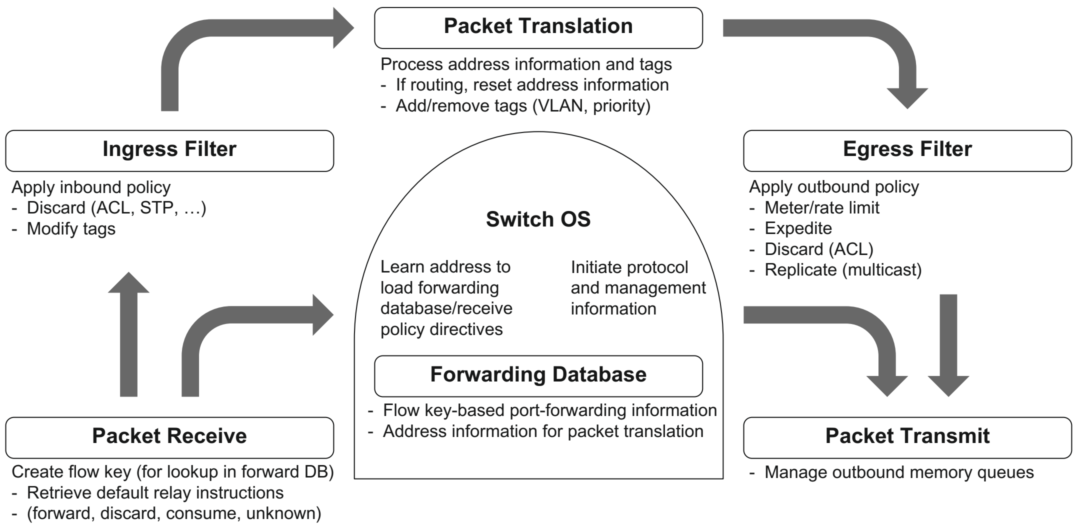

Packet Translation
Process address information and tags
- If routing, reset address information
- Add/remove tags (VLAN, priority)
Ingress Filter
Apply inbound policy
- Discard (ACL, STP, …)
- Modify tags
Egress Filter
Apply outbound policy
- Meter/rate limit
- Expedite
- Discard (ACL)
- Replicate (multicast)
Switch OS
Learn address to load forwarding database/receive policy directives
Initiate protocol and management information
Forwarding Database
- Flow key-based port-forwarding information
- Address information for packet translation
Packet Receive
Create flow key (for lookup in forward DB)
- Retrieve default relay instructions
- (forward, discard, consume, unknown)
Packet Transmit
- Manage outbound memory queues

FIGURE 1.3

A packet's journey through switching hardware.

key to serve as a lookup key into the forwarding database. The forwarding database will provide basic forwarding instructions as well as policy directives, if relevant in this switch. The ingress filter applies policy, perhaps resulting in the packet being dropped. Packet header modifications take place both at the ingress filter as well as in the packet translation function. Outbound policy is implemented at the egress filter, where again the packet may be dropped, if, for example, established quality-of-service (QoS) rates are being exceeded. Finally, the packet passes through the packet transmit function, where additional priority queue processing occurs prior to its actual transmission on the communications medium.

1.5 Autonomous and Dynamic Forwarding Tables

To respond to unstable communications links and the possible disappearance of an entire switching center, protocols were developed so that bridges and routers could dynamically and autonomously build and update their forwarding tables. In this section we provide background on some of the major protocols that were developed to provide this capability.

1.5.1 Layer Two Control

A layer two forwarding table is a lookup table indexed by destination MAC address, where the table entry indicates which port on the switch should be used to forward the packet. In the case of early, standalone bridges, when a frame was first received from a device on a given interface, the switch could *learn* the sender's MAC address and location and populate its forwarding table with this new information. When a frame arrived destined for that address, it could then use the forwarding table to send that frame out the correct port. In the case that an *unknown* destination address was received, the bridge could flood this out all interfaces, knowing that it would be dropped on all attached networks except the one where the destination actually resided. This approach no longer worked once bridges were interconnected into networks of bridges, because there were multiple paths to reach that destination and unless a single one was chosen predictably, an infinite loop could be created. A static mapping of MAC addresses to ports was an alternative, but protocols were developed that allowed the bridge to learn MAC addresses dynamically and to learn how to assign these addresses to ports in such a way as to automatically avoid creating switching loops yet predictably be able to reach the destination MAC address, even when that required transiting multiple intermediate bridges. We discuss some of these MAC-learning protocols here to illustrate ways that the legacy layer two switch's forwarding table can be built up automatically.

As we explained earlier, the Ethernet switch's role is to bridge multiple layer two networks. The most simplistic concept of such a device includes only the ability to maintain a forwarding table of MAC addresses such that when a frame reaches the switch, it understands on which attached layer two network that destination MAC address exists and that it should forward the address out on that layer two network. Clearly, if the frame arrived on a given port, and if the destination is reachable via that same port, the Ethernet switch need not be involved in forwarding that frame to its destination. In this case, the frame was dropped. The task of the Ethernet switch is considerably more complex when multiple Ethernet switches are connected into complicated topologies containing loops. Without taking care to prevent such situations, it was possible to accidentally construct forwarding loops whereby a frame is sent back to a switch that had forwarded it earlier, *ad infinitum*. The performance impact of such switching loops is more exaggerated in layer two switching than layer three, since the MAC header has no analog to the layer three *Time-to-Live* (TTL) field, which limits the number of times the packet can transit routers before it is dropped. Thus, an unintentional loop in a layer two network can literally bring the network to a halt, where the switch appears frozen but is actually 100% occupied in forwarding the same packet over and over through the loop.

The prevalence of broadcast frames in the layer two network exacerbates this situation. Since many layer two control frames must be broadcast to all MAC addresses on that layer two topology, it was very easy for a broadcast frame to return to a switch that had already broadcast that frame. This latter problem is known as *broadcast radiation*. The *Spanning Tree Protocol* (STP), specified in IEEE 802.1D [15], was designed specifically to prevent loops and thus addresses the problem of broadcast radiation. The protocol achieves this by having all switches in the topology collectively compute a *spanning tree*, a computer science concept whereby from a single root (source MAC address) there is exactly one path to every leaf (destination MAC address) on the tree. STP has been superseded, from a standards perspective at least, by protocols offering improvements on the original protocol's functionality, most recently the *Shortest Path Bridging* (SPB) [11] and the *Transparent Interconnection of Lots of Links* (TRILL) protocols [17]. In practice, though, STP is still prevalent in networks today.

1.5.2 Layer Three Control

The router's most fundamental task when it receives a packet is to determine whether the packet should be forwarded out one of its interfaces or if the packet needs some exception processing. In the former, normal case, the layer three destination address in the packet header is used to determine over which output port the packet should be forwarded. Unless the destination is directly attached to this router, the router does not need to know exactly where in the network the host is located; it only needs to know the next-hop router to which it should forward the packet. There may be large numbers of hosts that reside on the same destination network, and all of these hosts will share the same entry in the router's *routing table*. This is a table of layer three *networks* and sometimes layer three host addresses, not a table of just layer two destination host addresses, as is the case with the layer two forwarding table. Thus, the lookup that occurs in a router is to match the destination address in the packet with one of the networks in the forwarding table. Finding a match provides the next-hop router, which maps to a specific forwarding port, and the forwarding process can continue. In a traditional router, this routing table is built primarily through the use of *routing protocols*. Although there is a wide variety of these protocols and they are specialized for different situations, they all serve the common purpose of allowing the router to autonomously construct a layer three forwarding table that can automatically and dynamically change in the face of changes elsewhere in the network. Since these protocols are the basic mechanism by which the traditional layer three switch builds its forwarding table, we provide a brief overview of them here.

The *Routing Information Protocol* (RIP) [5] is a comparatively simple routing protocol that was widely deployed in small networks in the 1990s. Each router in the RIP-controlled routing domain periodically broadcasts its entire routing table on all of its interfaces. These broadcasts include the hop count from the broadcasting router to each reachable network. The weight of each hop can be tailored to accommodate relevant differences between the hops, such as link speed. Neighboring routers can integrate this reachability and hop-count information into their own routing tables, which will in turn be propagated to their neighbors. This propagation pervades the entire routing domain or *autonomous* *system* (AS). The routers in an AS share detailed routing information, allowing each member router in the AS to compute the shortest path to any destination within that AS. In a stable network, eventually all the routing tables in that routing domain will converge, and the hop-count information can be used to determine a least-cost route to reach any network in that domain. RIP is a distance-vector protocol, whereby each router uses the weighted hop count over one interface as its distance to that destination network and is able to select the best next-hop router as the one presenting the least total distance to the destination. RIP does not need to have a complete picture of the network topology in order to make this distance-vector routing computation. This differs from the next two protocols, *Open Shortest Path First* (OSPF) [2] and *Intermediate System to Intermediate System* (IS-IS) [4], both of which maintain a complete view of the network topology in order to compute best routing paths. OSPF is an IETF project, whereas IS-IS has its origins in OSI.

Both OSPF and IS-IS are link-state dynamic routing protocols. This name derives from the fact that they both maintain a complete and current view of the state of each link in the AS and can thus determine its topology. The fact that each router "knows" the topology permits it to compute the best routing paths. Since both protocols "know" the topology, they can represent all reachable networks as leaf nodes on a graph. Internal nodes represent the routers comprising that autonomous system. The edges on the graph represent the communication links joining the nodes. The *cost* of edges may be assigned by any metric, but an obvious one is bandwidth. The standard shortest-path algorithm, known as *Dijkstra's algorithm*, is used to compute the shortest path to each reachable network from each router's perspective. For this class of algorithm, the shortest path is defined as the one traversed with the smallest total cost. If a link goes down, this information is propagated via the respective protocol, and each router in the AS will shortly converge to have identical pictures of the network topology. OSPF is currently widely used in large enterprise networks. Large service provider networks tend more to use IS-IS for this purpose. This is primarily due to IS-IS having been a more mature and thus more viable alternative than OSPF in the early service provider implementations. Although OSPF and IS-IS are essentially equally capable now, there is little incentive for service providers to change.

Whereas the three routing protocols discussed above are *interior gateway protocols* (IGPs) and therefore concerned with optimizing routing within an AS, the Internet comprises a large set of interconnected autonomous systems, and it is obviously important to be able to determine the best path transiting these ASs to reach the destination address. This role is filled by an *exterior gateway protocol* (EGP). The only EGP used in the modern Internet is the *Border Gateway Protocol* (BGP) [3]. BGP routers primarily reside at the periphery of an AS and learn about networks that are reachable via peer edge routers in adjacent ASs. This network routing information can be shared with other routers in the same AS, either by using the BGP protocol between routers in the same AS, which is referred to as *internal BGP*, or by redistributing the external network routing information into the IGP routing protocol. The most common application of BGP is to provide the routing *glue* between ASs run by different service providers, allowing a host in one to transit through other providers' networks to reach the ultimate destination. Certain private networks may also require the use of BGP to interconnect multiple internal ASs that cannot grow larger due to reaching the maximum scale of the IGP they are using. BGP is called a *path vector protocol*, which is similar to a distance vector protocol like RIP, discussed above, in that it does not maintain a complete picture of the topology of the network for which it provides routing information. Unlike RIP, the metric is not a simple distance (i.e., hop count) but involves parameters such as network policies and rules as well as distance to the destination network. The policies are particularly important when used by service providers, because there may be business agreements in place that need to dictate routing choices that could trump simple routing cost computations.

1.5.3 Protocol Soup or (S)witch's Brew?

The control protocols we've listed by name are only a small part of a long list of protocols that must be implemented in a modern layer two or layer three switch in order to make it a fully functional participant in a state-of-the-art Internet. In addition to those protocols we have mentioned thus far, a modern router will likely also support LDP [18] for exchanging information about MPLS labels, as well as IGMP [12], MSDP [19], and PIM [16] for multicast routing. We see in Figure 1.4 how the control plane of the switch is bombarded with a constant barrage of control protocol traffic. Indeed, [8] indicates that in large-scale data center networks, 30% of the router's control plane capacity today is spent tracking network topology. We have explained that the Internet was conceived as and continues to evolve as an organism that is extremely robust in the face of flapping communications links and switches that may experience disruption in their operation. The Internet-family protocols were also designed to gracefully handle routers joining and leaving the topology, whether intentionally or otherwise. Such chaotic conditions still exist to varying degrees in the geographically diverse Internet around the globe, though switches and links are generally much more predictable today than they were at the Internet's inception. In particular, the data center is distinct in that it has virtually no unscheduled downtime.

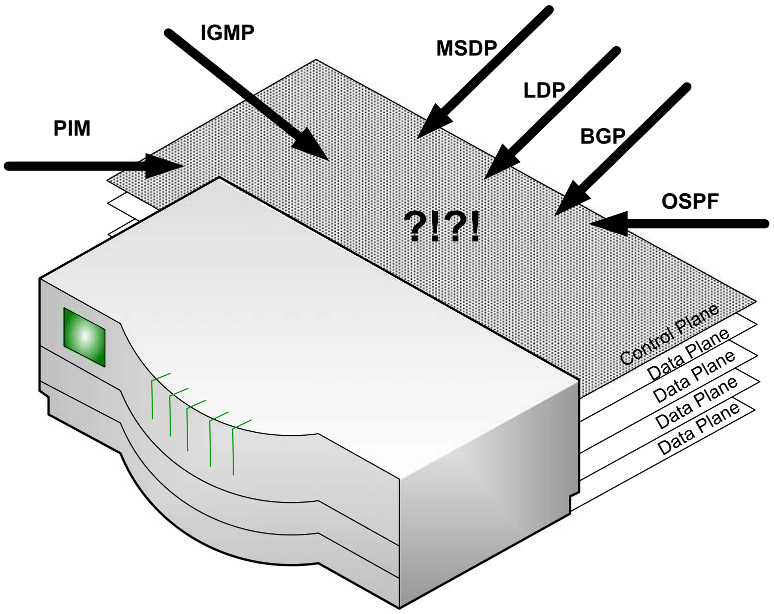


FIGURE 1.4

Control plane consternation in the switch.

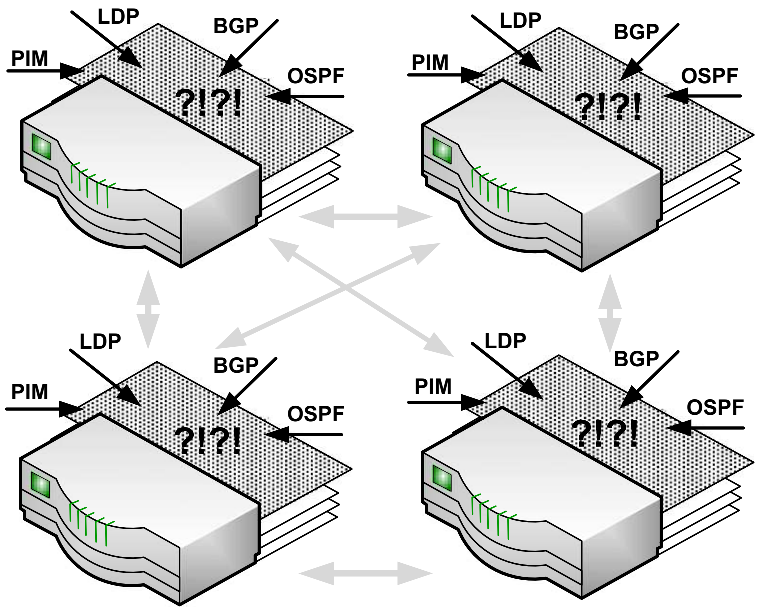


FIGURE 1.5

Overhead of dynamic distributed route computation.

Though there are links and nodes in the data center that fail occasionally, the base topology is centrally provisioned. Physical servers and VMs do not magically appear or move around in the data center; rather, they change only when the central orchestration software dictates they do so. Despite the fact that the data center's topology is static and the links are stable, the individual router's travails, shown in Figure 1.4, multiply at data center scale, as depicted in Figure 1.5. Indeed, since these control protocols are actually distributed protocols, when something is intentionally changed, such as a server being added or a communications link taken out of service, these distributed protocols take time to converge to a set of consistent forwarding tables, wreaking temporary havoc inside the data center. Thus, we reach a situation where most topology change is done programmatically and intentionally, reducing the benefit of the autonomous and distributed protocols, and the scale of the networks is larger than these protocols were designed for, making convergence times unacceptably long.

So, whereas this array of control protocols was necessary to develop autonomous switches that can dynamically and autonomously respond to rapidly changing network conditions, these conditions do not exist in the modern data center, and by using them in this overkill fashion, we have created a network management nightmare that is entirely avoidable. Instead of drinking this *(s)witch's brew of protocols*, is there a more simple approach? Is there hope for network "sobriety" by conceiving of a less distributed, less autonomous, but simpler approach of removing the control plane from the switches to a centralized control plane serving the entire network? This centralized control plane would program the forwarding tables in all the switches in the data center. The simplicity of this concept arises from the facts that

(1) the topology inside the data center is quite stable and under strict local administrative control, (2) knowledge of the topology is already centralized and controlled by the same administrators, and (3) when there are node or link failures, this global topology knowledge can be used by the centralized control to quickly reprovision a consistent set of forwarding tables. Thus, it should follow that a framework for programming the switches' forwarding tables, like that depicted in Figure 1.6, might be feasible. This more simple, centralized approach is one of the primary drivers behind SDN. This approach is expected to help address the issue of instability in the control plane, which results in long convergence times when changes occur in the network.

There are other important justifications for the migration to SDN. For example, there is a strong desire to drive down the cost of networking equipment. We review this and other arguments underpinning SDN in the next chapter. First, though, we consider a more fine-grained sort of packet switching that has been pursued for a number of years, one for which SDN has been specifically tailored.

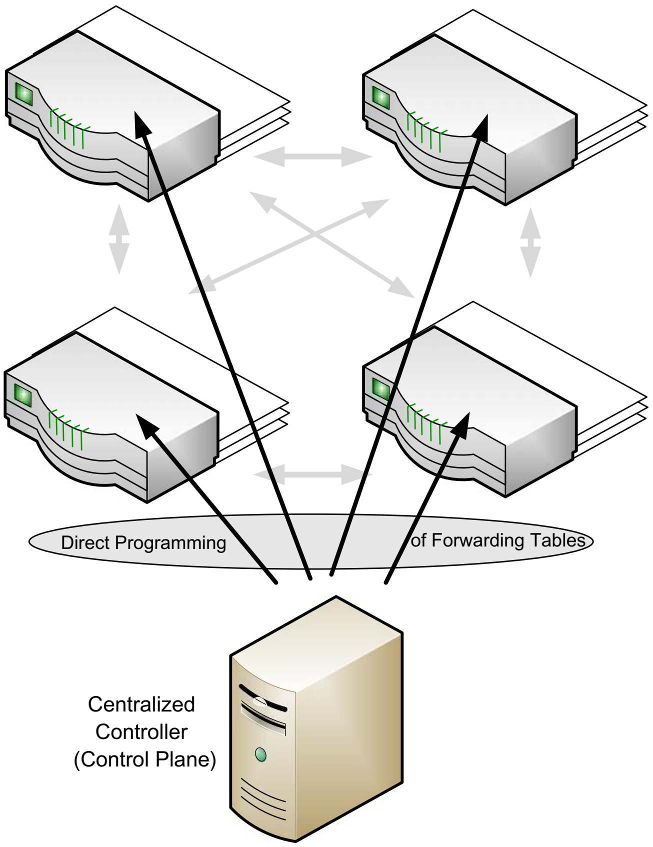


FIGURE 1.6

Centralized programming of forwarding tables.

1.6 Can We Increase the Packet-Forwarding IQ?

Whether centrally programmed or driven by distributed protocols, the forwarding tables we have discussed thus far have contained either layer two or layer three addresses. For many years now, network designers have implemented special processes and equipment that permit more fine-grained forwarding decisions. For example, it may be desirable to segregate traffic destined to different TCP ports on the same IP address such that they travel over separate paths. We may want to route different QoS classes such as data, voice, and video over different paths, even if they are directed to the same destination host. Once we leave the realm of routing on just the destination address, we enter a world where many different fields in the packet may be used for forwarding decisions. In general, this kind of routing is called *policy-based routing* (PBR). At this point we introduce a new term, *flow*, to describe a particular set of application traffic between two endpoints that receives the same forwarding treatment. On legacy commercial routers, PBR is normally implemented by using *access control lists* (ACLs) to define a set of criteria that determines whether an incoming packet corresponds to a particular flow. Thus, if, for example, we decide to forward all video traffic to a given IP address via one port and all email traffic via another port, the forwarding table will treat the traffic as two different flows with distinct forwarding rules for each. Though PBR has been in use for a number of years, this has only been possible by configuring these special forwarding rules through the management plane. Such more sophisticated routing has not been dynamically updated via the control plane in the way that simple layer two and three destinations have been. The fact that PBR has long been implemented via central configuration provides further underpinning for SDN's centralized approach of programming forwarding tables.

At the end of Section 1.5.3 we suggested that the SDN model of breaking the control plane out of the switch itself into a separate, centralized controller can provide greater simplicity and flexibility in programming the forwarding tables of the switches in a large and tightly controlled network, such as the modern data center. Figure 1.6 hints at an architecture that might allow us such programming capability, but the reality is that it becomes possible to do more than just replace the distributed routing algorithms with a cleaner alternative. In particular, we can identify particular user application traffic flows and give them different treatment, routing their traffic along different paths in some cases, even if they are destined for the same network endpoint. In this manner, we can incorporate PBR at the ground level of SDN. Many of the specialized network appliances, such as firewalls, load balancers, and *intrusion detection systems* (IDS), have long relied on performing deeper packet inspection than just the destination address. By looking at source and destination address and source and destination TCP port number, these devices have been able to segregate and quarantine traffic in a highly useful manner. If we consider that the network's forwarding tables are to be programmed from a centralized controller with global network knowledge, it follows naturally that the kind of information utilized by these special appliances could be programmed directly into the switches themselves, possibly obviating the need for the separate network appliances and simplifying the network topology. If the switches' forwarding tables have the capability to filter, segregate, and quarantine flows, the power of the network itself could grow exponentially. From its birth, SDN has been based on the notion of constructing forwarding tables defining actions to take on flows rather than having forwarding tables merely map destination address to output port.

SDN purports to incorporate a richer set of information in its processing of individual flows, including fields such as the TCP port number that provide clues regarding to which application that flow corresponds. Nonetheless, we emphasize that the SDN intelligence added at the switching layers will not require any modifications to user applications. SDN is intended to affect how layer two and layer three forwarding decisions are made and will not affect any application protocol layer.

1.7 Open Source and Technological Shifts

The SDN paradigm that this book examines represents a major technological deviation from the legacy network protocols and architectures that we have reviewed in this chapter. Those legacy technologies required countless human-years of engineering effort to develop and refine. A paradigm-shifting technology such as SDN will also require a huge engineering effort. Although some of this effort will come from commercial enterprises that expect to follow the SDN star to financial success, it is widely expected that SDN will be able to rapidly provide matching functionality to legacy systems by counting on the open source community to develop much of the SDN control plane software.

The open source model has revolutionized the way that software is developed and delivered. Functionality that used to be reinvented in every organization is now often readily available and incorporated into enterprise-class applications. Often software development organizations no longer have to devote time and energy to implement features that have already been developed.

Probably the most famous and influential body of open source code today is the Linux operating system. This complex and high-quality body of code is still developed by an amorphous and changing team of volunteer software developers around the world. It has changed the markets of commercial operating systems vendors and improved quality of both open source and competitive retail products. Numerous businesses have been spawned by innovating on top of this body of work. Small technology startups with minimal funding have had free source access to a world-class operating system, allowing them to innovate without incurring onerous licensing fees that could have prevented them from ever escaping their garage.

Another important example of open source is the OpenSSL implementation [14]. Encryption software is notoriously complex, yet it is required in more and more products in today's security-conscious world. This very complete body of asymmetrical and symmetrical encryption protocols and key algorithms has been a boon to many researchers and technology startups that could never afford to develop or license this technology, yet need it to begin pursuing their security-related ideas.

In the networking world, open source implementations of routing protocols such as BGP, OSPF, RIP, and switching functionality such as Spanning Tree have been available for many years now. Unlike the PC world, where a common *Wintel* hardware platform has made it possible to develop software trivially ported to numerous PC platforms, the hardware platforms from different *network equipment manufacturers* (NEMs) remain quite divergent. Thus, porting the open source implementations of these complex protocols to the proprietary hardware platforms of network switches has always been labor-intensive. Moving the control plane and its associated complex protocols off the proprietary switch hardware and onto a centralized server built on a PC platform makes the use of open source control plane software much more feasible.

It is not possible to fully grasp the origins of SDN without understanding that the early experimentation with SDN was done in a highly collaborative and open environment. Specifications and software were exchanged freely between participating university groups. An impressive amount of pioneering engineering work was performed without any explicit commercial funding, all based on the principle that something was being created for the greater good. It truly was and is a kind of network-world reflection of the Linux phenomenon. The fact that the most vocal supporters of SDN have been individuals and institutions that did not intend to extract financial profit from the sales of the technology made possible a natural marriage with open source. We will see in later chapters of this book that SDN has begun to have such a large commercial impact that an increasing number of proprietary implementations of SDN-related technology are reaching the market. This is an unsurprising companion on almost any journey where large sums of money can be made. Nevertheless, there remains a core SDN community that believes that openness should remain an abiding characteristic of SDN.

1.8 Organization of this Book

The first two chapters of the book define terminology about packet switching and provide a historical context for evolution of the data, control, and management planes extant in today's switches. We describe some of the limitations of the current technology and explain how these motivated the development of SDN technology. In Chapter 3 we explain the historical origins of SDN. Chapter 4 presents the key interfaces and protocols of SDN and explains how the traditional control plane function is implemented in a novel way. Up to this point in the book our references to SDN are fairly generic. Beginning in Chapter 4 we begin to distinguish the original, or *Open* SDN, from proprietary alternatives that also fall under the SDN umbrella. When we use the term *Open SDN* in this work, we refer to SDN as it was developed and defined in research institutions starting around 2008. Open SDN is tightly coupled with the *OpenFlow* protocol presented in Chapter 5 and strongly committed to the openness of the protocols and specifications of SDN. Alternative definitions of SDN exist that share some but not all of the basic precepts of Open SDN. In Chapter 6 we provide more details about these alternative SDN implementations, as well as a discussion of limitations and potential drawbacks of using SDN.

In Chapter 7 we present use cases related to the earliest driver of interest in SDN, the exploding networking demands of the data center. In Chapter 8 we discuss additional use cases that contribute to the current frenzy of interest in this new technology. Chapter 9 provides an inventory of major enterprises and universities developing SDN products today. Chapter 10 presents several real-world SDN applications, including a tutorial on writing one's own SDN application. Chapter 11 provides a survey of open source initiatives that will contribute to the standardization and dissemination of SDN. In Chapter 12 we examine the impact that this new technology is already making in the business world, and we consider what likely future technological and business consequences may be imminent. The final chapter delves into current SDN research directions and considers several novel applications of SDN. There is an appendix of SDN-related acronyms and source code from a sample SDN application as well as a comprehensive user-friendly index.

References

[1] Network effect. The Economist 2012;405(8815):67–8.
[2] Moy J. OSPF version 2, RFC 2328. Internet Engineering Task Force; 1998.
[3] Meyer D, Patel K. BGP-4 protocol analysis, RFC 4274. Internet Engineering Task Force; 2006.

[4] Oran D. OSI IS-IS intra-domain routing protocol, RFC 1142. Internet Engineering Task Force; 1990.
[5] Hendrick C. Routing information protocol, RFC 1058. Internet Engineering Task Force; 1988.
[6] de Prycker M. Asynchronous transfer mode: solution for broadband ISDN. London, UK: Ellis Horwood; 1992.
[7] Internet Hall of Fame. Paul Baran. Retrieved from <internethalloffame.org/inductees/paul-baran>.
[8] Gahsinsky I. Warehouse-scale datacenters: the case for a new approach to networking, Open Networking Summit. Palo Alto, CA, USA: Stanford University; October 2011.
[9] Christy P. OpenFlow and open networking: an introduction and overview, Ethernet Technology Summit, San Jose, CA, USA; February 2012.
[10] Bertsekas D, Gallager R. Data networks. NJ, USA: Prentice Hall; 1992.
[11] Draft standard for local and metropolitan area networks, media access control (MAC) bridges and virtual bridged local area networks, amendment XX: shortest path bridging, IEEE P802.1aq/D4.6; February 10, 2012.
[12] Cain B, Deering S, Kouvelas I, Fenner B, Thyagarajan A. Internet group management protocol, version 3, RFC 3376. Internet Engineering Task Force; 2002.
[13] Guis I. Enterprise data center networks, Open Networking Summit, Santa Clara, CA, USA; April 2012.
[14] OpenSSL. Retrieved from <www.openssl.org>.
[15] Information technology, telecommunications and information exchange between systems, local and metropolitan area networks – common specifications, Part III: Media Access Control (MAC) bridges, IEEE P802.1D/D17; May 25, 1998.
[16] Fenner B, Handley M, Holbrook H, Kouvelas I. Protocol independent multicast, sparse mode (PIM-SM): protocol specification (revised), RFC 2362. Internet Engineering Task Force; 2006.
[17] Perlman R, Eastlake D, Dutt D, Gai S, Ghanwani A. Routing bridges (RBridges): base protocol specification, RFC 6325. Internet Engineering Task Force; 2011.
[18] Andersson L, Minei I, Thomas B. LDP specification, RFC 3036. Internet Engineering Task Force; 2007.
[19] Fenner B, Meyer D. Multicast source discovery protocol (MSDP), RFC 3618. Internet Engineering Task Force; 2003.

CHAPTER

Why SDN?

2

Networking devices have been successfully developed and deployed for several decades. Repeaters and bridges, followed by routers and switches, have been used in a plethora of environments, performing their functions of filtering and forwarding packets throughout the network toward their ultimate destinations. Despite the impressive track record of these traditional technologies, the size and complexity of many modern deployments leaves them lacking. The reasons for this fact include the ever-increasing costs of owning and operating networking equipment, the need to accelerate innovation in networking, and, in particular, the increasing demands of the modern data center. This chapter investigates these trends and describes how they are nudging networking technology away from traditional methods and protocols toward the more open and innovation-friendly paradigm of SDN.

2.1 Evolution of Switches and Control Planes

We begin with a brief review of the evolution of switches and control planes that has culminated in a fertile playing field for SDN. This complements the material presented in Sections 1.4 and 1.5. The reader may find it useful to employ Figure 2.1 as a visual guide through the following sections; it provides a graphical summary of this evolution and allows the reader to understand the approximate timeframes when various switching components moved from software to hardware.

2.1.1 Simple Forwarding and Routing Using Software

In Chapter 1 we discussed the early days of computer networking, when almost everything other than the physical layer (layer one) was implemented in software. This was true for end-user systems as well as for networking devices. Whether the devices were bridges, switches, or routers, software was used extensively inside the devices in order to perform even the simplest of tasks, such as MAC-level forwarding decisions. This remained true even through the early days of the commercialized Internet in the early 1990s.

2.1.2 Independence and Autonomy in Early Devices

Early network device developers and standards creators wanted each device to perform in an autonomous and independent manner to the greatest extent possible. This was because networks were generally small and fixed, with large shared domains. A goal also was to simplify rudimentary management tasks and make the networks as *plug and play* as possible. Devices' relatively static configuration needs were performed manually. Developers went to great lengths to implement this distributed environment with

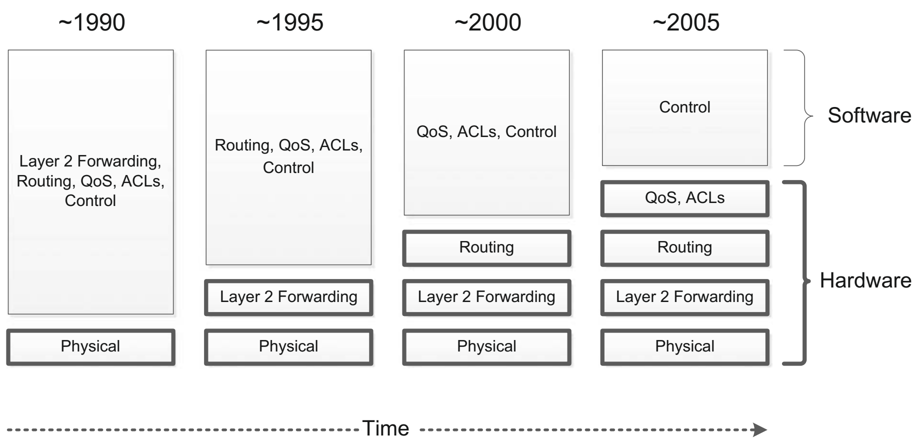


FIGURE 2.1

Networking functionality migrating to hardware.

intelligence resident in every device. Whenever coordination between devices was required, collective decisions could be made through the collaborative exchange of information between devices.

Interestingly, many of the goals of this distributed model, such as simplicity, ease of use, and automatic recovery, are similar to the goals of SDN, but as the scale and complexity of networks grew, the current distributed model has become increasingly dysfunctional.

Examples of this distributed intelligence are the layer two (bridging) and layer three (routing) protocols, which involved negotiation between the devices in order to reach a consensus on the way forwarding and routing would be performed. We introduced these protocols in Chapter 1 and provide more SDN-specific details on them here.

- **Spanning Tree Protocol.**
  Basic layer two forwarding, also known as *transparent bridging*, can be performed independently by each switch in the network. However, certain topologies require an imposition of a hierarchy on the network in order to prevent loops, which would cause broadcast radiation. The *Spanning Tree Protocol* (STP) is an example of the operation of autonomous devices participating in a distributed decision-making process to create and enforce a hierarchy on the network. The result is the correct operation of transparent bridging throughout the domain at the expense of convergence latency and possibly arbitrary configuration. This solution was a tradeoff between cost and complexity. Multiple paths could have been supported but at greater cost. STP was adequate when networks were of smaller scale, but as networks grew the spanning tree solution became problematic. These problems manifest themselves in a striking fashion when networks reach the scale of the modern data center. For example, IEEE 802.1D specifies the following default timers for STP: 15 seconds for listening, 15 seconds for learning, and 20 seconds for max-age timeout. In older networks, convergence times of 30–50 seconds were common. Such delays are not acceptable in today's data centers. As the scale of the layer two network grows, the likelihood of greater delays increases. The *Rapid Spanning Tree*

Protocol (RSTP) protocol, specified in IEEE 802.1D-2004 [6], improves this latency significantly but unfortunately is not deployed in many environments.

- **Shortest-Path Bridging.**
  STP allowed only one active path to a destination, suffered from relatively slow convergence times and was restricted to small network topologies. Although the newer implementations of STP have improved the convergence times, the single active path shortcoming has been addressed in a new layer two protocol, *Shortest-Path Bridging* (SPB), introduced in Section 1.5.1. SPB is a mechanism for allowing multiple concurrent paths through a layer two fabric via collaborative and distributed calculation of shortest and most efficient paths, then sharing that information among the participating nodes in the meshed network. This characteristic is called *multipath*. SPB accomplishes this goal by utilizing IS-IS to construct a graph representing the layer two link-state topology. Once this graph exists, shortest-path calculations are straightforward, though more complex than with spanning tree. To elaborate on what we mean by shortest-path calculations, in Figure 2.2 we depict a simple graph that can be used for calculating shortest paths in a network with five switches. The costs assigned to the various links may be assigned their values according to different criteria. A simple criterion is to make the cost of a network link inversely proportional to its bandwidth. Thus, the cost of transiting a 10 Gbps link is one-tenth that of transiting a 1 Gbps link. When the shortest-path calculation is complete, the node performing the calculation knows the least-cost path to any of the other nodes in the network. The least-cost path is considered the shortest path. For the sake of clarity, we should point out that IS-IS is used in the SPB context strictly for layer two path calculation. This differs from its classical application in calculating layer three routes, as described below. In the trivial

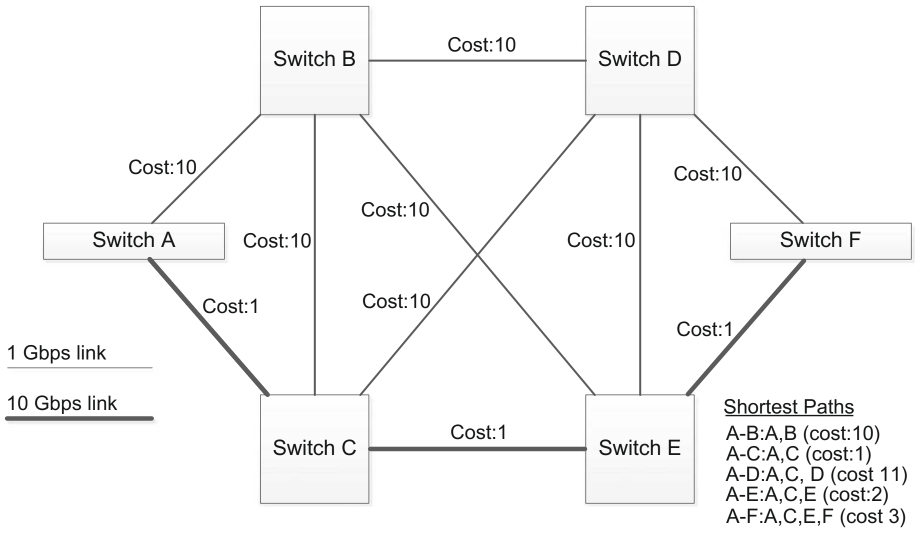


FIGURE 2.2

Example graph of a network for shortest-path calculation.

example of Figure 2.2, there is a single shortest path from node *A* to every other node. In real-life networks it is common for there to be more than one least-cost path between two nodes. The multipath characteristic of SPB would allow the traffic to be distributed across those multiple paths.

- **RIP, BGP, OSPF, and IS-IS.**
  Routing at layer three requires cooperation between devices in order to know which routers are attaching which subnets to the network. In Chapter 1 we provided background on four routing protocols: RIP, BGP, OSPF, and IS-IS. These routing protocols involve the sharing of local routing information by each device, either at the edge of the network or as an intermediate node. Their collective sharing of information allows the routing state to converge as devices share their information with each other. Each router remains autonomous in terms of its ability to make routing decisions as packets arrive. This process is one of peers sharing and negotiating among themselves, without a centralized entity aiding in the decision.

2.1.3 Software Moves into Silicon

Figure 2.1 shows that over time these functions moved from software into hardware. We now see most forwarding and filtering decisions implemented entirely in hardware. These decisions are driven by configured tables set by the control software above. This movement into hardware of the lower-level decision making has yielded tremendous benefits in terms of lowering device costs and increasing performance.

Today switching devices are typically composed of hardware components such as *application-specific integrated circuits* (ASICs), *fully programmable gate arrays* (FPGAs), and *ternary content-addressable memories* (TCAMs). The combined power of these integrated circuits allows the forwarding decisions to be made entirely in the hardware at line rate. This performance has become more critical as network speeds have increased from 1 Gbps to 10 Gbps to 40 Gbps and beyond. The hardware is now capable of handling all forwarding, routing, *access control list* (ACL), and QoS decisions. Higher-level control functions, responsible for network-wide collaboration with other devices, are implemented in software. This control software runs independently in each network device.

2.1.4 Hardware Forwarding and Control in Software

The network device evolution we have recounted thus far has yielded the following current situation:

- **Bridging** (layer two forwarding).
  Basic layer two MAC forwarding of packets is handled in the hardware tables.
- **Routing** (layer three forwarding).
  To keep up with today's high-speed links and to route packets at link speeds, layer three forwarding functionality is also implemented in hardware tables.
- **Advanced filtering and prioritization.**
  General traffic management rules such as ACLs, which filter, forward, and prioritize packets, are handled via hardware tables located in the hardware (e.g., in TCAMs) and accessed through low-level software.
- **Control.**
  The control software used to make broader routing decisions and to interact with other devices in order to converge on topologies and routing paths is implemented in software that runs autonomously

inside the devices. Since the current control plane software in networking devices lacks the ability to distribute policy information about such things as security, QoS, and ACLs, these features must still be provisioned through relatively primitive configuration and management interfaces.

Given this landscape of (1) layer two and layer three hardware handling most forwarding tasks, (2) software in the device providing control plane functionality, and (3) policy implemented via configuration and management interfaces, an opportunity presents itself to simplify networking devices and move forward to the next generation of networking.

2.1.5 The Growing Need for Simplification

In [12] the authors state that one of the major drivers for SDN is simplification. As time has passed, networking devices have become increasingly complex. This is due in part to the existing independent and autonomous design of devices that make it necessary for so much intelligence be placed inside each device. Placing more functionality in hardware in some ways simplifies the device but in other ways makes the device more complicated because of the difficult handshakes and tradeoffs between handling packets in hardware versus software.

Attempting to provide simplicity by adding features to legacy devices tends to complicate implementations rather than simplifying them. An analogy to the evolution of the *central processing unit* (CPU) can be made here. Over time CPUs became overly complex as they attempted to support more and more functions until ultimately that model of CPU was abandoned in favor of a simpler, easier-to-use CPU model called *reduced instruction set computing* (RISC). In the same way the RISC architecture served as a reset to CPU architecture, so too may SDN serve as a simplifying reset for network equipment design.

In addition to simplifying the devices themselves, there is an opportunity to simplify the management of the networks of these devices. Rather than using primitive network management tools such as SNMP and CLI, network operators would prefer to use policy-based management systems. SDN may enable such solutions [13].

2.1.6 Moving Control Off the Device

We remind the reader that control software in our context is the intelligence that determines optimal paths and responds to outages and new networking demands. At its core, SDN is about moving that control software off the device and into a centrally located compute resource that is capable of seeing the entire network and making decisions that are optimal, given a complete understanding of the situation. Though we will discuss this in much greater detail in the chapters that follow, basically SDN attempts to segregate network activities in the following manner:

- **Forwarding, filtering, and prioritization.**
  Forwarding responsibilities, implemented in hardware tables, remain on the device. In addition, features such as filtering based on ACLs and traffic prioritization are enforced locally on the device as well.
- **Control.**
  Complicated control software is removed from the device and placed into a centralized controller, which has a complete view of the network and the ability to make optimal forwarding and routing decisions. There is a migration to a programming paradigm for the control plane. The basic

forwarding hardware on the networking device is available to be programmed by external software on the controller. The control plane is no longer embedded, closed, closely coupled with the hardware, or optimized for particular embedded environments.

- **Application.**
  Above the controller is where the network applications run, implementing higher-level functions and, additionally, participating in decisions about how best to manage and control packet forwarding and distribution within the network.

Subsequent chapters will examine in greater detail how this segregation can be achieved with a minimum of investment and change by networking vendors while providing the maximum control and capability by the controller and its applications. The next section of this chapter discusses another major reason that SDN is needed today: the cost of networking devices.

2.2 Cost

Arguments related to the need for SDNs often include cost as a driving factor for this shift [1,2]. In this section we consider the impact of the status quo in networking on the cost of designing, building, purchasing, and operating network equipment.

2.2.1 Increased Cost of Development

Today's autonomous networking devices must store, manage, and run the complicated control plane software that we discussed in the previous section. The result of these demands on the device is manifested in increased cost per device due to the processing power required to run that advanced software as well as the storage capacity to hold it.

In Chapter 1 we described how software development outside the networking realm benefits greatly from the readily available open source software. For example, application server frameworks provide platforms that allow software developers to reuse code provided by those common frameworks and therefore to concentrate on solving domain-specific problems. Without the ability to leverage software functionality in this manner, each vendor would have to develop, test, and maintain large amounts of redundant code, which is not the case in an open software environment. With the closed networking environment that is prevalent today, little such leverage is available, and consequently each vendor must implement all of the common functionality required by their devices. Common network functionality and protocols must be developed by every device vendor. This clearly increases the costs attributable to software development.

In recent years, silicon vendors have been producing *common off-the-shelf* (COTS) ASICs that are capable of speeds and functionalities that rival or surpass the proprietary versions developed by networking hardware vendors. However, given the limited software leverage mentioned, vendors are not able to efficiently make use of their *merchant silicon* chips, since software must be reengineered for each product line. That limited ability to leverage the merchant silicon results in higher costs, which are passed along to the customer.

So, though their products may be quite profitable, NEMs must write and support larger amounts of software than would otherwise be necessary if networking devices were developed in truly open environments. The fact that such a large body of software must run on each and every network device serves to further increase this cost. There is additional overhead resulting from the requirement to support multiple versions of legacy protocols as well as keeping up with the latest protocols being defined by standards bodies.

2.2.2 Closed Environments Encourage Vendor Lock-in

It is true that over the years standards have been developed in the networking space for most relevant protocols and data that are used by switches and routers. For the most part, vendors do their best to implement these standards in a manner that allows heterogeneous networks of devices from multiple vendors to coexist with one another.

However, in spite of good intentions by vendors, enhancements are often added to these standard implementations, which attempt to allow a vendor's product to outperform its competition. With many vendors adding such enhancements, the end result is that each vendor product will have difficulty interoperating smoothly with products from another vendor. Adherence to standards helps alleviate the issues associated with attempting to support multiple vendor types in a network, but problems with interoperability and management often far outweigh the advantages that might be gained by choosing another vendor. As a result, customers frequently become effectively married to a vendor they chose years or even decades before. This sort of vendor lock-in alleviates downward pressure on cost because the vendor is largely safe from competition and can thus preserve high profit margins.

2.2.3 Complexity and Resistance to Change

Quite often in networking we arrive at a point of having made the network operational, and the normal impulse from that point on is to just leave things as they are, to not disturb things lest the system break and require us to start all over again. Others may have been burned by believing the latest vendor that proposed a new solution and, when the dust settled, their closed, proprietary, vendor-specific solution was just as complex as that of the previous vendor.

Unfortunately, in spite of efforts at standardization, there is still a strong argument to stay with that single-vendor solution. Often, that closed, complex solution may be easier to deploy precisely because there is only one vendor involved, and that vendor's accountability is not diluted in any way. By adopting and embracing a solution that works, we believe we lower our short-term risk. That resistance to change results in long-term technological stagnation and sluggishness. The ideal would be a simpler, more progressive world of networking, with open, efficient, and less expensive networking devices. This is a goal of SDN.

2.2.4 Increased Cost of Operating the Network

As networks become ever larger and more complex, the *operational expense* (OPEX) of the network grows. This component of the overall costs is increasingly seen to be more significant than the corresponding *capital expense* (CAPEX) component. SDN has the capacity to accelerate the automation of network management tasks in a multivendor environment [3,4]. This, combined with the fact that SDN will permit faster provisioning of new services and provides the agility to switch equipment between different services [5], should lead to lower OPEX with SDN. In Section 13.2.6 we examine proposals whereby SDN may be used in the future to reduce the power consumption of the networking equipment in a data center, which is another major contributor to network OPEX.

2.3 SDN Implications for Research and Innovation

Networking vendors have enjoyed an enviable position for over two decades. They control networking devices from the bottom up: the hardware, the low-level firmware, and the software required to produce an intelligent networking device. This platform on which the software runs is closed, and consequently only the networking vendors themselves can write the software for their own networking devices.

The reader should contrast this to the world of software and computing, where you can have many different hardware platforms created by multiple vendors and consisting of different and proprietary capabilities. Above that hardware reside multiple layers of software, which ultimately provide a common and open interface to the application layer. The *Java Virtual Machine* and the *Netbeans Integrated Development Environment* provide a good example of cross-platform development methods. Using such tools, we can develop software that may be ported between Windows PC, Linux, or Apple Mac environments. Analogously, tablets and smartphones, such as iPhones or Android-based phones, can share application software and run on different platforms with relative ease. Imagine if only Apple were able to write software for Apple products and only Microsoft were able to write software to run on PCs or Windows-based servers? Would the technological advances we have enjoyed in the last decade have taken place? Probably not.

In [11] the authors explain that this status quo has negatively impacted innovation in networking. In the next section, we examine this relationship and how, on the contrary, the emergence of SDN is likely to accelerate such innovation.

2.3.1 Status Quo Benefits Incumbent Vendors

The collective monopolies extant in the networking world today are advantageous to networking vendors. Their only legitimate competition comes from their fellow established NEMs. Although periodically new networking companies emerge to challenge the status quo, that is the exception rather than the rule. The result is that the competitive landscape evolves at a much slower pace than it would in a truly open environment. This is due to limited incentives to invest large amounts of money when the current status quo is generating reasonable profits due primarily to the lack of real competition and the resulting high margins they are able to charge for their products.

Without a more competitive environment, the market will naturally stagnate to some degree. The incumbent NEMs will continue to behave as they have in the past. The small players will struggle to survive, attempting to chip away at the industry giants but with limited success, especially since the profit margins of those giants are so large. This competition-poor environment is unhealthy for the market and for the consumers of products and services. Consider the difference in profit margins for a server vendor versus those for a networking vendor. Servers are sold at margins close to 5% or below. Networking device margins can be as high as 30% or more for the established vendors.

2.3.2 SDN Promotes Research and Innovation

Universities and research labs are focal points of innovation. In technology, innovations by academia and other research organizations have accelerated the rate of change in numerous industries. Open software environments such as Linux have helped to promote this rapid pace of advancement. For instance, if researchers are working in the area of operating systems, they can look at Linux and modify its behavior.

If they are working in the area of server virtualization or databases, they can look at KVM or Xen and MySQL or Postgres. All of these open source packages are used in large-scale commercial deployments. There is no equivalent in networking today. Unfortunately, the current closed nature of networking software, network protocols, network security, and network virtualization is such that it has been challenging to experiment, test, research, and innovate in these areas. This fact is one of the primary drivers of SDN [1]. A number of universities collaborated to propose a new standard for networking called OpenFlow, which would allow for this free and open research to take place. This makes one wonder if SDN will ultimately be to the world of networking what Linux has become to the world of computing.

General innovation, whether from academia or by entrepreneurs, is stifled by the closed nature of networking devices today. How can a creative researcher or entrepreneur develop a novel mechanism for forwarding traffic through the Internet? That would be nearly impossible today. Is it reasonable for a startup to produce a new way of providing hospitality-based networking access in airports, coffee shops, and malls? Perhaps, but it would be required to run across network vendor equipment such as *wireless access points* (APs) and access switches and routers that are themselves closed systems. The more that the software and hardware components of networking are commoditized, the lower their cost to customers. SDN promises both the hardware commoditization as well as openness, and both of these factors contribute to innovation.

To be fair, though, one must keep in mind that innovation is also driven by the prospect of generating wealth. It would be naïve to imagine that a world of low-cost networking products will have a purely positive impact on the pace of innovation. For some companies, the lower product margins presaged by SDN will reduce their willingness to invest in innovation. We examine this correlation and other business ramifications of SDN in Chapter 12.

2.4 Data Center Innovation

In Section 1.3 we explained that in the last few years, server virtualization has caused both the capacity and the efficiency of data centers to increase exponentially. This unbounded growth has made possible new computing trends such as the cloud, which is capable of holding massive amounts of computing power and storage capacity. The whole landscape of computing has changed as a result of these technological advances in the areas of compute and storage virtualization.

2.4.1 Compute and Storage Virtualization

Virtualization technology has been around for decades. The first commercially available VM technology for IBM mainframes was released in 1972 [7], complete with the *hypervisor* and the ability to abstract the hardware below, allowing multiple heterogeneous instances of other operating systems to run above it in their own space. In 1998 VMware was established and began to deliver software for virtualizing desktops as well as servers.

Use of this *compute virtualization* technology did not explode until data centers became prevalent and the need to dynamically create and tear down servers, as well as moving them from one physical server to another, became important. Once this occurred, however, the state of data center operations immediately changed. Servers could be instantiated with a mouse click and could be moved without significantly disrupting the operation of the server being moved.

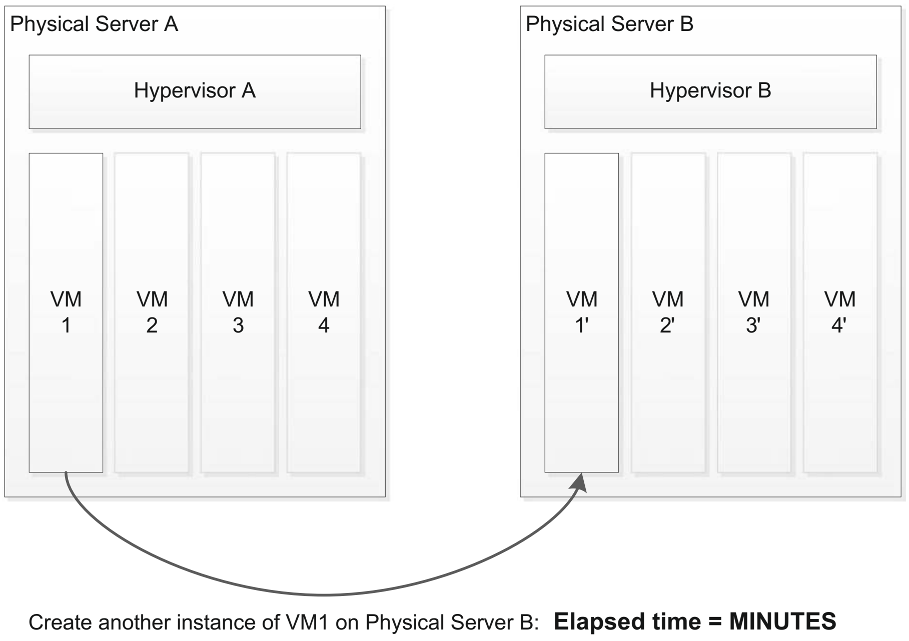


FIGURE 2.3

Server virtualization: creating a new VM instance.

Creating a new VM or moving a VM from one physical server to another is a straightforward process from a server administrator's perspective and may be accomplished very rapidly. Virtualization software, such as VMware, Hyper-V, KVM, and Citrix, are examples of products that allow server administrators to readily create and move virtual machines. These tools have reduced the time needed to start up a new instance of a server to a matter of minutes or even seconds. Figure 2.3 shows the simple creation of a new instance of a virtual machine on a different physical server.

Likewise, *storage virtualization* has existed for quite some time, as has the concept of abstracting storage blocks and allowing them to be separated from the actual physical storage hardware. As with servers, this achieves efficiency in terms of speed (e.g., moving frequently used data to a faster device) as well as in terms of utilization (e.g., allowing multiple servers to share the same physical storage device).

These technological advancements allow servers and storage to be manipulated quickly and efficiently. Although these advances in computer and storage virtualization have been taking place, the same has not been true in the networking domain [8].

2.4.2 Inadequacies in Networks Today

In Chapter 1 we discussed the evolution of networks that allowed them to survive catastrophic events such as outages and hardware or software failures. In large part, networks and networking devices have been designed to overcome these rare but severe challenges. However, with the advent of data centers, there is a growing need for networks to not only recover from these types of events but also to be able to respond quickly to frequent and immediate changes.

Although the tasks of creating a new network, moving a new network, and removing a network are similar to those performed for servers and storage, doing so requires work orders, coordination between server and networking administrators, physical or logical coordination of links, *network interface cards* (NICs), and ToR switches, to name a few. Figure 2.4 illustrates the elapsed time in creating a new instance of a VM, which is on the order of minutes, compared to the multiple days that it may take to

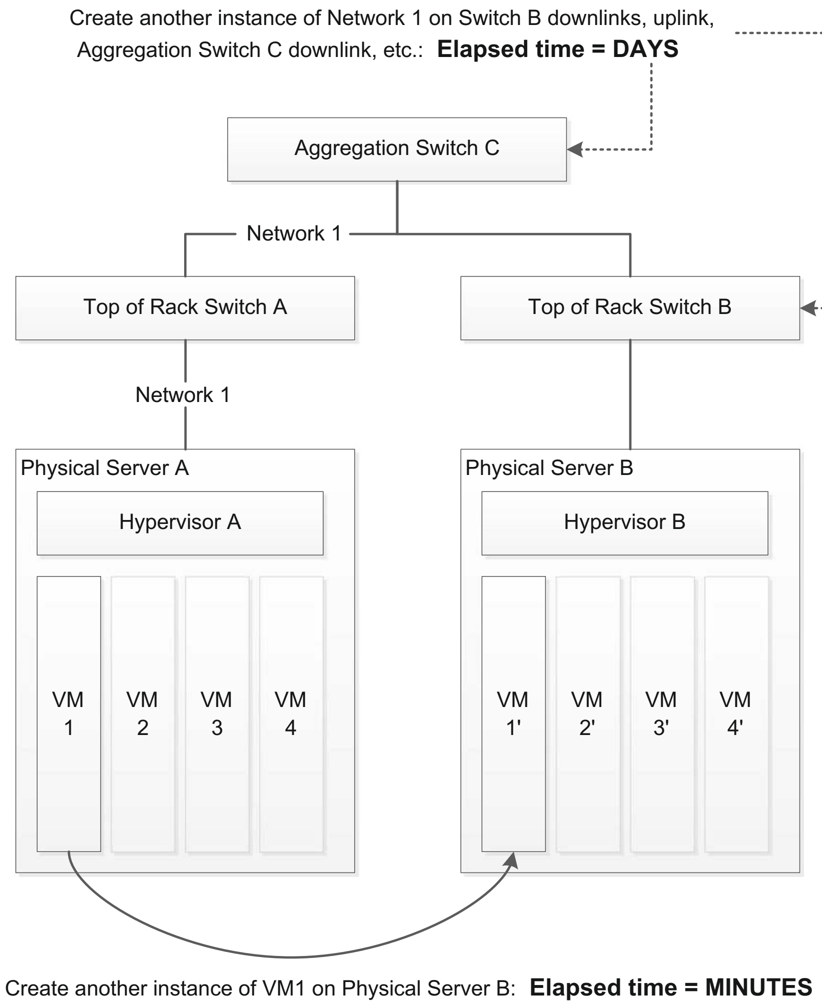

Create another instance of Network 1 on Switch B downlinks, uplink, Aggregation Switch C downlink, etc.: **Elapsed time = DAYS**
Aggregation Switch C
Network 1
Top of Rack Switch A
Top of Rack Switch B
Network 1
Physical Server A
Hypervisor A
VM 1
VM 2
VM 3
VM 4
Physical Server B
Hypervisor B
VM 1'
VM 2'
VM 3'
VM 4'
Create another instance of VM1 on Physical Server B: **Elapsed time = MINUTES**

FIGURE 2.4

Creating a new network instance in the old paradigm.

create a new instance of a network. This is because the servers are virtualized, yet the network is still purely a physical network. Even when we are configuring something virtual for the network, such as a VLAN, making changes is more cumbersome than in their server counterparts. In Chapter 1 we explained that although the control plane of legacy networks had sophisticated ways of autonomously and dynamically distributing layer two and layer three states, no corresponding protocols exist for distributing the policies that are used in policy-based routing. Thus, configuring security policy, such as ACLs or virtualization policy such as to which VLAN a host belongs, remains static and manual in traditional networks. Therefore, the task of reconfiguring a network in a modern data center does not take minutes but rather days. Such inflexible networks are hindering IT administrators in their attempts to automate and streamline their virtualized data center environments. SDN holds the promise that the time required for such network reconfiguration be reduced to the order of minutes, such as is already the case for reconfiguration of VMs.

2.5 Data Center Needs

The explosion of the size and speed of data centers has strained the capabilities of traditional networking technologies. We discuss these needs briefly here and cover them in greater detail in Chapter 7. This section serves as an indication of new requirements emerging from the technological advances taking place now in data center environments.

2.5.1 Automation

Automation allows networks to come and go at will, following the movements of servers and storage as needs change. This is sometimes referred to as *agility*—the ability to dynamically instantiate networks and to disable them when they are no longer needed. This must happen fast, efficiently, and with a minimum of human intervention. Not only do networks come and go—they also tend to expand and contract. Clearly, it may require a dynamic, agile, and automated network to keep pace with these changes.

2.5.2 Scalability

With data centers and cloud environments, the sheer number of end stations that connect to a single network has grown exponentially. The limitations of MAC address table sizes and number of VLANs have become impediments to network installations and deployments. The large number of physical devices present in the data centers also poses a *broadcast control* problem. The use of tunnels and virtual networks can contain the number of devices in a broadcast domain to a reasonable number.

2.5.3 Multipathing

Accompanying the large demands placed on the network by the scalability requirement we've stated is the need for the network to be efficient and reliable. That is, the network must make optimal use of its resources, and it must be resistant to failures of any kind; and, if failures do occur, the network must be able to recover immediately. Figure 2.5 shows a simple example of multipath through a network, both for choosing the shortest path as well as for alternate or redundant paths. Legacy layer two control plane software would block some of the alternate and redundant paths shown in the figure in order to eliminate

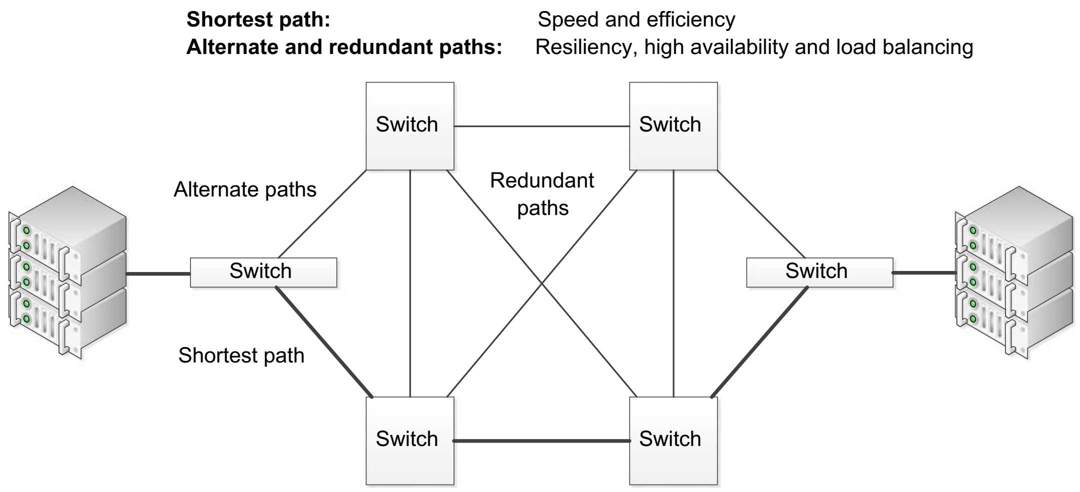

Shortest path: Speed and efficiency
Alternate and redundant paths: Resiliency, high availability and load balancing
Switch
Switch
Alternate paths
Redundant paths
Switch
Switch
Shortest path
Switch
Switch

FIGURE 2.5

Multipath.

forwarding loops. Because we are living with network technology invented years ago, the network is forced into a hierarchy that results in links that could have provided shortest-path routes between nodes lying entirely unused and dormant. In cases of failure, the current hierarchy can reconfigure itself in a nondeterministic manner and with unacceptable latency. The speed and high-availability requirements of the modern data center mandate that multiple paths not be wasted by being blocked and, instead, be put into use to improve efficiency as well as to achieve resiliency and load balancing.

2.5.4 Multitenancy

With the advances in data center technology described above and the subsequent advent of *cloud computing*, the idea of hosting dozens or even hundreds or thousands of customers, or *tenants*, in the same physical data center has become a requirement. One set of physical hardware hosting multiple tenants has been feasible for some time in the server and storage area. Multitenancy implies that the data center has to provide each of its multiple tenants with its own (virtual) network that it can manage in a manner similar to the way it would manage a physical network.

2.5.5 Network Virtualization

The urgency for automation, multitenancy, and multipathing has increased as a result of the scale and fluidity introduced by server and storage virtualization. The general idea of virtualization is that you create a higher-level abstraction that runs on top of the actual physical entity you are abstracting. The growth of compute and storage server virtualization has created demand for network virtualization. This means having a virtual abstraction of a network running on top of the actual physical network. With virtualization, the network administrator should be able to create a network anytime and anywhere he chooses, as well as expanding and contracting networks that already exist. Intelligent virtualization software should be capable of this task without requiring the upper virtualized layer to be aware of what is occurring at the physical layer.

Server virtualization has caused the scale of networks to increase as well, and this increased scale has put pressure on layer two and layer three networks as they exist today. Some of these pressures can be alleviated to some degree by tunnels and other types of technologies, but fundamental network issues remain, even in those situations. Consequently, the degree of network virtualization required to keep pace with data center expansion and innovation is not possible with the network technology that is available today.

To summarize, advances in data center technology have caused weaknesses in the current networking technology to become more apparent. This situation has spurred demand for better ways to construct and manage networks [9], and that demand has driven innovation around SDN [10].

2.6 Conclusion

The issues of reducing cost and the speed of innovation as motivators for SDN will be recurring themes in the balance of this work. The needs of the modern data center are so tightly linked with the demand for SDN that we dedicate all of Chapter 7 to the examination of use cases in the data center that benefit from SDN technology. These needs did not appear overnight, nor did SDN simply explode onto the networking scene in 2009, however. There were a number of tentative steps over many years that formed the basis for what ultimately appeared as the SDN revolution. In the next chapter we review this evolution and examine how it culminated in the birth of what we now call SDN. We also discuss the context in which SDN has matured and the organizations and forces that continue to mold its future.

References

[1] McKeown N, Parulkar G, Anderson T, Balakrishnan H, Peterson L, Rexford J, et al. OpenFlow: enabling innovation in campus networks. ACM SIGCOMM Comput Commun Rev 2008;38(2).

[2] Kirkpatrick K. software-defined networking. Commun ACM 2013;56(9).

[3] Malim G. SDN's value is in operational efficiency, not capex control. Global Telecoms Business; June 2013. Retrieved from <www.globaltelecomsbusiness.com/article/3225124/SDNs-value-is-in-operational-efficiency-not-capex-control.html>.

[4] Wilson C. NTT reaping opex rewards of SDN; August 2013. Retrieved from <www.lightreading.com/carrier-sdn/ntt-reaping-opex-rewards-of-sdn/d/d-id/705306>.

[5] Yegulalp S. Five SDN benefits enterprises should consider. Network computing; July 2013. Retrieved from <www.networkcomputing.com/next-generation-data-center/commentary/networking/five-sdn-benefits-enterprises-should-con/240158206>.

[6] IEEE standard for local and metropolitan area networks: media access control (MAC) bridges. IEEE 802.1D-2004, New York, USA; June 2004.

[7] Witner B, Wade B. Basics of z/VM virtualization, IBM. Retrieved from <www.vm.ibm.com/devpages/bkw/vmbasics.pdf>.

[8] Bari MF, Boutaba R, Esteves R, Granville LZ, Podlesny M, Rabbani MG, et al. Data center network virtualization: a survey. IEEE Commun Surv Tutor 2013;15(2).

[9] Narten T, Gray E, Black D, Fang L, Kreeger L, Napierala M. Problem statement: overlays for network virtualization. Internet draft, Internet Engineering Task Force; July 31, 2013.
[10] Brodkin J. Data center startups emerging to solve virtualization and cloud problems. Network world; June 14, 2011. Retrieved from <www.networkworld.com/news/2011/061411-data-center-startups.html>.
[11] Greenberg A, Hjalmtysson G, Maltz D, Myers A, Rexford J, Xie G, et al. A clean-slate 4D approach to network control and management. ACM SIGCOMM Comput Commun Rev 2005;35(3).
[12] Shenker S. The future of networking and the past of protocols. Open Networking Summit. Palo Alto, CA, USA: Stanford University; October 2011.
[13] Kim H, Feamster N. Improving network management with software-defined networking. IEEE Commun Mag 2013;51(2).

This page is intentionally left blank

CHAPTER

The Genesis of SDN 3

We have now seen a number of the reasons for the emergence of SDN, including the fact that there were too many disparate control plane protocols attempting to solve many different networking problems in a distributed fashion. This has resulted in relative confusion and inertia in network switching evolution as compared to compute and storage technologies. Overall, the system that has evolved is far too closed, and there is a strong desire to migrate to a model closer to that of open source development and open source initiatives (similar to Linux with respect to computer operating systems). We have also noted that for a variety of reasons, the *network equipment manufacturers* (NEMs) are bogged down in their current technologies and are unable to innovate at the rate required by the modern data center. This fact is partly attributable to the fact that the current business model for networking equipment is a very high-margin business, and it is not obviously in the incumbents' interest for the status quo to change drastically [21]. Even the standards bodies generally move too slowly [22]. Thus, the need for an SDN-like transformative technology has become apparent to many users of network equipment, especially in large data centers. In this chapter we explore how the SDN movement began. To understand the base from which these SDN pioneers started, we begin with a brief review of the evolution of networking technology up to the advent of SDN.

3.1 The Evolution of Networking Technology

In the earliest days of computing there was no networking; each individual computer occupied an entire room or even an entire floor. As mainframes emerged and began to find their way into our technological consciousness, these computing powerhouses still operated as islands, and any data sharing took place using physical media such as magnetic tapes.

3.1.1 Mainframe Networking: Remote Terminals

Even in the age of mainframes, remote connectivity to the mainframe was needed. This was provided in the form of remote terminal controllers and card readers, which operated as subservient devices known as *peripherals*, with control residing entirely in the central mainframe. Network connections in this case were simple point-to-point or point-to-multipoint links. Communication was solely between a few connected entities, in this case the mainframe and a small set of remote terminal controllers or card readers. This communication was entirely synchronous, with the peripherals allowed to transmit only when polled by the mainframe.

3.1.2 Peer-to-Peer, Point-to-Point Connections

As computer technology began to move from solely mainframes to the addition of minicomputers, these machines had a greater need to share information in a quick and efficient manner. Computer manufacturers began to create protocols for sharing data between two peer machines. The network in this case was also point-to-point, although the nature of the connection was peer-to-peer in that the two machines (e.g., minicomputers) would communicate with each other and share data as relative equals, at least compared to the earlier mainframe-to-terminal-controller type of connections.

Of course, in these point-to-point connections, the network was trivial, with only the two parties communicating with each other. Control for this communication resided not in any networking device but in the individual computers participating in this one-to-one communication.

3.1.3 Local Area Networks

Eventually, with further evolution of computing toward smaller, independent systems, the need arose for a way to connect these devices in order to allow them to share information and collaborate in a manner that wasn't required when everything ran on one large mainframe or even on a powerful minicomputer. Hence, *local area networking* (LAN) technology arrived, with various battles being fought between technologies (e.g., Ethernet/IEEE 802.3 versus Token Ring/IEEE802.5). Notably, these early LANs were running on shared media, so all traffic was seen by every device attached to the network.

IEEE 802.3 emerged as the more popular of these technologies. It uses *Carrier-Sense Multiple-Access/Collision Detect* (CSMA/CD) technology, which exhibits poor aggregate throughput when the number of active devices reaches a certain level. The exact number of devices where this would occur is dependent on the amount of data each device attempts to transmit. This decrease in performance resulted from each device backing off and waiting to transmit when a collision occurs, as stipulated by CSMA/CD. The number of collisions reaches a critical point as a function of the number of nodes and their respective transmission activity. Once this critical point is reached, the network becomes very inefficient, with too much time spent either in the midst of or recovering from collisions.

These flat, shared-media networks were quite simple, and the *repeater* devices provided physical extensions to the shared media by merely forwarding all frames to the extended medium. The greatest impact of these early devices was a new topology created by wire concentration. This made layer two networks more deployable and reliable than the former method of snaking a coaxial cable from one node to the next. This early networking required minimum control plane intelligence, if any. Simple repeaters basically just did one thing: They forwarded to everybody. More advanced, *managed* repeaters did have limited control planes where segmentation and error monitoring occurred. Managed repeaters did perform functions such as removing erroneous traffic from the network, as well as removing isolated ports that were causing problems. Segmentable repeaters were able to split themselves into multiple repeaters via configuration. Different groups of ports could be configured to reside in different collision domains. These separate collision domains would be connected by bridges. This feature provided more control over the size of the collision domains to optimize performance.

3.1.4 Bridged Networks

Eventually these shared-media networks needed to scale in physical extent as well as number of nodes. As explained in the previous section, shared-media networks do not scale well as the number of hosts grows.

It became desirable to split the shared-media network into separate segments. In so doing, and since not all the nodes are transmitting all the time, *spatial reuse* occurs and the aggregate available bandwith actually increases due to the locality of transmissions in each segment. The first devices to perform this functionality were called *bridges*, forerunners of today's switches but much simpler. They typically had only two ports connecting two shared domains. These bridges possessed an incipient control plane in that they were able to actually learn the location and MAC address of all devices and create forwarding tables that allowed them to make decisions about what to do with incoming packets. As we mentioned in the previous chapter, these forwarding tables were implemented entirely in software.

Furthermore, these bridges were implemented in such a way that each device was able to operate independently and autonomously, without requiring any centralized intelligence. The goal was to facilitate expansion of the network without a lot of coordination or interruption across all the devices in the network. One of the first manifestations of this distributed intelligence paradigm was the *Spanning Tree Protocol* (STP), which allowed a group of bridges to interact with each other and converge on a topology decision (the spanning tree), which eliminated loops and superimposed a hierarchical loop-free structure on the network.

The important point here is that this greater scale in terms of the number of nodes as well as physical extent drove the need for this new model of individually autonomous devices, with distributed protocols implementing the required control functionality. If we translate this period's realities into today's terminology, there was no centralized controller, merely a collector of statistics. *Policies*, if indeed one can use that term, were administered by setting specific parameters on each device in the network. We need to keep in mind that at the time, networks in question were small, and this solution was entirely acceptable.

3.1.5 Routed Networks

In the same manner that bridged and switched networks dealt with layer two domains with distributed protocols and intelligence, similar strategies were employed for layer three routing. Routers were directly connected locally to layer two domains and interconnected over large distances with point-to-point WAN links. Distributed routing protocols were developed to allow groups of interconnected routers to share information about those networks to which they were directly connected. By sharing this information among all the routers, each was able to construct a routing table, allowing it to route packets to remote layer three IP subnets using the optimal forwarding ports. This was another application of autonomous devices, utilizing distributed protocols to allow each to make appropriate forwarding decisions.

This sequence has led to the current state of affairs, with networking intelligence distributed in the networking devices themselves. During this evolution, however, the growth of network size and complexity was unrelenting. The size of MAC forwarding tables grew, control plane protocols became more complex, and network overlays and tunneling technology became more prevalent. Making major changes to these implementations was a continual challenge. Since the devices were designed to operate independently, centrally administered, large-scale upgrades were challenging. In addition, the fact that the actual control plane implementations were from many different sources and not perfectly matched created a sort of *lowest common denominator* effect, where only those features that were perfectly aligned between the varied implementations could truly be relied upon. In short, existing solutions were not scaling well with this growth and complexity. This situation led network engineers and researchers to question whether this evolution was headed in the right direction. In the following section we describe some of the innovations and research that resulted.

3.2 Forerunners of SDN

Prior to OpenFlow, and certainly prior to the birth of the term Software Defined Networking, forward-thinking researchers and technologists were considering fundamental changes to today's world of autonomous, independent devices and distributed networking intelligence and control. This section considers some of those early explorations of SDN-like technology. There is a steady progression of ideas around advancing networking technology toward what we now know as SDN. Table 3.1 shows this progression, which is discussed in more detail in the subsections that follow. The timeframes shown in the table represent approximate time periods when these respective technologies were developed or applied in the manner described.

3.2.1 Early Efforts

A good survey of early programmable networks that were stepping-stones on the path to SDN can be found in [12]. Some of the earliest work in programmable networks began not with Internet routers and switches but with ATM switches. Two notable examples were *Devolved Control of ATM Networks* (DCAN) and *open signaling*.

As its name indicates, DCAN [8] prescribed the separation of the control and management of the ATM switches from the switches themselves. This control would be assumed by an external device that is similar to the role of the controller in SDN networks.

Table 3.1 Precursors of SDN

Project	Description
Open signaling	Separating the forwarding and control planes in ATM switching (1999; see Section 3.2.1)
Active networking	Separating control and programmable switches (late 1990s; see Section 3.2.1)
DCAN	Separating the forwarding and control planes in ATM switching (1997; see Section 3.2.1)
IP switching	Controling layer two switches as a layer three routing fabric (late 1990s; see Section 3.2.1)
MPLS	Separating control software, establishing semi-static forwarding paths for flows in traditional routers (late 1990s; see Section 3.2.1)
RADIUS, COPS	Using admission control to dynamically provision policy (2010; see Section 3.2.2)
Orchestration	Using SNMP and CLI to help automate configuration of networking equipment (2008; see Section 3.2.3)
Virtualization Manager	Using plug-ins to perform network reconfiguration to support server virtualization (2011; see Section 3.2.4)
ForCES	Separating the forwarding and control planes (2003; see Section 3.2.5)
4D	Locating control plane intelligence in a centralized system (2005; see Section 3.2.6)
Ethane	Achieving complete enterprise and network access and control using separate forwarding and control planes and utilizing a centralized controller (2007; see Section 3.2.7)

Open signaling [6] proposed a set of open, programmable interfaces to the ATM switching hardware. The key concept was to separate the control software from the switching hardware. This work led to the IETF effort that resulted in the creaton of the *General Switch Management Protocol* (GSMP) [7]. In GSMP, a centralized controller is able to establish and release connections on an ATM switch as well as a multitude of other functions that may otherwise be achieved via distributed protocols on a traditional router. Tag switching was Cisco's version of label switching, and both of these terms refer to the technology that ultimately became known as *Multiprotocol Label Switching* (MPLS). Indeed, MPLS and related technologies are a deviation from the autonomous, distributed forwarding decisions characteristic of the traditional Internet router, and in that sense they were a small step toward a more SDN-like Internet switching paradigm. In the late 1990s, Ipsilon Networks utilized the GSMP protocol to set up and tear down ATM connections internally in the company's IP Switch product. The Ipsilon IP Switch [13] presented normal Internet router interfaces externally, but its internal switching fabric could utilize ATM switches for persistent flows. Flows were defined as a relatively persistent set of packets between the same two endpoints, where an endpoint was determined by IP address and TCP/UDP port number. Since there was some overhead in establishing the ATM connection that would carry that flow, this additional effort would be expended only if the IP Switch believed that a relatively large number of packets would be exchanged between the endpoints in a short period of time. This definition of flow is somewhat consistent with the notion of flow within SDN networks.

The *active networking* project [10,11] also included the concept of switches that could be programmed by out-of-band management protocols. In the case of active networking, the switching hardware was not ATM switches but Internet routers. Active networking also included a very novel proposal for small downloadable programs called *capsules* that would travel in packets to the routers and could reprogram the router's behavior on the fly. These programs could be so fine-grained as to prescribe the forwarding rules for a single packet, even possibly the payload of the packet that contained the program.

3.2.2 Network Access Control

Network access control (NAC) products control access to a network based on policies established by the network administration. The most basic control is to determine whether or not to admit the user onto the network. This decision is usually accomplished by some exchange of credentials between the user and the network. If the user is admitted, his rights of access will also be determined and restricted in accordance with those policies. In some early efforts, network policy beyond admission control was dynamically provisioned via NAC methods, such as *Remote Authentication Dial-In User Service* (RADIUS) [16] and *Common Open Policy Service* (COPS) [14].

RADIUS has been used to provide the automatic reconfiguration of the network. This solution was somewhat forward-looking in that RADIUS can be viewed as a precursor to SDN. The idea is that via RADIUS, networking attributes would change based on the identity of the compute resource that had just appeared and required connectivity to the network. Whereas RADIUS was originally designed for the *authentication, authorization, and accounting* (AAA) processes related to granting a user access to a network, that original paradigm maps well to our network reconfiguration problem. The identity of the resource connecting to the network serves to identify the resource to the RADIUS database, and the authorization attributes returned from that database could be used to change the networking attributes described above. Solutions of this nature achieved automatic reconfiguration of the network but never

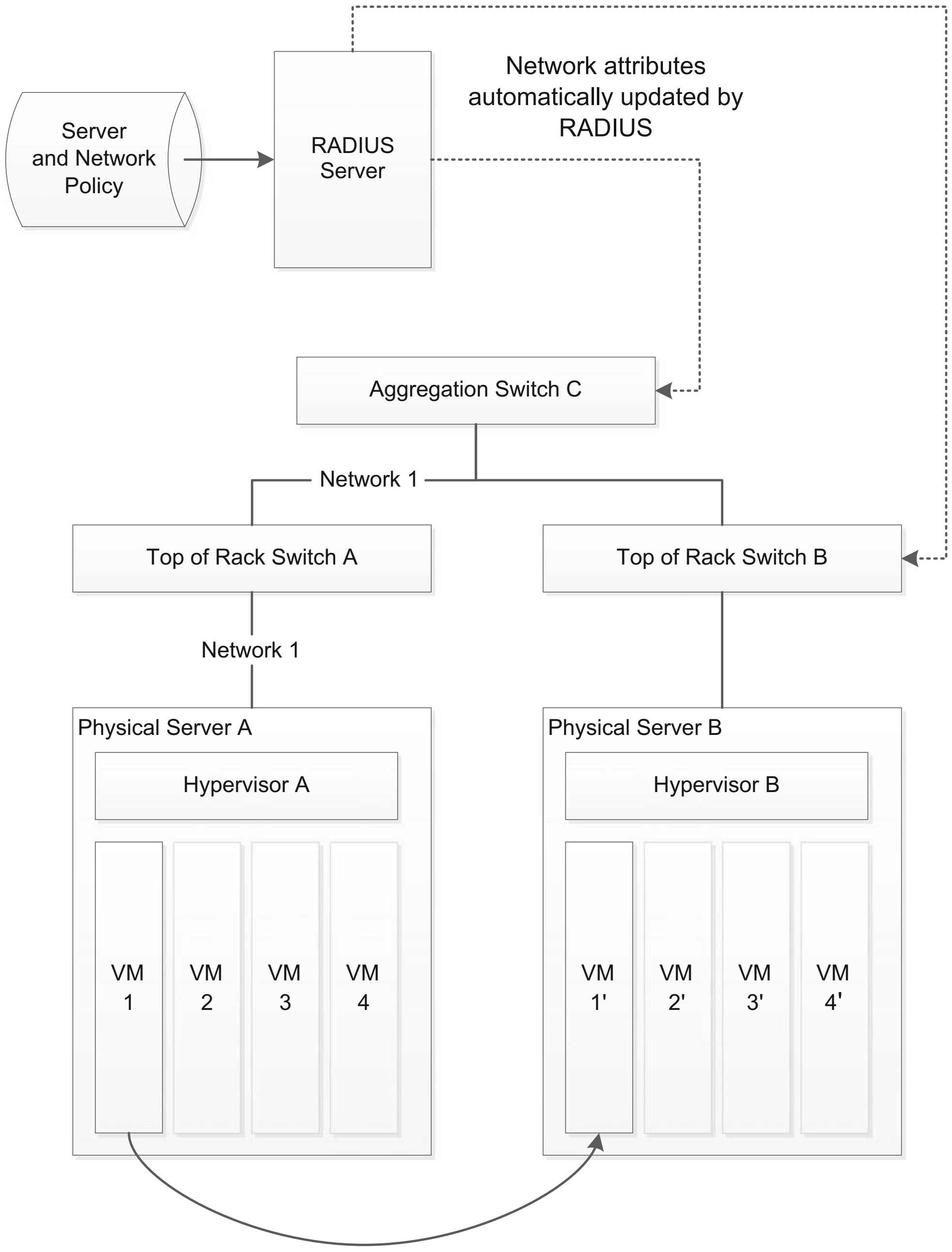


FIGURE 3.1

Early attempts at SDN: RADIUS.

gained the full trust of IT administrators and, thus, never became mainstream. Although this RADIUS solution did an adequate job of automatically reconfiguring the edge of the network, the static, manually configured core of the network remained the same. This problem still awaited a solution.

Figure 3.1 shows an overview of the layout and operation of a RADIUS-based solution for automating the process of creating network connectivity for a virtual server. In the figure, the process would be that a VM is moved from physical server A to physical server B, and as a result the RADIUS server becomes aware of the presence of this VM at a new location. The RADIUS server is then able to automatically configure the network based on this information, using standard RADIUS mechanisms.

3.2.3 Orchestration

Early attempts at automation involved applications that were commonly labeled *orchestrators* [17]. Just as a conductor can make a harmonious whole out of the divergent instruments of an orchestra, such applications could take a generalized command or goal and apply it across a wide range of often heterogeneous devices. These orchestration applications would typically utilize common device *application programmer interfaces* (APIs) such as the *Command-Line Interface* (CLI) or *Simple Network Management Protocol* (SNMP).

Figure 3.2 shows an overview of the layout and operation of an orchestration management solution for automating the process of creating network connectivity for a virtual server. The figure shows

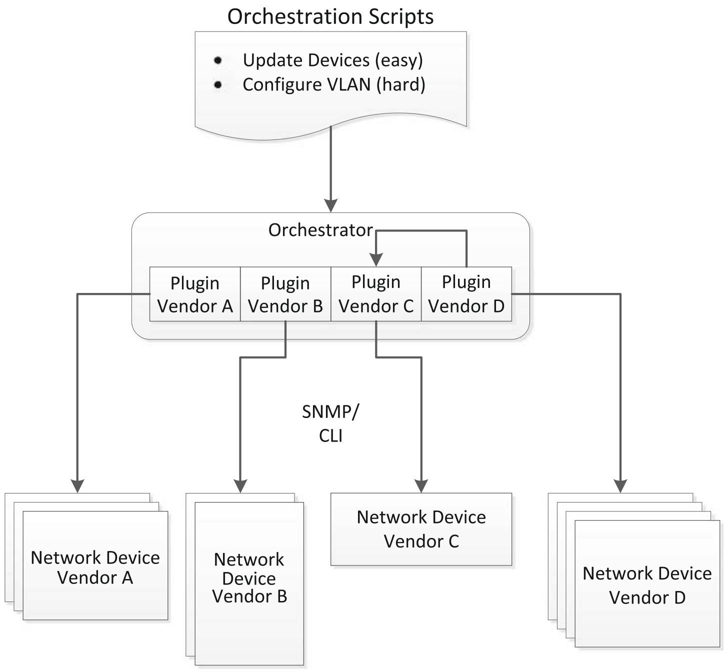


FIGURE 3.2

Early attempts at SDN: orchestration.

SNMP/CLI plug-ins for each vendor's specific type of equipment; an orchestration solution can then have certain higher-level policies that are in turn executed at lower levels by the appropriate plugins. The vendor-specific plugins are used to convert the higher-level policy requests into the corresponding native SNMP or CLI request specific to each vendor.

Such orchestration solutions alleviated the task of updating device configurations. But since they were really limited to providing a more convenient interface to existing capabilities, and since no capability existed in the legacy equipment for network-wide coordination of more complex policies such as security and virtual network management, tasks such as configuring VLANs remained hard. Consequently, orchestration management applications continued to be useful primarily for tasks such as software and firmware updates but not for tasks that would be necessary in order to automate today's data centers.

3.2.4 Virtualization Manager Network Plugins

The concept of *virtualization manager network plugins* builds on the notion of orchestration and attempts to automate the network updates that are required in a virtualization environment. Tools specifically targeted at the data center would often involve virtualization manager plugins (e.g., plugins for VMware's vCenter [18]), which would be configured to take action in the event of a server change, such as a vMotion [19]. The plugins would then take the appropriate actions on the networking devices they controlled in order to make the network follow the server and storage changes with changes of its own. Generally, the mechanism for making changes to the network devices would be SNMP or CLI commands. These plugins can be made to work, but since they must use the static configuration capabilities of SNMP and CLI, they suffer from being difficult to manage and prone to error.

Figure 3.3 shows an overview of the layout and operation of a virtualization manager plugin solution for automating the process of creating network connectivity for a virtual server. This figure reflects a similar use of SNMP/CLI plugins to that which we saw in Figure 3.2. The most notable difference is that this application starts from a base that supports virtualization. The nature of the reconfiguration managed by this application is oriented to support the networking of virtual machines via VLANs and tunnels.

3.2.5 ForCES: Separation of Forwarding and Control Planes

The *Forwarding and Control Element Separation* (ForCES) [9] work produced in the IETF began around 2003. ForCES was one of the original proposals recommending the decoupling of forwarding and control planes. The general idea of ForCES was to provide simple hardware-based forwarding entities at the foundation of a network device and software-based control elements above. These simple hardware forwarders were constructed using cell-switching or tag-switching technology. The software-based control had responsibility for the broader tasks, often involving coordination between multiple network devices (e.g., BGP routing updates).

The functional components of ForCES are as follows:

- **Forwarding Element.** The *Forwarding Element* (FE) would be typically implemented in hardware and located in the network. The FE is responsible for enforcement of the forwarding and filtering rules that it receives from the *controller*.
- **Control Element.** The *Control Element* (CE) is concerned with the coordination between the individual devices in the network and for communication forwarding and routing information to the FEs below.

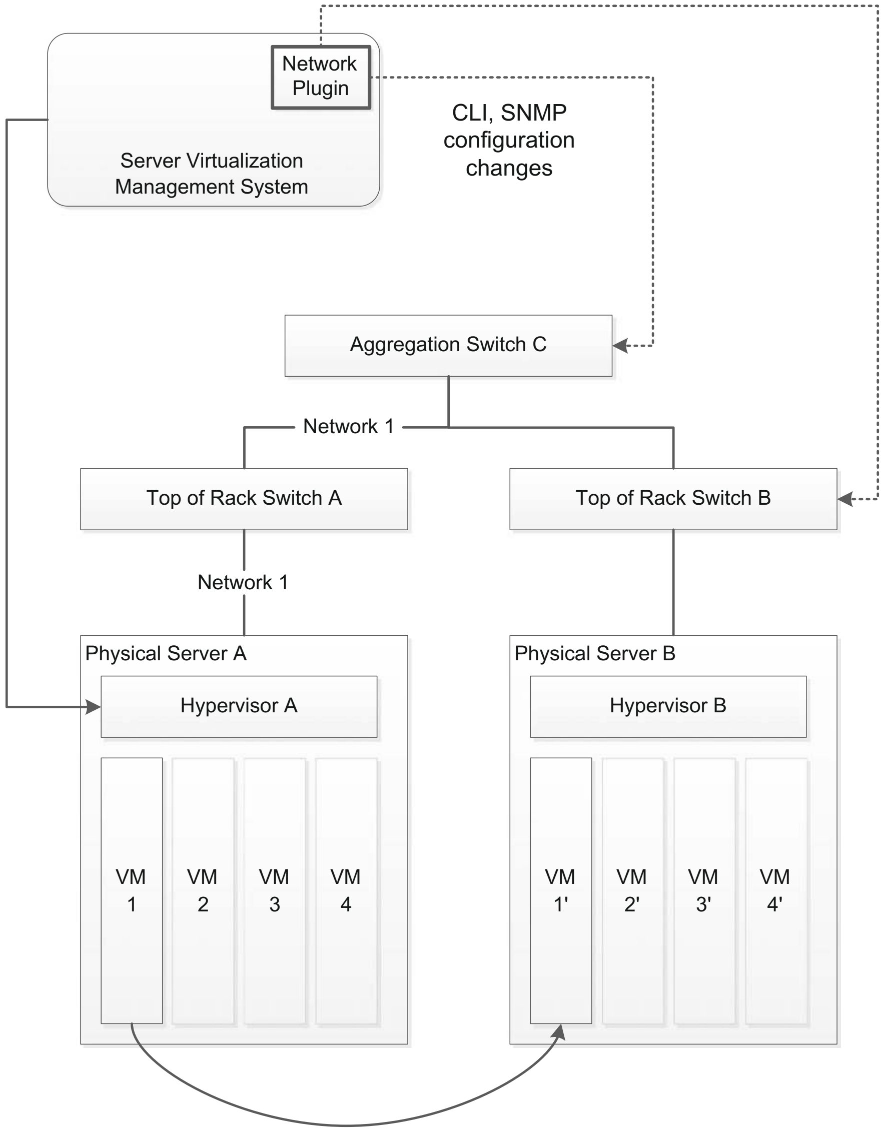


FIGURE 3.3

Early attempts at SDN: plugins.

- **Network Element.** The *Network Element* (NE) is the actual network device that consists of one or more FEs and one or more CEs.
- **ForCES protocol.** The ForCES protocol is used to communicate information back and forth between FEs and CEs.

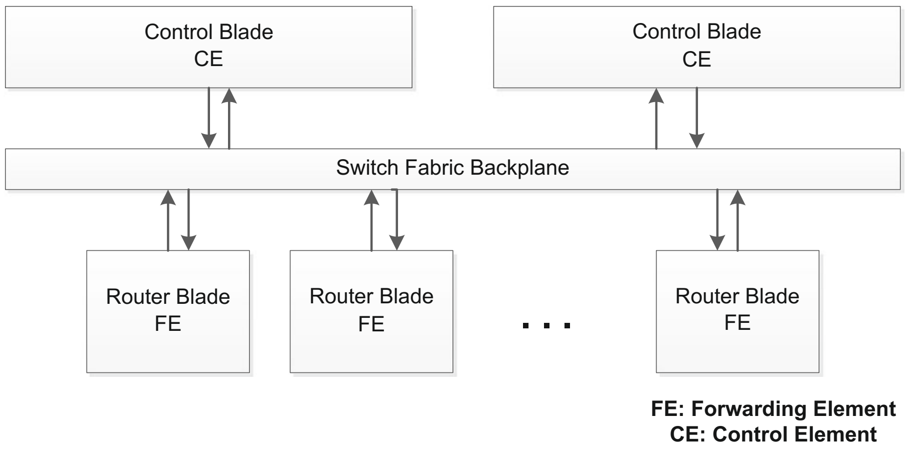


FIGURE 3.4

ForCES design.

ForCES proposes the separation of the forwarding plane from the control plane, and it suggests two different embodiments of this architecture. In one of these embodiments, both the forwarding and control elements are located within the networking device. The other embodiment speculates that it would be possible to actually move the control element(s) off the device and to locate them on an entirely different system. Although the suggestion of a separate controller thus exists in ForCES, the emphasis is on the communication between CE and FE over a switch backplane, as shown in Figure 3.4.

3.2.6 4D: Centralized Network Control

Seminal work on the topic of moving networking technology from distributed networking elements into a centralized controller appeared in the 4D proposal [2] *A Clean Slate 4D Approach to Network Control and Management*. 4D, named after the architecture's four planes—*decision*, *dissemination*, *discovery*, and *data*—proposes a complete refactoring of networking away from autonomous devices and toward the idea of concentrating control plane operation in a separate and independent system dedicated to that purpose.

4D argues that the state of networking today is fragile and, therefore, often teeters on the edge of failure because of its current design based on distributed, autonomous systems. Such systems exhibit a defining characteristic of unstable, complex systems: a small local event such as a misconfiguration of a routing protocol can have a severe, global impact. The proposal argues that the root cause is the fact that the control plane is running on the network elements themselves.

4D centers around three design principles:

- **Network-level objectives.** In short, the goals and objectives for the network system should be stated in network-level terms based on the entire network domain, separate from the network elements, rather than in terms related to individual network device performance.

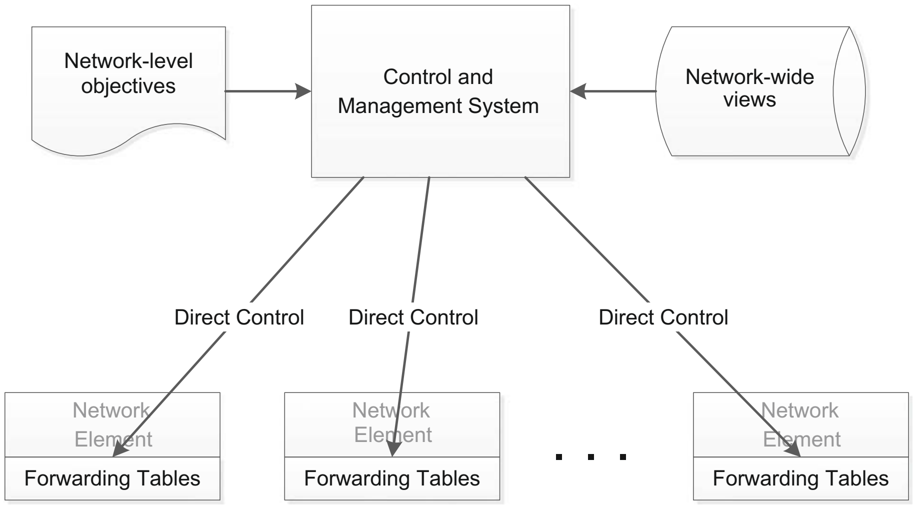


FIGURE 3.5

4D principles.

- **Network-wide view.** There should be a comprehensive understanding of the whole network. Topology, traffic, and events from the entire system should form the basis on which decisions are made and actions are taken.
- **Direct control.** The control and management systems should be able to exert direct control over the networking elements, with the ability to program the forwarding tables for each device rather than only being able to manipulate some remote and individual configuration parameters, as is the case today.

Figure 3.5 shows the general architecture of a 4D solution, with centralized network control via the control and management system.

One aspect of 4D that is actually stated in the title of the paper is the concept of a *clean slate*, meaning the abandoning of the current manner of networking in favor of this new method, as described by the three aforementioned principles. A quote from the 4D proposal states that "We hope that exploring an extreme design point (the clean-slate approach) will help focus the attention of the research and industrial communities on this crucially important and intellectually challenging area."

The 4D proposal delineates some of the challenges faced by the proposed centralized architecture. These challenges continue to be relevant today in SDN. We list a few of them here:

- **Latency.** Having a centralized controller means that a certain (hopefully small) number of decisions will suffer nontrivial round-trip latency as the networking element requests policy directions from the controller. The way this delay impacts the operation of the network, and to what extent, remains to be determined. Furthermore, with the central controller providing policy advice for a number of network devices, it is unknown whether the conventional servers on which the controller runs will be able to service these requests at sufficient speed to have minimal or no impact on network operation.

- **Scale.** Having a centralized controller means that responsibility for the topological organization of the network, determination of optimal paths, and responses to changes must be handled by the controller. As has been argued, this is the appropriate location for this functionality; however, as more and more network devices are added to the network, questions arise of scale and the ability of a single controller to handle all those devices. It is difficult to know how well a centralized system can handle hundreds, thousands, or tens of thousands of network devices and to know what is the solution when the number of network devices outgrows the capacity of the controller to handle them. If we attempt to scale by adding controllers, how do they communicate, and who orchestrates coordination among the controllers? Section 6.1.3 addresses these questions.
- **High availability (HA).** The centralized controller must not constitute a *single point of failure* for the network. This implies the need for redundancy schemes in a number of areas. First, there must be redundant controllers such that processing power is available in the event of failure of a single controller. Second, the actual data used by the set of controllers needs to be mirrored such that the controllers can program the network devices in a consistent fashion. Third, the communications paths to the various controllers need to be redundant to ensure that there is always a functioning communications path between a switch and at least one controller. We further discuss high availability in the context of a modern SDN network in Section 6.1.2.
- **Security.** Having a centralized controller means that security attacks are able to focus on that one point of failure, and thus the possibility exists that this type of solution is more vulnerable to attack than a more distributed system. It is important to consider what extra steps must be taken to protect both the centralized controller and the communication channels between it and the networking devices.

ForCES and 4D contributed greatly to the evolution of the concepts that underlie SDN: separation of forwarding and control planes (ForCES) and having a centralized controller responsible for overall routing and forwarding decisions (4D). However, both of these proposals suffer from a lack of actual implementations. Ethane, examined in the following section, benefited from the experience that can only come from a real-life implementation.

3.2.7 Ethane: Controller-Based Network Policy

Ethane was introduced in a paper titled *Rethinking Enterprise Network Control* [1] in 2007. Ethane is a policy-based solution that allows network administrators to define policies pertaining to network-level access for users, which includes authentication and quarantine for misbehaving users. Ethane was taken beyond the proposal phase. Multiple instances have been implemented and shown to behave as suggested in [1].

Ethane is built around three fundamental principles:

1. **The network should be governed by high-level policies.** Similar to 4D, Ethane espouses the idea that the network be governed by policies defined at high levels rather than on a per-device basis. These policies should be at the level of services and users and the machines through which users can connect to the network.
2. **Routing for the network should be aware of these policies.** Paths that packets take through the network are to be dictated by the higher-level policies described in the previous bullet point rather than as in the case of today's networks, in which paths are chosen based on lower-level directives.

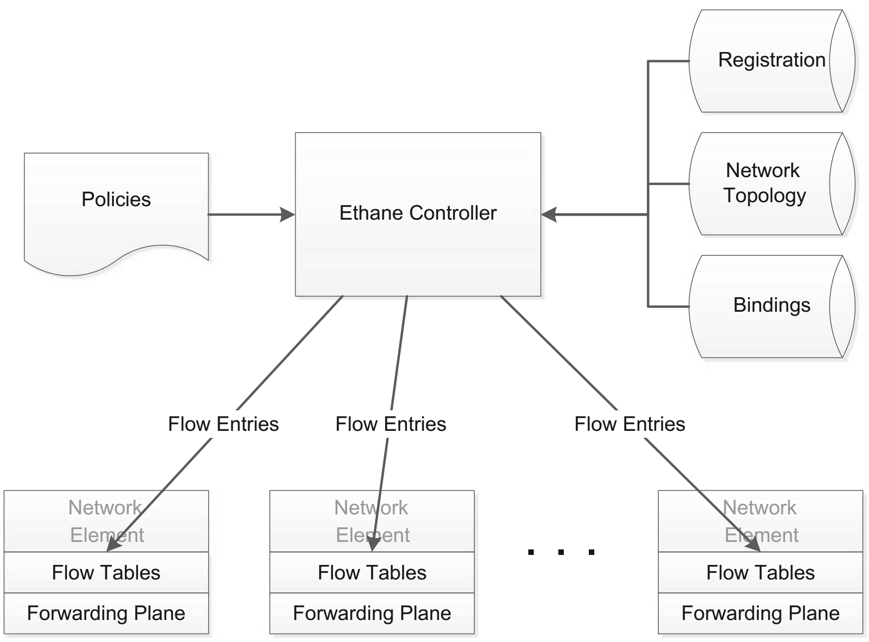


FIGURE 3.6

Ethane architecture.

For example, *guest* packets may be required to pass through a filter of some sort, or certain types of traffic may require routing across lightly loaded paths. Some traffic may be highly sensitive to packet loss; other traffic (e.g., *Voice over IP*, or VoIP) may tolerate dropped packets but not latency and delay. These higher-level policies are more powerful guidelines for organizing and directing traffic flows than are low-level and device-specific rules.

3. **The network should enforce binding between packets and their origin.** If policy decisions rely on higher-level concepts such as the concept of a *user*, then the packets circulating in the network must be traceable back to their point of origin (i.e., the user or machine that is the actual source of the packet).

Figure 3.6 illustrates the basics of the Ethane solution. As will become apparent in the following chapters, there are many similarities between this solution and OpenFlow.

To test Ethane, the researchers themselves had to develop switches in order to have them implement the protocol and the behavior of such a device—that is, a device that allows control plane functionality to be determined by an external entity (the controller) and that communicates with that external entity via a protocol that allows flow entries to be configured into its local flow tables, which then perform the forwarding functions as packets arrive.

The behavior of the Ethane switches is generally the same as today's OpenFlow switches, which forward and filter packets based on the flow tables that have been configured on the device. If the switch does not know what to do with the packet, it forwards it to the controller and awaits further instructions.

In short, Ethane is basically a Software Defined Networking technology, and its components are the antecedents of OpenFlow, which we describe in detail in Chapter 5.

3.3 Software Defined Networking is Born

3.3.1 The Birth of OpenFlow

Just as the previous sections presented standards and proposals that were precursors to SDN, seeing SDN through a gestation period, the arrival of OpenFlow is the point at which SDN was actually born. In reality, the term SDN did not come into use until a year after OpenFlow made its appearance on the scene in 2008, but the existence and adoption of OpenFlow by research communities and networking vendors marked a sea change in networking, one that we are still witnessing even now. Indeed, though the term SDN was in use in the research community as early as 2009, SDN did not begin to make a big impact in the broader networking industry until 2011.

For reasons identified in the previous chapter, OpenFlow was developed and designed to allow researchers to experiment and innovate with new protocols in everyday networks. The OpenFlow specification encouraged vendors to implement and enable OpenFlow in their switching products for deployment in college campus networks. Many network vendors have implemented OpenFlow in their products.

The OpenFlow specification delineates both the protocol to be used between the controller and the switch as well as the behavior expected of the switch. Figure 3.7 illustrates the simple architecture of an OpenFlow solution.

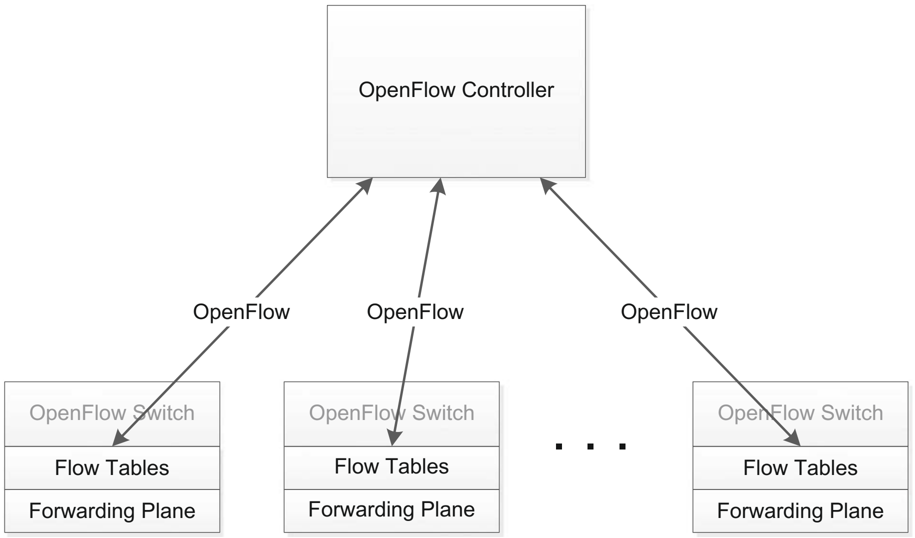


FIGURE 3.7

General OpenFlow design.

The following list describes the basic operation of an OpenFlow solution:

- The controller populates the switch with flow table entries.
- The switch evaluates incoming packets and finds a matching flow, then performs the associated action.
- If no match is found, the switch forwards the packet to the controller for instructions on how to deal with the packet.
- Typically the controller will update the switch with new flow entries as new packet patterns are received, so that the switch can deal with them locally. It is also possible that the controller will program *wildcard rules* that will govern many flows at once.

OpenFlow is examined in detail in Chapter 5. For now, though, the reader should understand that OpenFlow has been adopted by both the research community and by a number of networking vendors. This has resulted in a significant number of network devices supporting OpenFlow on which researchers can experiment and test new ideas.

3.3.2 Open Networking Foundation

OpenFlow began with the publication of the original proposal in 2008. By 2011 OpenFlow had gathered enough momentum that the responsibility for the standard itself moved to the *Open Networking Foundation* (ONF). The ONF was established in 2011 by Deutsche Telekom, Facebook, Google, Microsoft, Verizon, and Yahoo! It is now the guardian of the OpenFlow standard and consists of a number of working groups, many of which are listed in Table 3.2.

Table 3.2 Open Networking Foundation Working Groups

Workgroup	Description
Extensibility	Ongoing creation of new OpenFlow versions to support more features and implementations.
Architecture and Framework	Defining the scope of SDN that the ONF will attempt to standardize.
Forwarding Abstractions	Looking at how the standard interacts with actual hardware implementation components.
Testing and Interoperability	Defining test and interoperability criteria, certification, etc.
Configuration and Management	Defining a protocol for configuring non-flow-related OpenFlow parameters (e.g., setup, controller, etc.)
Migration	Study migration of existing networks towards an SDN network based on OpenFlow.
Optical Transport	Responsible for identifying areas in which the OpenFlow Standard can be applied to optical networks.
Market Education	Responsible for educating the SDN community on the value of OpenFlow and for channeling market feedback within the ONF.
Wireless & Mobile	Study methods by which OpenFlow can be used to control wireless radio area networks (RANs) and core networks.
Northbound Interface	Developing concrete requirements, architecture, and working code for northbound interfaces.

One novel aspect of the ONF is that its board of directors is entirely made up of major network *operators*, not the networking *vendors* themselves. As of this writing, the ONF board is composed of chief technology officers (CTOs), technical directors, and fellows from companies such as Google, Facebook, Deutsche Telekom, Verizon, Microsoft, and NTT, among others. This helps prevent the ONF from supporting the interests of one major networking vendor over another. It also helps provide a *real-world* perspective on what should be investigated and standardized. Conversely, it runs the risk of defining a specification that is difficult for NEMs to implement. Our previous comments about NEMs being locked into their status quo notwithstanding, it is also true that there is a wealth of engineering experience resident in the NEMs regarding how to actually design and build high-performance, high-reliability switches. Though the real-world experience of the users is indeed indispensable, it is imperative that the ONF seek input from the vendors to ensure that the specifications they produce are in fact implementable.

3.4 Sustaining SDN Interoperability

At the current point in the evolution of SDN, we have a standard that has been accepted and adopted by academia and industry alike, and we have an independent standards body to shepherd OpenFlow forward and to ensure that it remains independent and untethered to any specific institution or organization. It is now important to ensure that the implementations of the various players in the OpenFlow space adhere to the standards as they are defined, clarify and expand the standards where they are found to be incomplete or imprecise, and in general guarantee interoperability among OpenFlow implementations. This goal can be achieved in a few ways:

- **Plugfests.** Plugfests, normally staged at conferences, summits, and congresses, are environments in which vendors can bring their devices and software in order to test them with devices and software from other vendors. These are rich opportunities to determine where implementations may be lacking or where the standard itself is unclear and needs to be made more precise and specific.
- **Interoperability labs.** Certain institutions have built dedicated test labs for the purpose of testing the interoperability of equipment from various vendors and organizations. One such lab is the *Indiana Center for Network Translational Research and Education* (InCNTRE) at Indiana University, which hosts a large collection of vendor devices and controllers as well as experimental devices and controllers from open source contributors. We discuss open source contributions to SDN in the next section.
- **Certification programs.** There is a need for certification of switches so that buyers can know they are getting a switch that is certified to support a particular version or versions of OpenFlow. The ONF has now implemented such a program [5].
- **Education and consulting.** A complex, game-changing technological shift such as that represented by SDN will not easily permeate a large industry without the existence of an infrastructure to train and advise networking staff about the migration. It is important that a cadre of highly qualified yet vendor-neutral organizations address this need.

Initially, many SDN interoperability tests revealed dissimilarities and issues. Existing implementations of OpenFlow are increasingly interoperable, but challenges remain. For example, as of this writing, it remains difficult for an OpenFlow controller to work consistently across multiple, varied switch implementations of OpenFlow. This is largely due to limitations in existing ASICs that lead to switches only supporting different subsets of OpenFlow consistent with their particular hardware. Other problems emanate from the fact that the OpenFlow 1.0 specification is vague in terms of specifying which features are required versus which are optional. As we discuss in Chapter 5, the OpenFlow standard is not static, and the goalposts of interoperability are moved with each new release.

3.5 Open Source Contributions

One of the basic rationales for SDN is that innovation and advancement have been stifled as a result of the closed and inflexible environment that exists in networking today. Thus, the openness that results from the creation of standards such as OpenFlow should encourage researchers to dissect old networking methods and should usher in a new dawn of network operation, management, and control. This section examines ways in which open source contributes to this process.

3.5.1 The Power of the Collective

Technological advancement sometimes arises from the efforts of corporations and major organizations, quite often due to the fact that they are the only ones in the position to make contributions in their domains. In the world of software, however, it is occasionally possible for small players to develop technology and make it freely available to the general public. Some examples are:

- **Operating systems.** The Linux operating system was developed as open source and is used today to control countless devices that we use every day, from *digital video recorders* (DVRs) to smartphones.
- **Databases.** Many of the websites we visit for news reports or to purchase products store their information and product data in databases that were developed, at least initially, by the open source community. MySQL is an example of such an open source database.
- **Servers.** When we access locations on the Internet, many of those servers are running application server software and using tools that have been developed over time by the open source community. For example, the open source Apache Web Server is used in countless applications worldwide.
- **Security.** Applying the open source model is also often considered in order to deliver more secure environments [20]. Open source can be more secure because of the peer review and white-box evaluation that naturally occur in the open source development paradigm. Proprietary protocols may be less secure because they are not open and evaluated. Many security products providing antivirus protection and maintaining lists of malicious sites and programs are running open source software to accomplish their tasks. OpenSSL is probably the foremost example of a widely used open source encryption toolkit.
- **File sharing.** The BitTorrent protocol is an example of a hugely successful [3] open protocol used for file sharing. BitTorrent works as a *P2P/overlay* network that achieves high-bandwidth file downloading by performing the download of a file piecemeal and in parallel from multiple servers.

These are just a few examples. Imagine our world today without those items. Would private institutions have eventually implemented the software solutions required in order to provide that functionality? Most likely. Would these advancements in technology have occurred at the velocity that we have witnessed in the past 10 years, without current and ongoing contributions from open source? Almost certainly not.

3.5.2 The Danger of the Collective

Of course, with an endeavor being driven by individuals who are governed not only by their own desire to contribute but also by their free will, whims, and other interests, there is bound to be some risk. The areas of risk include quality, security, timeliness, and support. Here we explain how these risks can be mitigated.

Open source software must undergo tests and scrutiny by even larger numbers of individuals than its commercial counterpart. This is due to the fact that an entire world of individuals has access to and can test those contributions. For any given feature being added or any problem being solved, the open source community may offer a number of competing approaches to accomplish the task. Even open source initiatives are subject to release cycles, and there are key individuals involved in deciding what code will make its way into the next release. Just because an open source developer creates a body of code for an open source product does not mean that code will make it into a release. To keep quality high, competing approaches may be assessed by the community, and admission into a release is actually controlled by key individuals associated with the open source effort. These same factors serve to minimize the risk of security threats. Since the source code is open for all to view, it is more difficult to hide malicious threats such as back doors.

Timeliness is certainly an issue since there are no committed schedules, especially since open source contributors are often doing so in their spare time. The prudent approach is not to depend on future deliverables, but to use existing functionality.

There seem to be two fundamental solutions to the issue of support. If you are a business intending to utilize open source in a product you are developing, you need to feel either that you have the resources to support it on your own or that there is such a large community that has been using that code for a long enough time that you simply trust that the bad bugs have already surfaced and that you are using a stable code base.

It is important to remember that open source licensing models differ greatly from one model to the next. Some models severely restrict the way contributions can be made and are used. Section 11.2 discusses in more detail some of the different major open source licenses and the issues with them.

3.5.3 Open Source Contributions to SDN

Based on the previous discussions, it is easy to see the potential value of open source contributions in the drive toward SDN. Huge advances in SDN technology are attributable to open source projects. Multiple open source implementations of SDN switches, controllers, and applications are available. In Chapter 11 we provide details of the open source projects that have been specifically targeted to accelerate innovation in SDN. In that chapter we also discuss other open source efforts that, though not as directly related to SDN, are nonetheless influential on the ever-growing acceptance of the SDN paradigm.

3.6 Legacy Mechanisms Evolve Toward SDN

The urgent need for SDN we described in Chapter 2 could not wait for a completely new network paradigm to be fleshed out through years of research and experimentation. It is not surprising, then, that there were early attempts to achieve some SDN-like functionality within the traditional networking model. One such example was Cisco's *policy-based routing* that we described in Chapter 1. The capabilities of legacy switches were sometimes extended to support detailed policy configuration related to security, QoS, and other areas. Old APIs were extended to allow centralized programming of these features. Some SDN providers have based their entire SDN solution on a rich family of extended APIs on legacy switches, orchestrated by a centralized controller. In Chapter 6 we examine how these alternative solutions work and whether or not they genuinely constitute an SDN solution.

3.7 Network Virtualization

In Chapter 2 we discussed how *network virtualization* lagged behind its compute and storage counterparts and how this has resulted in a strong demand for network virtualization in the data center. Network virtualization, in essence, provides a network service that is decoupled from the physical hardware below that offers a feature set identical to the behavior of its physical counterpart. An important and early approach to such network virtualization was the *virtual local area network* (VLAN). VLANs permitted multiple virtual local area networks to co-reside on the same layer two physical network in total isolation from one another. Although this technical concept is very sound, the provisioning of VLANs is not particularly dynamic, and they scale only to the extent of a layer two topology. Layer three counterparts based on tunneling scale better than VLANs to larger topologies. Complex systems have evolved to use both VLAN as well as tunneling technologies to provide network virtualization solutions.

One of the most successful commercial endeavors in this space was Nicira, now part of VMware. Early on, Nicira claimed that there were seven properties of network virtualization [4]:

1. Independence from network hardware
2. Faithful reproduction of the physical network service model
3. Following an operational model of compute virtualization
4. Compatibility with any hypervisor platform
5. Secure isolation among virtual networks, the physical networks, and the control plane
6. Cloud performance and scale
7. Programmatic network provisioning and control

Several of these characteristics closely resemble what we have said is required from an SDN solution. SDN promises to provide a mechanism for automating the network and abstracting the physical hardware below from the Software Defined Network above. Network virtualization for data centers has undoubtedly been the largest commercial driver behind SDN. This momentum has become so strong that to some, network virtualization has become synonymous with SDN. Indeed, VMware's (Nicira) standpoint [15] on this issue is that SDN is simply about abstracting control plane from data plane, and therefore network virtualization is SDN. Well, is it SDN or not?

3.8 May I Please Call My Network SDN?

If one were to ask four different attendees at a 2013 networking conference what they thought qualified a network to be called SDN, they would likely have provided divergent answers. Based on the genesis of SDN as presented in Section 3.3, in this book we define an SDN network as characterized by five fundamental traits: *plane separation*, *a simplified device*, *centralized control*, *network automation and virtualization*, and *openness*. We call an SDN solution possessing these five traits an *Open SDN* technology. We acknowledge that there are many competing technologies offered today that claim that their solution is an SDN solution. Some of these technologies have had larger economic impact in terms of the real-life deployments and dollars spent by customers than those that meet all five of our criteria. In some respects they may address customers' needs better than Open SDN. For example, a network virtualization vendor such as Nicira has had huge economic success and widespread installations in data centers but does this without simplified devices. We define these five essential criteria not to pass judgment on these other SDN solutions but in acknowledgment of what the SDN pioneers had in mind when they coined the term SDN in 2009 to refer to their work on OpenFlow. We provide details about each of these five fundament traits in Section 4.1 and compare and contrast competing SDN technologies against these five as well as other criteria in Chapter 6.

3.9 Conclusion

With the research and open source communities clamoring for an open environment for expanded research and experimentation, as well as the urgent needs of data centers for increased agility and virtualization, networking vendors have been forced into the SDN world. Some have moved readily into that world; others have dragged their feet or have attempted to redefine SDN. In the next chapter we examine what SDN is and how it actually works.

References

[1] Casado M, Freedman M, Pettit J, McKeown N, Shenker S. Ethane: taking control of the enterprise. ACM SIGCOMM Comput Commun Rev 2007;37(4):1–12.

[2] Greenberg A, Hjalmtysson G, Maltz D, Myers A, Rexford J, Xie G, et al. A clean-slate 4D approach to network control and management. ACM SIGCOMM Comput Commun Rev 2005;35(3).

[3] BitTorrent and μ Torrent Software Surpass 150 Million User Milestone. Announce new consumer electronics partnerships. BitTorrent; January 2012. Retrieved from <www.bittorrent.com/intl/es/company/about/ces_2012_150m_users>.

[4] Gourley B. The seven properties of network virtualization. CTOvision; August 2012. Retrieved from <ctovision.com/2012/08/the-seven-properties-of-network-virtualization>.

[5] Lightwave Staff. Open networking foundation launches OpenFlow certification program. Lightwave; July 2013. Retrieved from <www.lightwaveonline.com/articles/2013/07/open-networking-foundation-launches-openflow-certification-program.html>.

[6] Campbell A, Katzela I, Miki K, Vicente J. Open signalling for ATM, internet and mobile networks (OPENSIG'98). ACM SIGCOMM Comput Commun Rev 1999;29(1):97–108.

[7] Doria A, Hellstrand F, Sundell K, Worster T. General switch management protocol (GSMP) V3, RFC 3292. Internet Engineering Task Force; 2002.

[8] Devolved Control of ATM Networks. Retrieved from <www.cl.cam.ac.uk/research/srg/netos/old-projects/dcan>.

[9] Doria A, Hadi Salim J, Haas R, Khosravi H, Wang W, Dong L, et al. Forwarding and control element separation (ForCES) protocol specification, RFC 5810. Internet Engineering Task Force; 2010.

[10] Tennehouse D, Smith J, Sincoskie W, Wetherall D, Minden G. A survey of active network research. IEEE Commun Mag 1997;35(1):80–6.
[11] Tennehouse D, Wetherall D. Towards an active network architecture. In: Proceedings of the DARPA active networks conference and exposition. IEEE; 2002. p. 2–15.
[12] Mendonca M, Astuto B, Nunes A, Nguyen X, Obraczka K, Turletti T. A survey of software-defined networking: past, present and future of programmable networks. To appear in IEEE Communications Surveys & Tutorials (2014). Available at http://hal.inria.fr/hal-00825087.
[13] A comparison of IP switching technologies from 3Com, Cascade, and IBM. CSE 588 network systems, Spring 1997. University of Washington. Retrieved from <www.cs.washington.edu/education/courses/csep561/97sp/paper1/paper11.txt>.
[14] Durham D, Boyle J, Cohen R, Herzog S, Rajan R, Sastry A. The COPS (Common Open Policy Service) protocol, RFC 2748. Internet Engineering Task Force; January 2000.
[15] Metz C. What is a virtual network? It's not what you think it is. Wired; May 9, 2012. Retrieved from <www.wired.com/wiredenterprise/2012/05/what-is-a-virtual-network>.
[16] Rigney C, Willens S, Rubens A, Simpson W. Remote authentication dial-in user service (RADIUS), RFC 2865. Internet Engineering Task Force; 2000.
[17] Ferro G. Automation and orchestration. Network computing; September 8, 2011. Retrieved from <www.networkcomputing.com/private-cloud-tech-center/automation-and-orchestration/231600896>.
[18] VMWare vCenter. Retrieved from <www.vmware.com/products/vcenter-server/overview.html>.
[19] VMWare VMotion. Retrieved from <www.vmware.com/files/pdf/VMware-VMotion-DS-EN.pdf>.
[20] Benefits of Open Source Software. Open source for America. Retrieved from <opensourceforamerica.org/learn-more/benefits-of-open-source-software>.
[21] Ferro G. Networking vendors should step up with an SDN strategy. Network computing; June 7, 2012. Retrieved from <www.networkcomputing.com/next-gen-network-tech-center/networking-vendors-should-step-up-with-a/240001600>.
[22] Godbole A. Data communications and networks. Tata McGraw-Hill; 2002.

This page is intentionally left blank

CHAPTER

How SDN Works

4

In previous chapters we have seen why SDN is necessary and what preceded the actual advent of SDN in the research and industrial communities. In this chapter we provide an overview of how SDN actually works, including discussion of the basic components of a Software Defined Networking system, their roles, and how they interact with one another. In the first part of this chapter we focus on the methods used by Open SDN. We also examine how some *alternate* forms of SDN work. As SDN has gained momentum, some networking vendors have responded with alternate definitions of SDN, which better align with their own product offerings. Some of these methods of implementing SDN-like solutions are new (but some are not) and are innovative in their approach. We group the most important of these alternate SDN implementations in two categories: *SDN via existing APIs* and *SDN via hypervisor-based overlay networks*, which we discuss separately in the latter half of this chapter.

4.1 Fundamental Characteristics of SDN

As introduced in Chapter 3, Software Defined Networking, as it evolved from prior proposals, standards, and implementations such as ForCES, 4D, and Ethane, is characterized by five fundamental traits: *plane separation*, *a simplified device*, *centralized control*, *network automation and virtualization*, and *openness*.

4.1.1 Plane Separation

The first fundamental characteristic of SDN is the separation of the forwarding and control planes. Forwarding functionality, including the logic and tables for choosing how to deal with incoming packets based on characteristics such as MAC address, IP address, and VLAN ID, resides in the forwarding plane. The fundamental actions performed by the forwarding plane can be described by the way it dispenses with arriving packets. It may *forward*, *drop*, *consume*, or *replicate* an incoming packet. For basic forwarding, the device determines the correct output port by performing a lookup in the address table in the hardware ASIC. A packet may be dropped due to buffer overflow conditions or due to specific *filtering* resulting from a QoS rate-limiting function, for example. Special-case packets that require processing by the control or management planes are consumed and passed to the appropriate plane. Finally, a special case of forwarding pertains to multicast, where the incoming packet must be replicated before forwarding the various copies out different output ports.

The protocols, logic, and algorithms that are used to program the forwarding plane reside in the control plane. Many of these protocols and algorithms require global knowledge of the network. The control plane determines how the forwarding tables and logic in the data plane should be programmed or configured. Since in a traditional network each device has its own control plane, the primary task of that control plane is to run routing or switching protocols so that all the distributed forwarding tables on the devices throughout the network stay synchronized. The most basic outcome of this synchronization is the prevention of loops.

Although these planes have traditionally been considered logically separate, they co-reside in legacy Internet switches. In SDN, the control plane is moved off the switching device and onto a centralized controller. This is the inspiration behind Figure 1.6 in Chapter 1.

4.1.2 A Simple Device and Centralized Control

Building on the idea of separation of forwarding and control planes, the next characteristic is the simplification of devices, which are then controlled by a centralized system running management and control software. Instead of hundreds of thousands of lines of complicated control plane software running on the device and allowing the device to behave autonomously, that software is removed from the device and placed in a centralized controller. This software-based controller manages the network using higher-level policies. The controller then provides primitive instructions to the simplified devices when appropriate in order to allow them to make fast decisions about how to deal with incoming packets.

4.1.3 Network Automation and Virtualization

Three basic abstractions forming the basis for SDN are defined in [15]. This asserts that SDN can be derived precisely from the abstractions of *distributed state*, *forwarding*, and *configuration*. They are derived from decomposing into simplifying abstractions the actual complex problem of network control faced by networks today. For a historical analogy, note that today's high-level programming languages represent an evolution from their machine language roots through the intermediate stage of languages such as C, where today's languages allow great productivity gains by allowing the programmer to simply specify complex actions through programming abstractions. In a similar manner, [15] purports that SDN is a similar natural evolution for the problem of network control. The distributed state abstraction provides the network programmer with a global network view that shields the programmer from the realities of a network that is actually comprised of many machines, each with its own state, collaborating to solve network-wide problems. The forwarding abstraction allows the programmer to specify the necessary forwarding behaviors without any knowledge of vendor-specific hardware. This implies that whatever language or languages emerge from the abstraction need to represent a sort of lowest common denominator of forwarding capabilities of network hardware. Finally, the configuration abstraction, which is sometimes called the *specification* abstraction, must be able to express the desired goals of the overall network without getting lost in the details of how the physical network will implement those goals. To return to the programming analogy, consider how unproductive software developers would be if they needed to be aware of what is actually involved in writing a block of data to a hard disk when they are instead happily productive with the abstraction of file input and output. Working with the network through this configuration abstraction is really network virtualization at its most basic level. This kind of virtualization lies at the heart of how we define Open SDN in this work.

The centralized software-based controller in SDN provides an open interface on the controller to allow for automated control of the network. In the context of Open SDN, the terms *northbound* and *southbound* are often used to distinguish whether the interface is to the applications or to the devices. These terms derive from the fact that in most diagrams the applications are depicted above (i.e., to the north of) the controller, whereas devices are depicted below (i.e., to the south of) the controller. The southbound API is the OpenFlow interface that the controller uses to program the network devices. The controller offers a northbound API, allowing software applications to be plugged into the controller and thereby allowing that software to provide the algorithms and protocols that can run the network efficiently. These applications can quickly and dynamically make network changes as the need arises. The northbound API of the controller is intended to provide an abstraction of the network devices and topology. That is, the northbound API provides a generalized interface that allows the software above it to operate without knowledge of the individual characteristics and idiosyncrasies of the network devices themselves. In this way, applications can be developed that work over a wide array of manufacturers' equipment that may differ substantially in their implementation details.

One of the results of this level of abstraction is that it provides the ability to virtualize the network, decoupling the network service from the underlying physical network. Those services are still presented to host devices in such a way that those hosts are unaware that the network resources they are using are virtual and not the physical ones for which they were originally designed.

4.1.4 Openness

A characteristic of Open SDN is that its interfaces should remain standard, well documented, and not proprietary. The APIs that are defined should give software sufficient control to experiment with and control various control plane options. The premise is that keeping open both the northbound and southbound interfaces to the SDN controller will allow for research into new and innovative methods of network operation. Research institutions as well as entrepreneurs can take advantage of this capability in order to easily experiment with and test new ideas. Hence the speed at which network technology is developed and deployed is greatly increased as much larger numbers of individuals and organizations are able to apply themselves to today's network problems, resulting in better and faster technological advancement in the structure and functioning of networks. The presence of these open interfaces also encourages SDN-related open source projects. As discussed in Sections 1.7 and 3.5, harnessing the power of the open source development community should greatly accelerate innovation in SDN [6].

In addition to facilitating research and experimentation, open interfaces permit equipment from different vendors to interoperate. This normally produces a competitive environment which lowers costs to consumers of network equipment. This reduction in network equipment costs has been part of the SDN agenda since its inception.

4.2 SDN Operation

At a conceptual level, the behavior and operation of a Software Defined Network is straightforward. In Figure 4.1 we provide a graphical depiction of the operation of the basic components of SDN: the SDN devices, the controller, and the applications. The easiest way to understand the operation is to look at it from the bottom up, starting with the SDN device. As shown in Figure 4.1, the SDN devices contain forwarding functionality for deciding what to do with each incoming packet. The devices also contain the data that drives those forwarding decisions. The data itself is actually represented by the flows defined by the controller, as depicted in the upper-left portion of each device.

A flow describes a set of packets transferred from one network endpoint (or set of endpoints) to another endpoint (or set of endpoints). The endpoints may be defined as IP address-TCP/UDP port

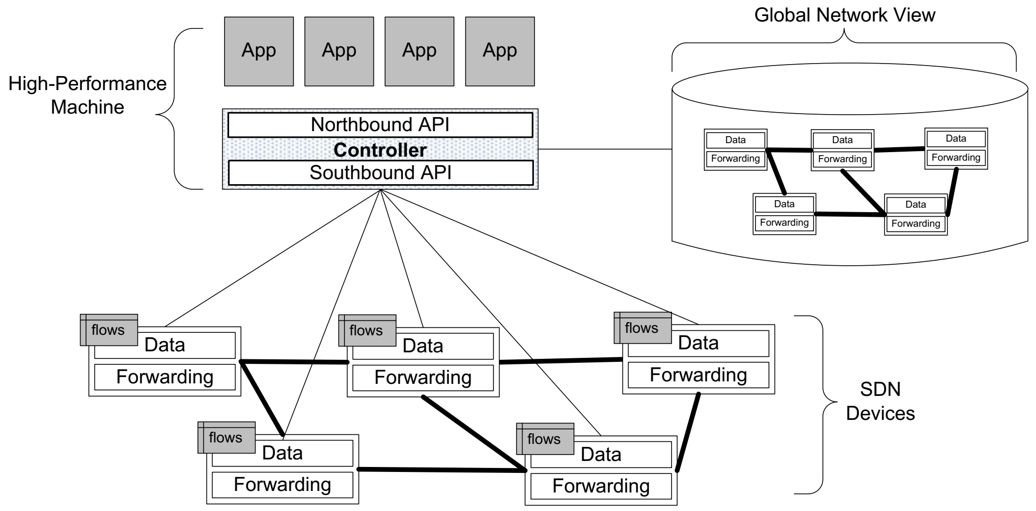


FIGURE 4.1

SDN operation overview.

pairs, VLAN endpoints, layer three tunnel endpoints, and input ports, among other things. One set of rules describes the forwarding actions that the device should take for all packets belonging to that flow. A flow is unidirectional in that packets flowing between the same two endpoints in the opposite direction could each constitute a separate flow. Flows are represented on a device as a flow entry.

A flow table resides on the network device and consists of a series of flow entries and the actions to perform when a packet matching that flow arrives at the device. When the SDN device receives a packet, it consults its flow tables in search of a match. These flow tables had been constructed previously when the controller downloaded appropriate flow rules to the device. If the SDN device finds a match, it takes the appropriate configured action, which usually entails forwarding the packet. If it does not find a match, the switch can either drop the packet or pass it to the controller, depending on the version of OpenFlow and the configuration of the switch. We describe flow tables and this packet-matching process in greater detail in Sections 4.3 and 5.3.

The definition of a flow is a relatively simple programming expression of what may be a very complex control plane calculation previously performed by the controller. For the reader who is less familiar with traditional switching hardware architecture, it is important to understand that this complexity is such that it simply cannot be performed at line rates and instead must be digested by the control plane and reduced to simple rules that can be processed at that speed. In Open SDN, this digested form is the flow entry.

The SDN controller is responsible for abstracting the network of SDN devices it controls and presenting an abstraction of these network resources to the SDN applications running above. The controller allows the SDN application to define flows on devices and to help the application respond to packets that are forwarded to the controller by the SDN devices. In Figure 4.1 we see on the right side of the controller that it maintains a view of the entire network that it controls. This permits it to calculate optimal forwarding solutions for the network in a deterministic, predictable manner. Since one controller can control a large number of network devices, these calculations are normally performed on a high-performance machine with an order-of-magnitude performance advantage over the CPU and memory capacity than is typically afforded to the network devices themselves. For example, a controller might be implemented on an eight-core, 2-GHz CPU versus the single-core, 1-GHz CPU that is more typical on a switch.

SDN applications are built on top of the controller. These applications should not be confused with the application layer defined in the seven-layer OSI model of computer networking. Since SDN applications are really part of network layers two and three, this concept is orthogonal to that of applications in the tight hierarchy of OSI protocol layers. The SDN application interfaces with the controller, using it to set *proactive* flows on the devices and to receive packets that have been forwarded to the controller. Proactive flows are established by the application; typically the application will set these flows when the application starts up, and the flows will persist until some configuration change is made. This kind of proactive flow is known as a *static* flow. Another kind of proactive flow is where the controller decides to modify a flow based on the traffic load currently being driven through a network device.

In addition to flows defined proactively by the application, some flows are defined in response to a packet forwarded to the controller. Upon receipt of incoming packets that have been forwarded to the controller, the SDN application will instruct the controller as to how to respond to the packet and, if appropriate, will establish new flows on the device in order to allow that device to respond locally the next time it sees a packet belonging to that flow. Such flows are called *reactive* flows. In this way, it is now possible to write software applications that implement forwarding, routing, overlay, multipath, and access control functions, among others.

There are also reactive flows that are defined or modified as a result of stimuli from sources other than packets from the controller. For example, the controller can insert flows reactively in response to other data sources such as *intrusion detection systems* (IDS) or the NetFlow traffic analyzer [16].

Figure 4.2 depicts the OpenFlow protocol as the means of communication between the controller and the device. Though OpenFlow is the defined standard for such communication in Open SDN, there

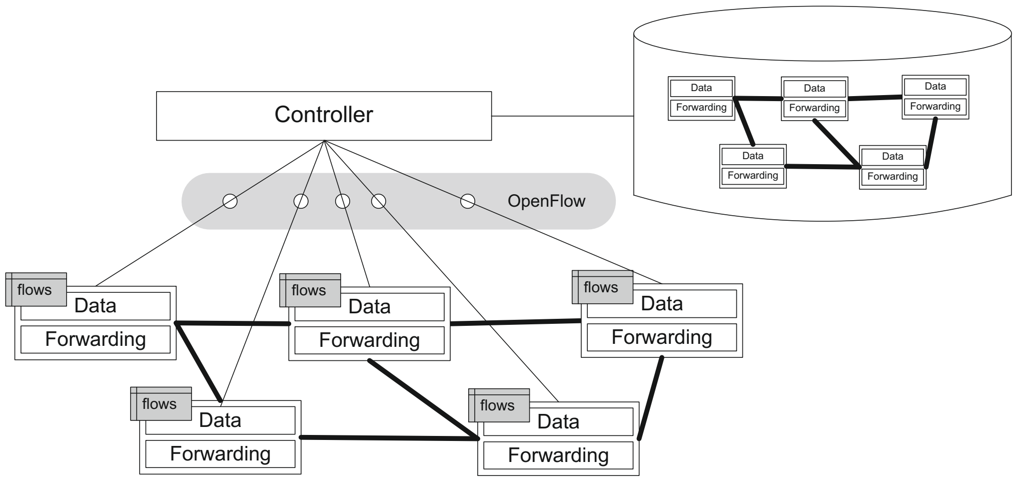


FIGURE 4.2

Controller-to-device communication.

are alternative SDN solutions, discussed later in this chapter, that may use vendor-specific proprietary protocols. The next sections discuss SDN devices, controllers, and applications in greater detail.

4.3 SDN Devices

An SDN device is composed of an API for communication with the controller, an abstraction layer, and a packet-processing function. In the case of a virtual switch, this packet-processing function is packet-processing software, as shown in Figure 4.3. In the case of a physical switch, the packet-processing function is embodied in the hardware for packet-processing logic, as shown in Figure 4.4.

The abstraction layer embodies one or more flow tables, which we discuss in Section 4.3.1. The packet-processing logic consists of the mechanisms to take actions based on the results of evaluating incoming packets and finding the highest-priority match. When a match is found, the incoming packet is processed locally unless it is explicitly forwarded to the controller. When no match is found, the packet may be copied to the controller for further processing. This process is also referred to as the controller *consuming* the packet. In the case of a hardware switch, these mechanisms are implemented by the specialized hardware we discuss in Section 4.3.3. In the case of a software switch, these same functions are mirrored by software. Since the case of the software switch is somewhat simpler than the hardware switch, we will present that first in Section 4.3.2. Some readers may be confused by the distinction between a hardware switch and a software switch. The earliest routers that we described in Chapter 1 were indeed just software switches. Later, as we depicted in Figure 2.1, we explained that over time the actual packet-forwarding logic migrated into hardware for switches that needed to process packets arriving at ever-increasing line rates. More recently, a role has reemerged in the data center for the pure software switch. Such a switch is implemented as a software application usually running in conjunction with a hypervisor in a data center rack. Like a VM, the virtual switch can be instantiated or moved under software control. It normally serves as a virtual switch and works collectively with a set of other such virtual switches to constitute a virtual network. We discuss this concept in greater depth in Chapter 7.

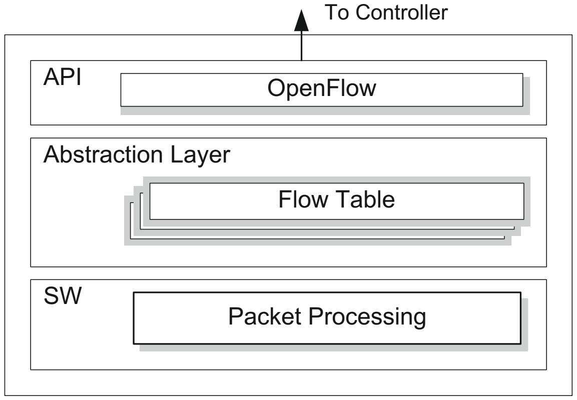


FIGURE 4.3

SDN software switch anatomy.

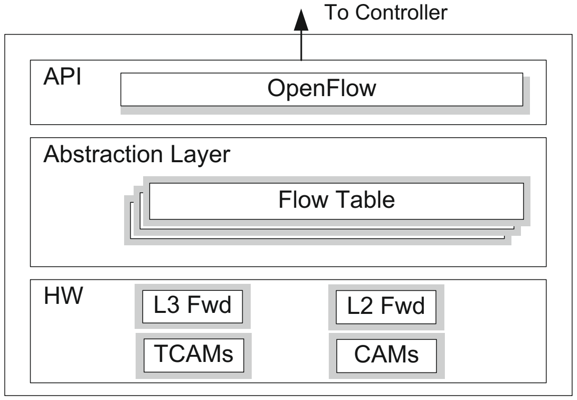


FIGURE 4.4

SDN hardware switch anatomy.

4.3.1 Flow Tables

Flow tables are the fundamental data structures in an SDN device. These flow tables allow the device to evaluate incoming packets and take the appropriate action based on the contents of the packet that has just been received. Packets have traditionally been received by networking devices and evaluated based on certain fields. Depending on that evaluation, actions are taken. These actions may include forwarding the packet to a specific port, dropping the packet, or flooding the packet on all ports, among others. An SDN device is not fundamentally different except that this basic operation has been rendered more generic and more programmable via the flow tables and their associated logic.

Flow tables consist of a number of prioritized flow entries, each of which typically consists of two components: *match fields* and *actions*. Match fields are used to compare against incoming packets. An incoming packet is compared against the match fields in priority order, and the first complete match is selected. Actions are the instructions that the network device should perform if an incoming packet matches the match fields specified for that flow entry.

Match fields can have wildcards for fields that are not relevant to a particular match. For example, when matching packets based just on IP address or subnet, all other fields would be wildcarded. Similarly, if matching on only MAC address or UDP/TCP port, the other fields are irrelevant, and consequently those fields are wildcarded. Depending on the application needs, all fields may be important, in which case there would be no wildcards. The flow table and flow entry constructs allow the SDN application developer to have a wide range of possibilities for matching packets and taking appropriate actions.

Given this general description of an SDN device, we now look at two embodiments of an SDN device: first, the more simple software SDN device and then a hardware SDN device.

4.3.2 SDN Software Switches

In Figure 4.3 we provide a graphical depiction of a purely software-based SDN device. Implementation of SDN devices in software is the simplest means of creating an SDN device, because the flow tables, flow entries, and match fields involved are easily mapped to general software data structures, such as sorted arrays and hash tables. Consequently, it is more probable that two software SDN devices produced by different development teams will behave consistently than will two different hardware implementations. Conversely, implementations in software are likely to be slower and less efficient than those implemented in hardware, since they do not benefit from any hardware acceleration. Consequently, for network devices that must run at high speeds, such as 10 Gbps, 40 Gbps, and 100 Gbps, only hardware implementations are feasible.

Due to the use of wildcards in matching, which poses a problem for typical hash tables, the packet-processing function depicted in Figure 4.3 uses sophisticated software logic to implement efficient match field lookups. Hence, in the early days of SDN, there was a wide variance in the performance of different software implementations, based on the efficiency with which these lookups are accomplished. Fortunately, software SDN device implementations have matured. The fact that there are two widely recognized software reference implementations (see Section 4.3.4), both of which use sophisticated and efficient methods of performing these lookups, has resulted in greater uniformity in software SDN device performance.

Software device implementations also suffer less from resource constraints, since considerations such as processing power and memory size are not an issue in typical implementations. Thus, whereas a hardware SDN device implementation will support only a comparatively limited number of flow entries, the ceiling on the number of flow entries on a software device may be orders of magnitude larger. As software device implementations have more flexibility to implement more complex actions, we expect to see a richer set of actions available on software SDN device implementations than on the hardware SDN devices that we examine in the next section.

Software SDN device implementations are most often found in software-based network devices, such as the hypervisors of a virtualization system. These hypervisors often incorporate a software switch implementation that connects the various virtual machines to the virtual network. The virtual switch working with a hypervisor is a natural fit for SDN. In fact, the whole virtualization system is often controlled by a centralized management system, which also meshes well with the centralized controller aspect of the SDN paradigm.

4.3.3 Hardware SDN Devices

Hardware implementations of SDN devices hold the promise of operating much faster than their software counterparts and, thus, are more applicable to performance-sensitive environments, such as in data centers and network cores. To understand how SDN objects such as flow tables and flow entries can be translated into hardware, her we briefly review some of the hardware components of today's networking devices.

Currently, network devices utilize specialized hardware designed to facilitate the inspection of incoming packets and the subsequent decisions that follow based on the packet-matching operation. We see in Figure 4.4 that the packet-processing logic shown in Figure 4.3 has been replaced by this specialized hardware. This hardware includes the layer two and layer three forwarding tables, usually implemented using *content-addressable memories* (CAMs) and *ternary content-addressable memories* (TCAMs). The layer three forwarding table is used for making IP-level routing decisions. This is the fundamental operation of a router. It matches the destination IP address against entries in the table and, based on the match, takes the appropriate routing action (e.g., forwards the packet out interface B3). The layer two forwarding table is used for making MAC-level forwarding decisions. This is the fundamental operation of a switch. It matches the destination MAC address against entries in the table and, based on the match, takes the appropriate forwarding action (e.g., forwards out interface 15).

The layer two forwarding table is typically implemented using regular CAM or hardware-based hashing. These kinds of associative memories are used when there are precise indices, such as a 48-bit MAC address. TCAMs, however, are associated with more complex matching functions. TCAMs are used in hardware to check not only for an exact match but also for a third state, which uses a mask to treat certain parts of the match field as wildcards. A straightforward example of this process is matching an IP destination address against networks where a *longest prefix match* is performed. Depending on subnet masks, multiple table entries may match the search key, and the goal is to determine the closest match. A more important and innovative use of TCAMs is for potentially matching some but not all header fields of an incoming packet. These TCAMs are thus essential for functions such as *policy-based routing* (PBR).

This hardware functionality allows the device to both match packets and then take actions at a very high rate. However, it also presents a series of challenges to the SDN device developer. Specifically:

- How best to translate from flow entries to hardware entries; for example, how best to utilize the CAMs, TCAMs, or hardware-based hash tables?
- Which of the flow entries to handle in hardware versus how many to fall back to using software? Most implementations are able to use hardware to handle some of the lookups, but others are handed off to software to be handled there. Obviously, hardware will handle the flow lookups much faster than software, but hardware tables have limitations on the number of flow entries they can hold at any time, and software tables could be used to handle the overflow.
- How to deal with hardware action limitations that may impact whether to implement the flow in hardware versus software? For example, certain actions such as packet modification may be limited or not available if handled in hardware.
- How to track statistics on individual flows? In using devices such as TCAMs, which may match multiple flows, it is not possible to use those devices to count individual flows separately. Furthermore, gathering statistics across the various tables can be problematic because the tables may count something twice or not at all.

These and other factors will impact the quality, functionality, and efficiency of the SDN device being developed and must be considered during the design process. For example, hardware table sizes may limit the number of flows, and hardware table capabilities may limit the breadth and depth of special features supported. The limitations presented by a hardware SDN device might require adaptations to SDN applications in order to interoperate with multiple heterogeneous types of SDN devices.

Although the challenges to the SDN device designer are no doubt formidable, the range of variables confronting the SDN application developer is vast. The first-generation SDN device developer is corralled into basically retrofitting existing hardware to SDN and thus does not have many choices—and indeed may be unable to implement all specified features. The SDN application developer, on the other hand, must deal with inconsistencies across vendor implementations, with scaling performance on a network-wide basis, and a host of other more nebulous issues. We discuss some of these application-specific issues in Section 4.5.

This section provided an overview of the composition of SDN devices and the considerations that must be taken into account during their development and their use as part of an SDN application. We provide further specifics on flow tables, flow entries, and actions in Chapter 5.

4.3.4 Existing SDN Device Implementations

A number of SDN device implementations are available today, both commercial and open source. Software SDN devices are predominantly open source. Currently, two main alternatives are available: *Open vSwitch* (OVS) [4] from Nicira and *Indigo* [5] from Big Switch. Incumbent *network equipment manufacturers* (NEMs), such as Cisco, HP, NEC, IBM, Juniper, and Extreme, have added OpenFlow support to some of their legacy switches. Generally, these switches may operate in both legacy mode as well as OpenFlow mode. There is also a new class of devices called *white-box switches*, which are minimalist in that they are built primarily from merchant silicon switching chips and a commodity CPU and memory by a low-cost *original device manufacturer* (ODM) lacking a well-known brand name. One of the premises of SDN is that the physical switching infrastructure may be built from OpenFlow-enabled white-box switches at far less direct cost than switches from established NEMs. Most legacy control plane software is absent from these devices, since this functionality is largely expected to be provided by a centralized controller. Such white-box devices often use the open source OVS or Indigo switch code for the OpenFlow logic, then map the packet-processing part of those switch implementations to their particular hardware.

4.3.5 Scaling the Number of Flows

The granularity of flow definitions will generally be more fine as the device holding them approaches the edge of the network and will be more general as the device approaches the core. At the edge, flows will permit different policies to be applied to individual users and even different traffic types of the same user. This will imply, in some cases, multiple flow entries for a single user. This level of flow granularity would simply not scale if it were applied closer to the network core, where large switches deal with the traffic for tens of thousands of users simultaneously. In those core devices, the flow definitions will be generally more coarse, with a single aggregated flow entry matching the traffic from a large number of users whose traffic is aggregated in some way, such as a tunnel, a VLAN, or a MPLS LSP. Policies applied deep into the network will likely not be user-centric policies but rather policies that apply to these aggregated flows. One positive result is that there will not be an explosion in the number of flow entries in the core switches due to handling the traffic emanating from thousands of flows in edge switches.

4.4 SDN Controller

We have noted that the controller maintains a view of the entire network, implements policy decisions, controls all the SDN devices that comprise the network infrastructure, and provides a northbound API for applications. When we have said that the controller implements policy decisions regarding routing, forwarding, redirecting, load balancing, and the like, these statements referred to both the controller and the applications that make use of that controller. Controllers often come with their own set of common application modules, such as a learning switch, a router, a basic firewall, and a simple load balancer. These are really SDN applications, but they are often bundled with the controller. Here we focus strictly on the controller.

Figure 4.5 exposes the anatomy of an SDN controller. The figure depicts the modules that provide the controller's core functionality, both a northbound and a southbound API, and a few sample applications that might use the controller. As we described earlier, the southbound API is used to interface with the SDN devices. This API is OpenFlow in the case of Open SDN or some proprietary alternative in other SDN solutions. It is worth noting that in some product offerings, both OpenFlow and alternatives coexist on the same controller. Early work on the southbound API has resulted in more maturity of that interface with respect to its definition and standardization. OpenFlow itself is the best example of this maturity, but de facto standards such as the Cisco CLI and SNMP also represent standardization in the southbound-facing interface. OpenFlow's companion protocol, OF-Config [17], and Nicira's *Open vSwitch Database Management Protocol* (OVSDB) [18] are both open protocols for the southbound interface, though these are limited to configuration roles.

Unfortunately, there is currently no northbound counterpart to the southbound OpenFlow standard or even the de facto legacy standards. This lack of a standard for the controller-to-application interface is considered a current deficiency in SDN, and some bodies are developing proposals to standardize it. The absence of a standard notwithstanding, northbound interfaces have been implemented in a number of disparate forms. For example, the Floodlight controller [2] includes a Java API and a *Representational State Transfer* (RESTful) [13] API. The OpenDaylight controller [14] provides a RESTful API for applications running on separate machines. The northbound API represents an outstanding opportunity for innovation and collaboration among vendors and the open source community.

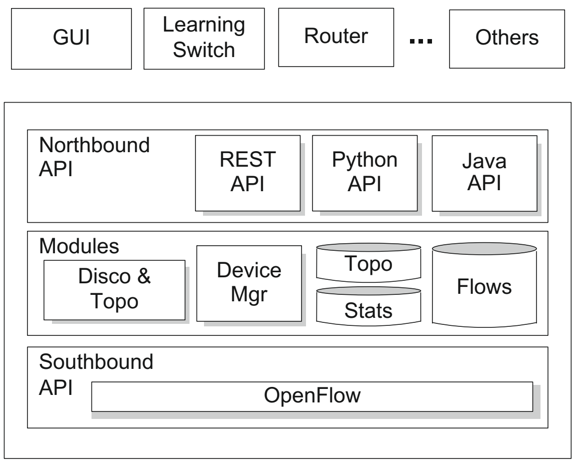


FIGURE 4.5

SDN controller anatomy.

4.4.1 SDN Controller Core Modules

The controller abstracts the details of the SDN controller-to-device protocol so that the applications above are able to communicate with those SDN devices without knowing their nuances. Figure 4.5 shows the API below the controller, which is OpenFlow in Open SDN, and the interface provided for applications. Every controller provides core functionality between these raw interfaces. Core features in the controller include:

- **End-user device discovery.** Discovery of end-user devices such as laptops, desktops, printers, mobile devices, and so on.
- **Network device discovery.** Discovery of network devices that comprise the infrastructure of the network, such as switches, routers, and wireless access points.
- **Network device topology management.** Maintain information about the interconnection details of the network devices to each other and to the end-user devices to which they are directly attached.
- **Flow management.** Maintain a database of the flows being managed by the controller and perform all necessary coordination with the devices to ensure synchronization of the device flow entries with that database.

The core functions of the controller are device and topology discovery and tracking, flow management, device management, and statistics tracking. These are all implemented by a set of modules internal to the controller. As shown in Figure 4.5, these modules need to maintain local databases containing the current topology and statistics. The controller tracks the topology by learning of the existence of switches (SDN devices) and end-user devices and tracking the connectivity between them. It maintains a *flow cache* that mirrors the flow tables on the various switches it controls. The controller locally maintains per-flow statistics that it has gathered from its switches. The controller may be designed such that functions are implemented via pluggable modules such that the feature set of the controller may be tailored to an individual network's requirements.

4.4.2 SDN Controller Interfaces

For SDN applications, a key function provided by the SDN controller is the API for accessing the network. In some cases, this northbound API is a low-level interface, providing access to the network devices in a common and consistent manner. In this case, that application is aware of individual devices but is shielded from their differences. In other instances the controller may provide high-level APIs that give an abstraction of the network itself, so that the application developer need not be concerned with individual devices but rather with the network as a whole.

Figure 4.6 takes a closer look at how the controller interfaces with applications. The controller informs the application of *events* that occur in the network. Events are communicated from the controller to the application. Events may pertain to an individual packet that has been received by the controller or some state change in the network topology, such as a link going down. Applications use different *methods* to affect the operation of the network. Such methods may be invoked in response to a received event and may result in a received packet being dropped, modified, and/or forwarded or the addition, deletion, or modification of a flow. The applications may also invoke methods independently, without the stimulus of an event from the controller, as we explain in Section 4.5.1. Such inputs are represented by the “Other Context” box in Figure 4.6.

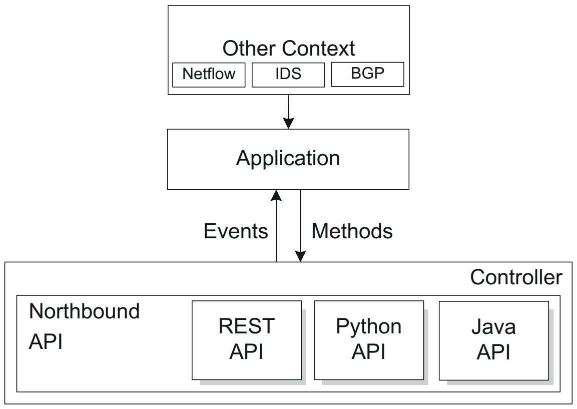


FIGURE 4.6

SDN controller northbound API.

4.4.3 Existing SDN Controller Implementations

There are a number of implementations of SDN controllers available on the market today. They include both open source SDN controllers and commercial SDN controllers. Open source SDN controllers come in many forms, from basic C-language controllers such as NOX [7] to Java-based versions such as Beacon [1] and Floodlight [2]. There is even a Ruby-based [8] controller called Trema [9]. Interfaces to these controllers may be offered in the language in which the controller is written or other alternatives, such as REST or Python. An open source controller called OpenDaylight [3] has been built by a consortium of vendors. Other vendors offer their own commercial versions of an SDN controller. Vendors such as NEC, IBM, and HP offer controllers that are primarily OpenFlow implementations. Most other NEMs offer vendor-specific and proprietary SDN controllers that include some level of OpenFlow support.

There are pros and cons to the proprietary alternative controllers. Although proprietary controllers are more closed than the nominally open systems, they do offer some of the automation and programmability advantages of SDN while providing a *buck stops here* level of support for the network equipment. They permit SDN-like operation of legacy switches, obviating the need to replace older switching equipment in order to begin the migration to SDN. They do constitute closed systems, however, which ostensibly violates one of the original tenets of SDN. They also may do little to offload control functionality from devices, resulting in the continued high cost of network devices. These proprietary alternative controllers are generally a component of the alternative SDN methodologies we introduce in Section 4.6.

4.4.4 Potential Issues with the SDN Controller

In general, the Open SDN controller suffers from the birthing pains common to any new technology. Although many important problems are addressed by the concept and architecture of the controller, there have been comparatively few large-scale commercial deployments thus far. As more commercial deployments scale, more real-life experience in large, demanding networks will be needed. In particular, experience with a wider array of applications with a more heterogeneous mix of equipment types is needed before widespread confidence in this architecture is established. Achieving success in these varied deployments will require that a number of potential controller issues be adequately addressed. In some cases, these solutions will come in multiple forms from different vendors. In other cases, a standards body such as the ONF will have to mandate a standard. In Section 3.2.6 we stated that a centralized control architecture needed to grapple with the issues of latency, scale, high availability, and security. In addition to these more general issues, the centralized SDN controller will need to confront the challenges of *coordination between applications*, *the lack of a standard northbound API*, and *flow prioritization*.

There may be more than one SDN application running on a single controller. When this is the case, issues related to application prioritization and flow handling become important. Which application should receive an event first? Should the application be required to pass along this event to the next application in line, or can it deem the processing complete, in which case no other applications get a chance to examine and act on the received event?

The lack of an emerging standard for the northbound API is stymieing efforts to develop applications that will be reusable across a wide range of controllers. Early standardization efforts for OpenFlow generally assumed that such a northbound counterpart would emerge, and much of the efficiency gain assumed to come from a migration to SDN will be lost without it. Late in 2013 the ONF formed a workgroup that focuses on the standardization of the northbound API (see Table 3.2).

Flows in an SDN device are processed in priority order. The first flow that matches the incoming packet is acted upon. Within a single SDN application, it is critical for the flows on the SDN device to be prioritized correctly. If they are not, the resulting behavior will be incorrect. For example, the designer of an application will put more specific flows at a higher priority (e.g., match all packets from IP address 10.10.10.2 and TCP port 80) and the most general flows at the lowest priority (e.g., match everything else). This is relatively easy to do for a single application. However, when there are multiple SDN applications, flow entry prioritization becomes more difficult to manage. How does the controller appropriately interleave the flows from all applications? This is a challenge and requires special coordination between the applications.

4.5 SDN Applications

SDN applications run above the SDN controller, interfacing to the network via the controller's northbound API. SDN applications are ultimately responsible for managing the flow entries that are programmed on the network devices using the controller's API to manage flows. Through this API the applications are able to (1) configure the flows to route packets through the best path between two endpoints; (2) balance traffic loads across multiple paths or destined to a set of endpoints; (3) react to changes in the network topology such as link failures and the addition of new devices and paths, and (4) redirect traffic for purposes of inspection, authentication, segregation, and similar security-related tasks.

Figure 4.5 includes some standard applications, such as a *graphical user interface* (GUI) for managing the controller, a learning switch, and a routing application. The reader should note that even the basic functionality of a simple layer two learning switch is not obtained by simply pairing an SDN device with an SDN controller. Additional logic is necessary to react to the newly seen MAC address and update the forwarding tables in the SDN devices being controlled in such a way as to provide connectivity to that new MAC address throughout the network while avoiding switching loops. This additional logic is embodied in the learning switch application in Figure 4.5. One of the perceived strengths of the SDN architecture is the fact that switching decisions can be controlled by an ever-richer family of applications that control the controller. In this way, the power of the SDN architecture is highly expandable. Other applications that are well suited to this architecture are load balancers and firewalls, among many others.

These examples represent some typical SDN applications that have been developed by researchers and vendors today. Applications such as these demonstrate the promise of SDN: being able to take complex functionality that formerly resided in each individual network device or appliance and allowing it to operate in an Open SDN environment.

4.5.1 SDN Application Responsibilities

The general responsibility of an SDN application is to perform whatever function for which it was designed, whether load balancing, firewalling, or some other operation. Once the controller has finished initializing devices and has reported the network topology to the application, the application spends most of its processing time responding to events. The core functionality of the application will vary from one application to another, but application behavior is driven by events coming from the controller as well as external inputs. External inputs could include network monitoring systems such as Netflow, IDS, or BGP peers. The application affects the network by responding to the events as modeled in Figure 4.6. The SDN application registers as a *listener* for certain events, and the controller will invoke the application's callback *method* whenever such an event occurs. This invocation will be accompanied by the appropriate details related to the event. Some examples of events handled by an SDN application are *end-user device discovery*, *network device discovery*, and *incoming packet*. In the first two cases, events are sent to the SDN application upon the discovery of a new end-user device (i.e., a MAC address) or a new network device (e.g., a switch, router, or wireless access point), respectively. Incoming packet events are sent to the SDN application when a packet is received from an SDN device due to either a flow entry instructing the SDN device to forward the packet to the controller or because there is no matching flow entry at the SDN device. When there is no matching flow entry, the default action is usually to forward the packet to the controller, though it could be to drop the packet, depending on the nature of the applications.

There are many ways in which an SDN application can respond to events that have been received from the SDN controller. There are simple responses, such as downloading a set of default flow entries to a newly discovered device. These default or *static* flows typically are the same for every class of discovered network device, and hence little processing is required by the application. There are also more complex responses that may require state information gathered from some other source apart from the controller.

This can result in variable responses, depending on that state information. For example, based on a user's state, an SDN application may decide to process the current packet in a certain manner, or it may take some other action, such as downloading a set of user-specific flows.

4.6 Alternate SDN Methods

Thus far in this chapter we have examined what we consider the original definition of SDN, which we distinguish from other alternatives by the term *Open* SDN. Open SDN most certainly has the broadest support in the research community and among the operators of large data centers such as Google, Yahoo!, and their ilk. Nevertheless, there are other proposed methods of accomplishing at least some of the goals of SDN. These methods are generally associated with a single networking vendor or a consortium of vendors. We define here two alternate categories of SDN implementations: *SDN via existing APIs* and *SDN via hypervisor-based overlay networks*. The first of these consists of employing functions that exist on networking devices that can be invoked remotely, typically via traditional methods such as SNMP or CLI or by the newer, more flexible mechanisms provided by RESTful APIs. In SDN via hypervisor-based overlay networks, the details of the underlying network infrastructure are not relevant. Virtualized *overlay* networks are instantiated across the top of the physical network. We make the distinction that the overlays be hypervisor-based since network overlays exist in other, non-hypervisor-based forms as well. One early example of this sort of network virtualization is VLAN technology. Another type of overlay network that is not related to our use of the term is the *P2P/overlay* network, such as Napster and BitTorrent. We further clarify the distinction between SDN and P2P/overlay networks later in Section 8.8.

In the following sections we provide an introduction of how these two alternate SDN methods work. We defer a more detailed treatment of these as well as the introduction of some others to Chapter 6.

4.6.1 SDN via Existing APIs

If a basic concept of SDN is to move control functionality from devices to a centralized controller, this can be accomplished in other ways than the OpenFlow-centric approach coupled to Open SDN. In particular, one method is to provide a richer set of control points on the devices so that centrally located software can manipulate those devices and provide the intelligent and predictable behavior that is expected in an SDN-controlled network. Consequently, many vendors offer SDN solutions by improving the means of affecting configuration changes on their network devices.

We depict *SDN via existing APIs* graphically in Figure 4.7. The diagram shows a controller communicating with devices via a proprietary API. Often with SDN via existing APIs solutions, vendors provide an enhanced level of APIs on their devices, rather than just the traditional CLI and SNMP. The architecture shown in the figure is reminiscent of the earlier diagrams in this chapter. As before, there is a set of applications that use a centralized controller to affect forwarding in the physical network. The genesis of this idea derives from the fact that legacy switches all afford some degree of control over forwarding decisions via their existing management interfaces. By providing a simplifying abstraction to this plethora of interfaces, some network programmability can be gained by this approach.

Since the early days of the commercial Internet, it has been possible to set configuration parameters on devices using methods such as the CLI and SNMP. Although these mechanisms have long been available, they can be cumbersome and difficult to maintain. Furthermore, they are geared toward relatively rare static management tasks, not the dynamic, frequent, and automated tasks required in environments such as today's data centers. Newer methods of providing the means to make remote configuration changes have been developed in the last few years. The most common of these is the RESTful API. REST has become the dominant method of making API calls across networks. REST uses *HyperText Transfer Protocol* (HTTP), the protocol commonly used to pass web traffic. RESTful APIs are simple and extensible and have the advantage of using a standard TCP port and thus require no special firewall configuration to permit the API calls to pass through firewalls. We provide a more detailed discussion of RESTful APIs in Section 6.2.3.

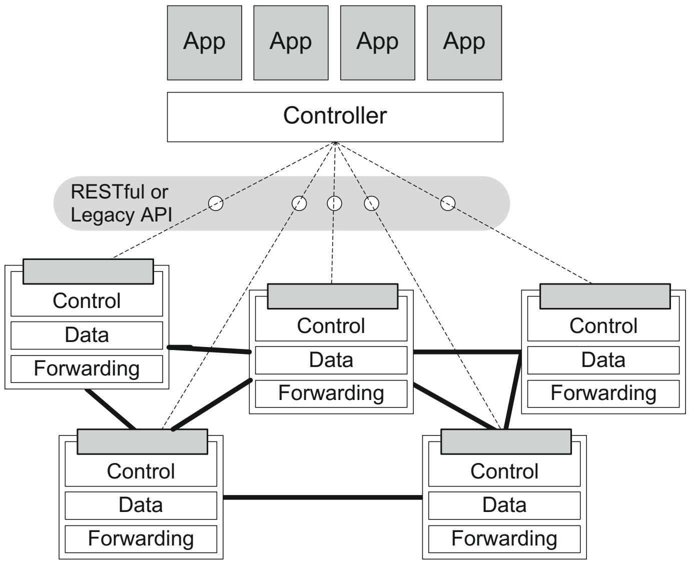


FIGURE 4.7

SDN via existing APIs.

There are a number of benefits of SDN via existing APIs. One distinct advantage of this approach is that, because it uses legacy management interfaces, it therefore works with legacy switches. Thus, this solution does not require upgrading to OpenFlow-enabled switches. Another benefit of this approach is that it allows for some improvement in agility and automation. These APIs also make it easier to write software such as orchestration tools that can respond quickly and automatically to changes in the network (e.g., the movement of a virtual machine in a data center). A third advantage is that these APIs allow for some amount of centralized control of the devices in the network. Therefore, it is possible to build an SDN solution using the provided APIs on the distributed network devices. Finally, there is potential for increased openness in the SDN via existing APIs approach. Although the individual interfaces may be proprietary to individual vendors, when they are exposed to the applications, they are made open for exploitation by applications. The degree of openness will vary from one NEM to another.

Of course, the API-based SDN methods have their limitations. First, in most cases *there is no controller at all*. The network programmer needs to interact directly with each switch. Second, even when there is a controller, it does not provide an abstract, *network-wide* view to the programmer. Instead, the programmer needs to think in terms of individual switches. Third, since there is still a control plane operating on each switch, the controller and, more important, the programmer developing applications on top of that controller must synchronize with what the distributed control plane is doing. Another drawback is that the solution is proprietary. Since these APIs are nonstandard (as opposed to a protocol such as OpenFlow), SDN-like software applications using this type of API-based approach will only work with devices from that specific vendor or a small group of compatible vendors. This limitation is sometimes circumvented by extending this approach to provide support for multiple vendors' APIs.

This masks the differences between the device APIs to the application developer, who will see a single northbound API despite the incompatible device interfaces on the southbound side. Obviously, this homogeneity on the northbound interface is achieved by increased complexity within the controller.

In addition, the SDN precept of moving control off the switch onto a common controller was in part intended to create simpler, less expensive switches. The SDN via existing APIs approach relies on the same complicated, expensive switches as before. Admittedly, this is a double-edged sword, since a company that already has the expensive switches may find it more expensive to change to less expensive switches, considering they already have made the investment in the legacy devices.

Finally, though the SDN via existing APIs approach does allow some control over forwarding, in particular with VLANs and VPNs, it does not allow the same degree of fine-grained control of individual flows afforded by OpenFlow.

In summary, SDN via existing APIs is a step in the right direction, moving toward the goal of centralized, software-based network control. It is possible to view SDN via existing APIs as a practical extension of current functionality that is useful when the more radical OpenFlow solution is not yet available or is otherwise inappropriate.

4.6.2 SDN via Hypervisor-Based Overlay Networks

Another more innovative alternate SDN method is what we refer to as *hypervisor-based overlay* networks. Under this concept the current physical network is left as it is, with networking devices and their configurations remaining unchanged. Above that network, however, *hypervisor-based virtualized* networks are erected. The systems at the edges of the network interact with these virtual networks, which obscure the details of the physical network from the devices that connect to the overlays.

We depict such an arrangement in Figure 4.8, where we see the virtualized networks *overlaying* the physical network infrastructure. The SDN applications making use of these overlay network resources

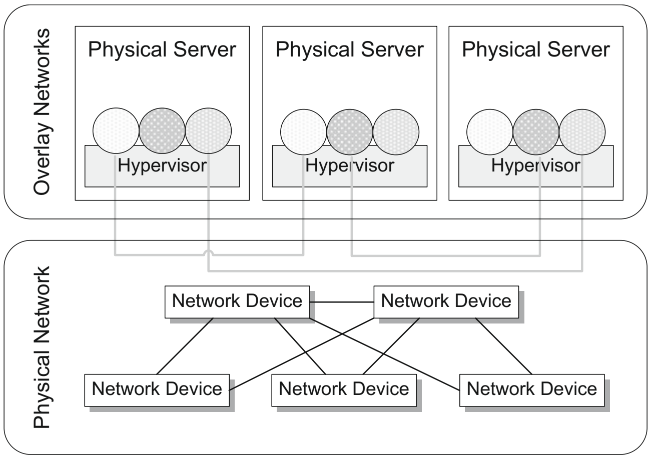


FIGURE 4.8

Virtualized networks.

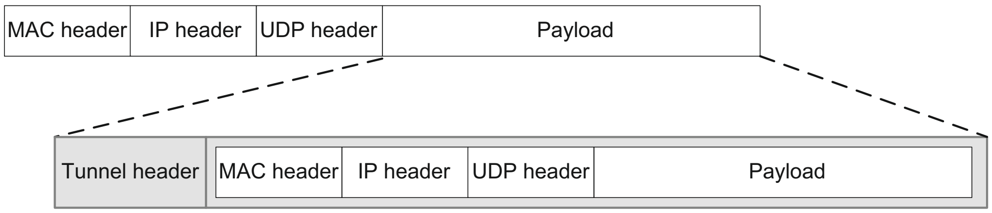


FIGURE 4.9

Encapsulated frames.

are given access to virtualized networks and ports, which are abstract in nature and do not necessarily relate directly to their physical counterparts below.

As shown in Figure 4.8, conceptually the virtual network traffic runs *above* the physical network infrastructure. The hypervisors inject traffic into the virtual network and receive traffic from it. The traffic of the virtual networks is passed through those physical devices, but the endpoints are unaware of the details of the physical topology, the way routing occurs, or other basic network functions. Since these virtual networks exist above the physical infrastructure, they can be controlled entirely by the devices at the very edge of the network. In data centers, these would typically be the hypervisors of the VMs that are running on each server.

The mechanism that makes this possible is tunneling, which uses encapsulation. When a packet enters the edge of the virtual network at the source, the networking device (usually the hypervisor) will take the packet in its entirety and encapsulate it within another frame. This is shown in Figure 4.9. Note that the edge of the virtual network is called a *tunnel endpoint* or *virtual tunnel endpoint* (VTEP).

The hypervisor then takes this encapsulated packet and, based on information programmed by the controller, sends it to the destination's VTEP. This VTEP decapsulates the packet and forwards it to the destination host. As the encapsulated packet is sent across the physical infrastructure, it is being sent from the source's VTEP to the destination's VTEP. Consequently, the IP addresses are those of the source and destination VTEP. Normally, in network virtualization, the VTEPs are associated with hypervisors.

This tunneling mechanism is referred to as *MAC-in-IP* tunneling because the entire frame, from MAC address inward, is encapsulated within this unicast IP frame, as shown in Figure 4.9. Different vendors have established their own proprietary methods for MAC-in-IP tunneling. Specifically, Cisco offers VXLAN [10], Microsoft uses NVGRE [11], and Nicira's is called STT [12].

This approach mandates that a centralized controller be in charge of making sure there is always a mapping from the actual destination host to the destination hypervisor that serves that host.

Figure 4.10 shows the roles of these VTEPs as they serve the source and destination host devices. The virtual network capability is typically added to a hypervisor by extending it with a virtual switch. We introduced the notion of virtual switch in Section 4.3.2, and it is well suited to the overlay network concept. The virtual network has a virtual topology consisting of the virtual switches interconnected by virtual point-to-point links. The virtual switches are depicted as the VTEPs in Figure 4.10, and the virtual links are the tunnels interconnecting them. All the traffic on each virtual network is encapsulated as shown in Figure 4.9 and sent VTEP-to-VTEP. The reader should note that the tunnels depicted in

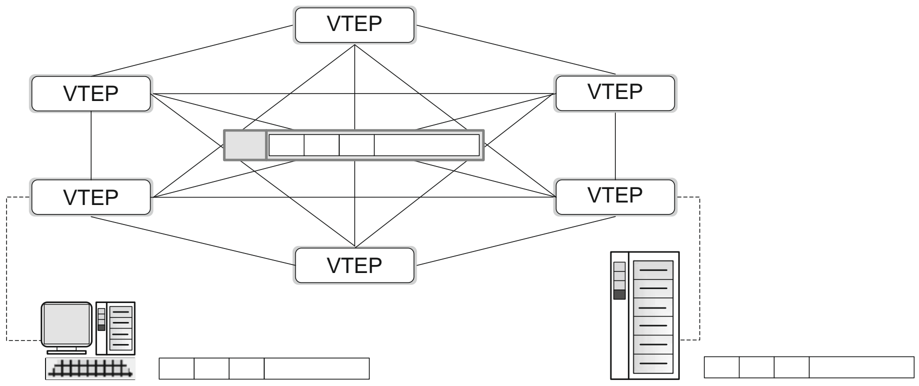


FIGURE 4.10

Virtual tunnel endpoints.

Figure 4.10 are the same as the links interconnecting hypervisors in Figure 4.8. As Figure 4.8 indicates, multiple overlay networks can exist independently and simultaneously over the same physical network.

In summary, SDN via hypervisor-based overlay networks is well suited to environments such as data centers already running compute and storage virtualization software for their servers. It does address a number of the needs of an SDN solution. First, it addresses MAC address explosion in data centers and cloud environments because all those host MAC addresses are hidden within the encapsulated frame. Second, it addresses VLAN limitations because all traffic is tunneled and VLANs are not required for supporting the isolation of multiple tenants. Third, it addresses agility and automation needs because it is implemented in software, and these virtual networks can be constructed and taken down in a fraction of the time that would be required to change the physical network infrastructure.

Nevertheless, these overlay networks do not solve all the problems that can be addressed by an Open SDN solution. In particular, they do not address existing issues within the physical infrastructure, which still requires manual configuration and maintenance. Moreover, they fail to address traffic prioritization and efficiency in the physical infrastructure, so confronting STP's blocked links and QoS settings continues to challenge the network engineer. Finally, hypervisor-based overlays do not address the desire to open up network devices for innovation and simplification, since those physical network devices have not changed at all.

4.7 Conclusion

This chapter has described the basic functionality related to the manner in which an SDN solution actually works. It is important to realize that there is no fundamental incompatibility between the hypervisor-based overlay network approach to SDN and Open SDN. In fact, some implementations use OpenFlow to create and utilize the tunnels required in this kind of network virtualization. It is not unreasonable to think of these overlay networks as stepping stones toward a more complete SDN solution that includes SDN and OpenFlow for addressing the virtual as well as the physical needs of the network. In Chapter 6 we delve more deeply into the SDN alternatives introduced in this chapter as well as other alternatives not yet discussed. First, though, in the next chapter we provide a detailed overview of the OpenFlow specification.

References

[1] Erikson D. Beacon, OpenFlow@Stanford; February 2013. Retrieved from <openflow.stanford.edu/display/Beacon/Home>.

[2] Wang K. Floodlight documentation. Project floodlight; December 2013. Retrieved from <docs.projectfloodlight.org/display/floodlightcontroller>.

[3] Lawson S. Network heavy hitters to pool SDN efforts in OpenDaylight project. Network World; April 8, 2013. Retrieved from <www.networkworld.com/news/2013/040813-network-heavy-hitters-to-pool-268479.html>.

[4] Production quality, multilayer open virtual switch. Open vSwitch; December 15, 2013. Retrieved from <openvswitch.org>.

[5] Open thin switching, open for business. Big switch networks; June 27, 2013. Retrieved from <www.bigswitch.com/topics/introduction-of-indigo-virtual-switch-and-switch-light-beta>.

[6] Industry leaders collaborate on OpenDaylight project, donate key technologies to accelerate software-defined networking. OpenDaylight; April 2013. Retrieved from <www.opendaylight.org/announcements/2013/04/industry-leaders-collaborate-opendaylight-project-donate-key-technologies>.

[7] NOX; December 15, 2013. Retrieved from <www.noxrepo.org>.

[8] Ruby: a programmer's best friend; December 15, 2013. Retrieved from <www.ruby-lang.org>.

[9] Trema: full-stack OpenFlow framework in Ruby and C; December 15, 2013. Retrieved from <trema.github.io/trema>.

[10] Mahalingam M, Dutt D, Duda K, Agarwal P, Kreeger L, Sridhar T, et al. VXLAN: a framework for overlaying virtualized layer 2 networks over layer 3 networks. Internet Engineering Task Force; August 26, 2011 [internet draft].

[11] Sridharan M, et al. NVGRE: network virtualization using generic routing encapsulation. Internet Engineering Task Force; September 2011 [internet draft].

[12] Davie B, Gross J. STT: a stateless transport tunneling protocol for network virtualization (STT). Internet Engineering Task Force; March 2012 [internet draft].

[13] Learn REST: a RESTful tutorial; December 15, 2013. Retrieved from <www.restapitutorial.com>.

[14] Open Daylight technical overview; December 15, 2013. Retrieved from <www.opendaylight.org/project/technical-overview>.

[15] Shenker S. The future of networking, and the past of protocols. Open networking summit. Palo Alto, CA, USA: Stanford University; October 2011.

[16] NetFlow traffic analyzer, solarwinds; December 15, 2013. Retrieved from <www.solarwinds.com/netflow-traffic-analyzer.aspx>.

[17] OpenFlow management and configuration protocol(OF-Config 1.1.1). Open Networking Foundation; March 23, 2013. Retrieved from <www.opennetworking.org/sdn-resources/onf-specifications>.

[18] Pfaff B, Davie B. The open vSwitch database management protocol. Internet Engineering Task Force; October 2013 [internet draft].

This page is intentionally left blank

CHAPTER

The OpenFlow Specification 5

Casual followers of networking news can experience confusion as to whether SDN and OpenFlow are one and the same thing. Indeed, they are not. OpenFlow is definitely a distinct subset of the technologies included under the big tent of SDN. An important tool and catalyst for innovation, OpenFlow defines both the communications protocol between the SDN data plane and the SDN control plane and part of the behavior of the data plane. It does not describe the behavior of the controller itself. There are other approaches to SDN, but today OpenFlow is the only nonproprietary, general-purpose protocol for programming the forwarding plane of SDN switches. This chapter focuses only on the protocol and behaviors dictated by OpenFlow, since there are no directly competing alternatives.

We see the basic components of an OpenFlow system in Figure 5.1. There is always an *OpenFlow controller* that communicates to one or more *OpenFlow switches*. The OpenFlow protocol defines the specific messages and message formats exchanged between controller (control plane) and device (data plane). The OpenFlow behavior specifies how the device should react in various situations and how it should respond to commands from the controller. There have been various versions of OpenFlow, which we examine in detail in this chapter. We also consider some of the potential drawbacks and limitations of the OpenFlow specification.

Note that our goal in this chapter is not to provide an alternative to a detailed reading of the specification itself. We hope to provide the reader with a rudimentary understanding of the elements of OpenFlow and how they interoperate to provide basic switch and routing functions, as well as an experimental platform that allows each version to serve as a springboard for innovations that may appear in future versions of OpenFlow. In particular, though we provide tables that list the different ports, messages, instructions, and actions that make up OpenFlow, we do not attempt to explain each in detail, because this would essentially require that we replicate the specification here. We use the most important of these in clear examples so that the reader is left with a basic understanding of OpenFlow operation. We provide a reference to each of the four versions of OpenFlow in the section where it is introduced.

5.1 Chapter-Specific Terminology

The following new operations are introduced in this chapter:

To *pop* an item is to remove it from a *last-in-first-out* (LIFO) ordered list of like items.

To *push* an item is to add it to a LIFO ordered list of like items.

These terms are frequently used in conjunction with the term *stack*, which is a computer science term for a LIFO ordered list. If one pictures a stack of items such as books, and if three books are then added to that stack, the order in which those books are normally retrieved from the pile is a LIFO order. That is, the first book pulled off the top of the pile is the last one that was added to it. If we alter slightly

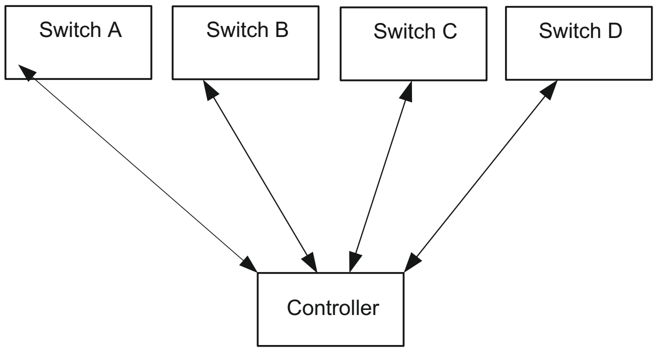


FIGURE 5.1

OpenFlow components.

the perception of the stack of books to imagine that the stack exists in a spring-loaded container such that the topmost book is level with the top of the container, the use of the terms *pop* and *push* become apparent. When we add a book to such a stack we *push* it onto the top and later *pop* it off to retrieve it. The OpenFlow specification includes several definitions that use these concepts. In this context, the *push* involves adding a new header element to the existing packet header, such as an MPLS label. Since multiple such labels may be pushed on before any are popped off, and since they are popped off in LIFO order, the stack analogy is apt.

5.2 OpenFlow Overview

The OpenFlow specification has been evolving for a number of years. The nonprofit Internet organization *openflow.org* was created in 2008 as a mooring to promote and support OpenFlow. Though openflow.org existed formally on the Internet, in the early years the physical organization was really just a group of people that met informally at Stanford University. From its inception OpenFlow was intended to "belong" to the research community to serve as a platform for open network switching experimentation, with an eye on commercial use through commercial implementations of this public specification. The first release, Version 1.0.0, appeared on December 31, 2009, though numerous point prereleases existed before then and were made available for experimental purposes as the specification evolved. At this point and continuing up through release 1.1.0, development and management of the specification were performed under the auspices of openflow.org. On March 21, 2011, the *Open Network Foundation* (ONF) was created for the express purpose of accelerating the delivery and commercialization of SDN. As we explain in Chapter 6, there are a number of proponents of SDN that offer SDN solutions that are not based on OpenFlow. For the ONF, however, OpenFlow remains at the core of its SDN vision for the future. For this reason the ONF has become the responsible entity for the evolving OpenFlow specification. Starting after the release of V.1.1, revisions to the OpenFlow specification have been released and managed by the ONF.

One could get the impression from the fanfare surrounding OpenFlow that the advent of this technology has been accompanied by concomitant innovation in switching hardware. The reality is a bit more complicated. The OpenFlow designers realized a number of years ago that many switches were really built around ASICs controlled by rules encoded in tables that could be programmed. Over time, fewer homegrown versions of these switching chips were being developed, and there was greater consolidation in the semiconductor industry. More manufacturers' switches were based on ever-consolidating switching architecture and programmability, with ever-increasing use of programmable switching chips from a relatively small number of merchant silicon vendors. OpenFlow is an attempt to allow the programming, in a generic way, of the various implementations of switches that conform to this new paradigm. OpenFlow attempts to exploit the table-driven design extant in many of the current silicon solutions. As the number of silicon vendors consolidates, there should be a greater possibility for alignment with future OpenFlow versions.

It is worth pausing here to remark on the fact that we are talking a lot about ASICs for a technology called *Software* Defined Networking. Yet hardware must be part of the discussion, since it is necessary to use this specialized silicon in order to switch packets at high line rates. We explained in Chapter 4 that though pure software SDN implementations exist, they cannot switch packets at sufficiently high rates to keep up with high-speed interfaces. What is really meant by the word *software* in the name SDN, then, is that the SDN devices are fully programmable, not that everything is done using software running on a traditional CPU.

The sections that follow introduce the formal terminology used by OpenFlow and provide basic background that will allow us to explore the details of the different versions of the OpenFlow specification that have been released up to the time of the writing of this book.

5.2.1 The OpenFlow Switch

Figure 5.2 depicts the basic functions of an OpenFlow V.1.0 switch and its relationship to a controller. As would be expected in a packet switch, we see that the core function is to take packets that arrive on one port (path X on port 2 in the figure) and forward it through another port (port N in the figure), making any necessary packet modifications along the way. A unique aspect of the OpenFlow switch is embodied in the *packet-matching function* shown in Figure 5.2. The adjacent table is a *flow table*, and we give separate treatment to this in Section 5.3.2. The wide, gray, double arrow in Figure 5.2 starts in the decision logic, shows a match with a particular entry in that table, and directs the now-matched packet to an *action box* on the right. This action box has three fundamental options for the disposition of this arriving packet:

- $\boldsymbol{A}$. Forward the packet out a local port, possibly modifying certain header fields first.
- $\boldsymbol{B}$. Drop the packet.
- $\boldsymbol{C}$. Pass the packet to the controller.

These three fundamental packet paths are illustrated in Figure 5.2. In the case of path C, the packet is passed to the controller over the *secure channel* shown in the figure. If the controller has either a control message or a data packet to give to the switch, the controller uses this same secure channel in the reverse direction. When the controller has a data packet to forward out through the switch, it uses the OpenFlow PACKET_OUT message. We see in Figure 5.2 that such a data packet coming from the controller may take two different paths through the OpenFlow logic, both denoted Y. In the rightmost case, the controller directly specifies the output port and the packet is passed to that port N

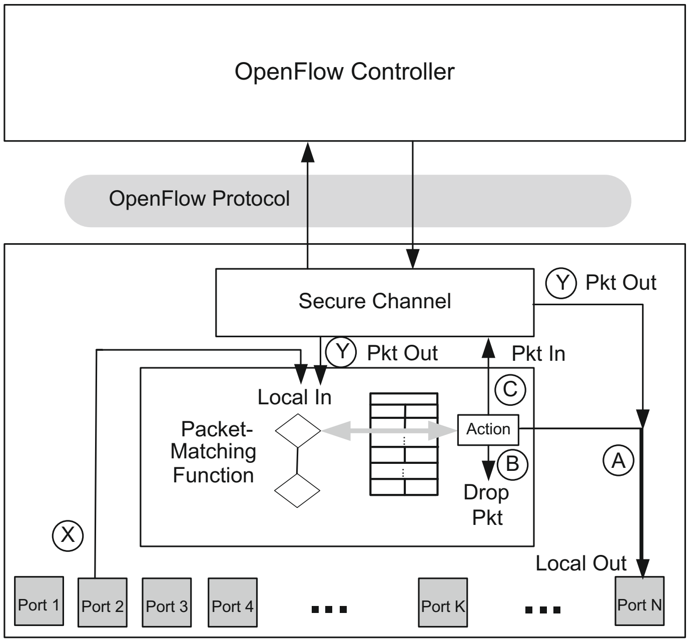


FIGURE 5.2

OpenFlow V.1.0 switch.

in the example. In the leftmost path *Y* case, the controller indicates that it wants to defer the forwarding decision to the packet-matching logic. Section 5.3.4 demonstrates that the controller dictates this by stipulating the virtual port TABLE as the output port.

A given OpenFlow switch implementation is either *OpenFlow-only* or *OpenFlow-hybrid*. An OpenFlow-only switch is one that forwards packets *only* according to the OpenFlow logic described above. An OpenFlow hybrid is a switch that can also switch packets in its legacy mode as an Ethernet switch or IP router. One can view the hybrid case as an OpenFlow switch residing next to a completely independent traditional switch. Such a hybrid switch requires a preprocessing classification mechanism that directs packets to either OpenFlow processing or the traditional packet processing. It is probable that hybrid switches will be the norm during the migration to pure OpenFlow implementations.

Note that we use the term OpenFlow *switch* in this chapter instead of the term OpenFlow *device* we customarily use. This is because switch is the term used in the OpenFlow specification. In general, though, we opt to use the term *device*, since there are already nonswitch devices being controlled by OpenFlow controllers, such as wireless access points.

5.2.2 The OpenFlow Controller

Modern Internet switches make millions of decisions per second about whether or not to forward an incoming packet, to what set of output ports it should be forwarded, and what header fields in the packet may need to be modified, added, or removed. This is a very complex task. The fact that this can be carried out at line rates on multigigabit media is a technological wonder. The switching industry has long understood that not all functions on the switching datapath can be carried out at line rates, so there has long been the notion of splitting the pure data plane from the control plane. The data plane matches headers, modifies packets, and forwards them based on a set of forwarding tables and associated logic, and it does this very, very fast. The rate of decisions being made as packets stream into a switch through a 100 Gbps interface is astoundingly high. The control plane runs routing and switching protocols and other logic to determine what the forwarding tables and logic in the data plane should be. This process is very complex and cannot be done at line rates as the packets are being processed, and it is for this reason we have seen the control plane separated from the data plane, even in legacy network switches.

The OpenFlow control plane differs from the legacy control plane in three key ways. First, it can program different data plane elements with a common, standard language, OpenFlow. Second, it exists on a separate hardware device than the forwarding plane, unlike traditional switches, where the control plane and data plane are instantiated in the same physical box. This separation is made possible because the controller can program the data plane elements remotely over the Internet. Third, the controller can program multiple data plane elements from a single control plane instance.

The OpenFlow controller is responsible for programming all the packet-matching and forwarding *rules* in the switch. Whereas a traditional router would run routing algorithms to determine how to program its forwarding table, that function or *an equivalent replacement* to it is now performed by the controller. Any changes that result in recomputing routes will be programmed onto the switch by the controller.

5.2.3 The OpenFlow Protocol

As shown in Figure 5.2, the OpenFlow protocol defines the communication between an OpenFlow controller and an OpenFlow switch. This protocol is what most uniquely identifies OpenFlow technology. At its essence, the protocol consists of a set of messages that are sent from the controller to the switch and a corresponding set of messages that are sent in the opposite direction. Collectively the messages allow the controller to program the switch so as to allow fine-grained control over the switching of user traffic. The most basic programming defines, modifies, and deletes flows. Recall that in Chapter 4 we defined a flow as *a set of packets transferred from one network endpoint (or set of endpoints) to another endpoint (or set of endpoints). The endpoints may be defined as IP address-TCP/UDP port pairs, VLAN endpoints, layer three tunnel endpoints, or input ports, among other things.* One set of rules describes the forwarding actions that the device should take for all packets belonging to that flow. When the controller defines a flow, it is providing the switch with the information it needs to know how to treat incoming packets that match that flow. The possibilities for treatment have grown more complex as the OpenFlow protocol has evolved, but the most basic prescriptions for treatment of an incoming packet are denoted by paths A, B, and C in Figure 5.2. These three options are to forward the packet out one or more output ports, drop the packet, or pass the packet to the controller for exception handling.

The OpenFlow protocol has evolved significantly with each version of OpenFlow, so we cover the detailed messages of the protocol in the version-specific sections that follow. The specification has evolved from development point release 0.2.0 on March 28, 2008, through release V.1.3.0, released in 2012. Numerous point releases over the intervening years have addressed problems with earlier releases and added incremental functionality. OpenFlow was viewed primarily as an experimental platform in its early years. For that reason, there was little concern on the part of the development community in advancing this standard to provide for interoperability between releases. As OpenFlow began to see more widespread commercial deployment, backward compatibility has become an increasingly important issue. Many features, however, were introduced in earlier versions of OpenFlow that are no longer present in the current version. Since the goal of this chapter is to provide a road map to understanding OpenFlow as it exists today, we take a hybrid approach to covering the major releases that have occurred since V.1.0. We focus on those key components of each release that became the basis for the advances in subsequent releases and do not focus on functionality in earlier releases that has subsequently been subsumed by new features.

5.2.4 The Controller-Switch Secure Channel

The secure channel is the path used for communications between the OpenFlow controller and the OpenFlow device. Generally, this communication is secured by TLS-based asymmetrical encryption, though unencrypted TCP connections are allowed. These connections may be *in-band* or *out-of-band*. Figure 5.3 depicts these two variants of the secure channel. In the out-of-band example, we see in the figure that the secure channel connection enters the switch via port *Z*, which is *not* switched by the OpenFlow data plane. Some legacy network stack will deliver the OpenFlow messages via the secure

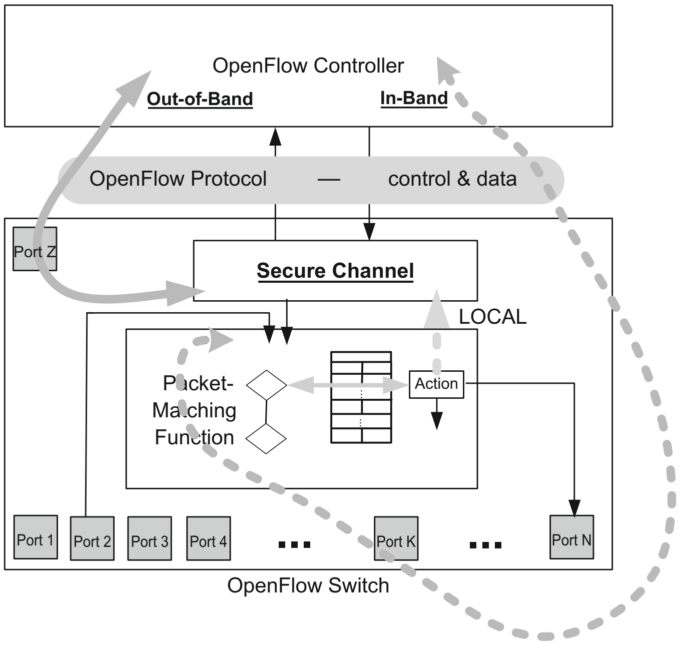


FIGURE 5.3

OpenFlow controller-switch secure channel.

channel to the secure channel process in the switch, where all OpenFlow messages are parsed and handled. Thus, the out-of-band secure channel is relevant only in the case of an OpenFlow-hybrid switch.

In the in-band example, we see the OpenFlow messages from the controller arriving via port K, which is part of the OpenFlow data plane. In this case these packets will be handled by the OpenFlow packet-matching logic shown in the figure. The flow tables will have been constructed so that this OpenFlow traffic is forwarded to the LOCAL virtual port, which results in the messages being passed to the secure channel process. We discuss the LOCAL virtual port in Section 5.3.4.

Note that when the controller and all the switches it controls are located entirely within a tightly controlled environment such as a data center, it may be wise to consider not using TLS-based encryption to secure the channel. This is because a performance overhead is incurred by using this type of security, and if it is not necessary, it is preferable to not pay this performance penalty.

5.3 OpenFlow 1.0 and OpenFlow Basics

OpenFlow 1.0 [1] was released on December 31, 2009. For the purposes of this work, we treat OpenFlow 1.0 as the initial release of OpenFlow. Indeed, years of work and multiple point releases preceded the OpenFlow 1.0 release, but we subsume all this incremental progress into the single initial release of 1.0 as though it had occurred atomically. In this section we describe in detail the basic components of this initial OpenFlow implementation.

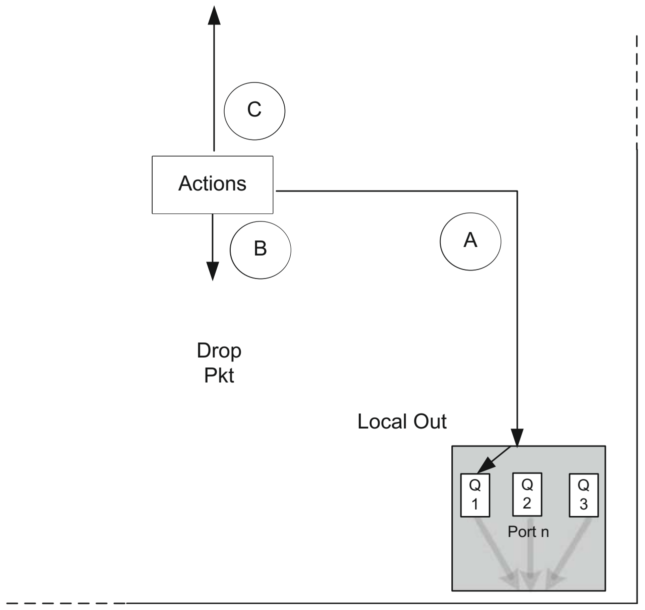


FIGURE 5.4

OpenFlow support for multiple queues per port.

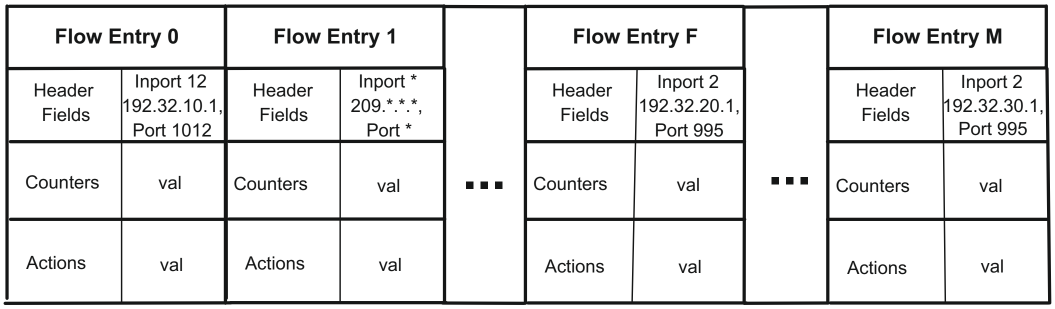
5.3.1 Ports and Port Queues

The OpenFlow specification defines the concept of an *OpenFlow port*. An OpenFlow V.1.0 port corresponds to a physical port. This concept is expanded in subsequent releases of OpenFlow. For many years, sophisticated switches have supported multiple *queues* per physical port. These queues are generally served by scheduling algorithms that allow the provisioning of different *quality of service* (QoS) levels for different types of packets. OpenFlow embraces this concept and permits a flow to be mapped to an already defined queue at an output port. Thus, if we look back to Figure 5.2, the output of a packet on port N may include specifying onto which queue on port N the packet should be placed. Thus, if we zoom in on option A in Figure 5.2, we reveal what we now see in Figure 5.4. In our zoomed-in figure we can see that the *actions* box specifically enqueued the packet being processed to queue 1 in port N.

Note that the support for QoS is very basic in V.1.0. QoS support in OpenFlow was expanded considerably in later versions (see Section 5.6.3).

5.3.2 Flow Table

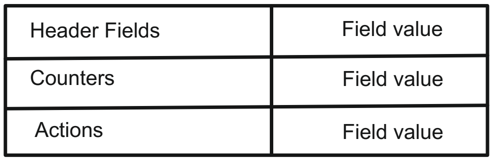
The *flow table* lies at the core of the definition of an OpenFlow switch. We depict a generic flow table in Figure 5.5. A flow table consists of *flow entries*, one of which is shown in Figure 5.6. A flow entry consists of *header fields*, *counters*, and *actions* associated with that entry. The header fields are used as match criteria to determine whether an incoming packet matches this entry. If a match exists, the packet belongs to this flow. The counters are used to track statistics relative to this flow, such as how many packets have been forwarded or dropped for this flow. The actions fields prescribe what the switch should do with a packet matching this entry. We describe this process of packet matching and actions in Section 5.3.3.



Flow Entry 0		Flow Entry 1			Flow Entry F			Flow Entry M	
Header Fields	Inport 12 192.32.10.1, Port 1012	Header Fields	Inport * 209.*.*.*, Port *		Header Fields	Inport 2 192.32.20.1, Port 995		Header Fields	Inport 2 192.32.30.1, Port 995
Counters	val	Counters	val	...	Counters	val	...	Counters	val
Actions	val	Actions	val		Actions	val		Actions	val

FIGURE 5.5

OpenFlow V.1.0 flow table.



Header Fields	Field value
Counters	Field value
Actions	Field value

FIGURE 5.6

Basic flow entry.

5.3.3 Packet Matching

When a packet arrives at the OpenFlow switch from an input port (or, in some cases, from the controller), it is matched against the flow table to determine whether there is a matching flow entry. The following match fields associated with the incoming packet may be used for matching against flow entries:

- Switch input port
- VLAN ID
- VLAN priority
- Ethernet source address
- Ethernet destination address
- Ethernet frame type
- IP source address
- IP destination address
- IP protocol
- IP *Type of Service* (ToS) bits
- TCP/UDP source port
- TCP/UDP destination port

These 12 match fields are collectively referred to as the basic 12-tuple of match fields. The flow entry's match fields may be wildcarded using a bit mask, meaning that any value that matches on the unmasked bits in the incoming packet's match fields will be a match. Flow entries are processed in order, and once a match is found, no further match attempts are made against that flow table. (We will see in subsequent versions of OpenFlow that there may be *additional* flow tables against which packet matching may continue.) For this reason, it is possible for there to be multiple matching flow entries for a packet to be present in a flow table. Only the first flow entry to match is meaningful; the others will not be found, because packet matching stops upon the first match.

The V.1.0 specification is silent about which of these 12 match fields are required versus those that are optional. The ONF has clarified this confusion by defining three different types of conformance in its V.1.0 conformance-testing program. The three levels are *full conformance*, meaning all 12 match fields are supported; *layer two conformance*, when only the layer two header field matching is supported; and, finally, *layer three conformance*, when only layer three header field matching is supported.

If the end of the flow table is reached without finding a match, this is called a *table miss*. In the event of a table miss in V.1.0, the packet is forwarded to the controller. (Note that this is no longer strictly true in later versions.) If a matching flow entry is found, the actions associated with that flow entry determine how the packet is handled. The most basic action prescribed by an OpenFlow switch entry is how to forward this packet. We discuss this concept in the following section.

It is important to note that this V.1.0 packet-matching function was designed as an abstraction of the way that real-life switching hardware works today. Early versions of OpenFlow were designed to specify the forwarding behavior of existing commercial switches via this abstraction. A good abstraction hides the details of the thing being abstracted while still permitting sufficiently fine-grained control to accomplish the needed tasks. As richer functionality is added in later versions of the OpenFlow protocol, we will see that the specification outpaces the reality of today's hardware. At that point, it is no longer providing a clean abstraction for current implementations but specifying behavior for switching hardware that has yet to be built.

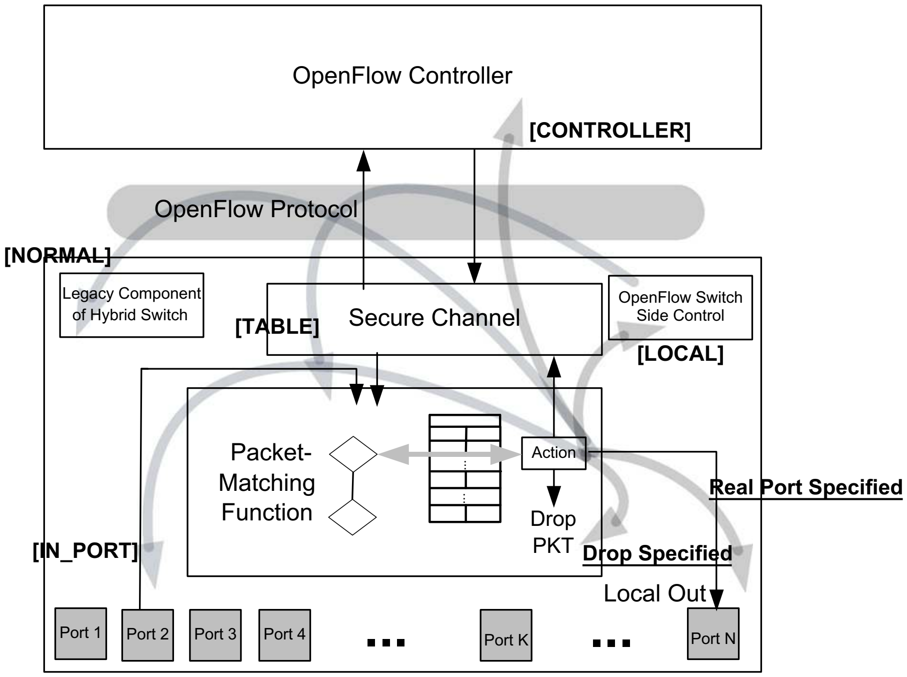


FIGURE 5.7

Packet paths corresponding to virtual ports.

5.3.4 Actions and Packet Forwarding

The required actions that must be supported by a flow entry are to either *output (forward)* or *drop* the matched packet. The most common case is that the output action specifies a physical port on which the packet should be forwarded. There are, however, five special virtual ports defined in V.1.0 that have special significance for the output action. They are *LOCAL*, *ALL*, *CONTROLLER*, *IN_PORT*, and *TABLE*. We depict the packet transfers resulting from these various virtual port designations in Figure 5.7. In the figure, the notations in brackets next to the wide shaded arrows represent one of the virtual port types discussed in this section.[1]

LOCAL dictates that the packet should be forwarded to the switch's local OpenFlow control software, circumventing further OpenFlow pipeline processing. LOCAL is used when OpenFlow messages from the controller are received on a port that is receiving packets switched by the OpenFlow data plane. LOCAL indicates that the packet needs to be processed by the local OpenFlow control software.

ALL is used to flood a packet out all ports on the switch except the input port. This provides rudimentary broadcast capability to the OpenFlow switch.

CONTROLLER indicates that the switch should forward this packet to the OpenFlow controller.

[1]The ALL and FLOOD virtual ports do not appear in the figure, for reasons explained later in this section.

IN_PORT instructs the switch to forward the packet back out of the port on which it arrived. Effectively, IN_PORT normally creates a loopback situation, which could be useful for certain scenarios. One scenario in which this is useful is the case of an 802.11 wireless port. In this case, it is quite normal to receive a packet from that port from one host and to forward it to the receiving host via the same port. This needs to be done very carefully so as not to create unintended loopback situations. Thus the protocol requires explicit stipulation of this intent via this special virtual port.

Another instance when IN_PORT is required is *edge virtual bridging* (EVB) [9]. EVB defines a reflective relay service between a physical switch in a data center and a lightweight virtual switch within the server known as a *virtual edge port aggregator* (VEPA). The standard IEEE 802.1Q bridge at the edge of the network will reflect packets back out the port on which they arrive to allow two virtual machines on the same server to talk to one another. This reflective relay service can be supported by the IN_PORT destination in OpenFlow rules.

Finally, there is the TABLE virtual port, which only applies to packets that the controller sends to the switch. Such packets arrive as part of the PACKET_OUT message from the controller, which includes an *action list*. This action list will generally contain an output action, which will specify a port number. The controller may want to directly specify the output port for this data packet, or, if it wants the output port to be determined by the normal OpenFlow packet-processing pipeline, it may do so by stipulating TABLE as the output port. These two options are depicted in the two *Y* paths shown in Figure 5.2.

There are two additional virtual ports, but support for these is optional in V.1.0. The first is the NORMAL virtual port. When the output action forwards a packet to the NORMAL virtual port, it sends the packet to the legacy forwarding logic of the switch. We contrast this with the LOCAL virtual port, which designates that the packet be passed to the local OpenFlow control processing. Conversely, packets for which matching rules indicate NORMAL as the output port will remain in the ASIC to be looked up in other forwarding tables that are populated by the local (non-OpenFlow) control plane. Use of NORMAL makes sense only in the case of a hybrid switch.

The remaining virtual port is FLOOD. In this case, the switch sends a copy of the packet out all ports except the ingress port.

Note that we have excluded the ALL and FLOOD representations from Figure 5.7 because the number of arrows would have overly congested the portrayal. The reader should understand that ALL and FLOOD would show arrows to all ports on the OpenFlow switch except the ingress port. Also note that the figure depicts the more normal *non-virtual* cases of *real port specified* and *drop specified*, underlined without brackets. When the V.1.0 switch passes a packet to the controller as a result of finding no table matches, the path taken in Figure 5.7 is the same as that denoted for the virtual port CONTROLLER.

There are two optional actions in V.1.0: *enqueue* and *modify field*. The enqueue action selects a specific queue belonging to a particular port. This would be used in conjunction with the output action and is used to achieve desired QoS levels via the use of multiple priority queues on a port. Finally, the modify-field action informs the switch how to modify certain header fields. The specification contains a lengthy list of fields that may be modified via this action. In particular, VLAN headers, Ethernet source and destination address, IPv4 source and destination address, and TTL field may be modified. Modify field is essential for the most basic routing functions. To route layer three packets, a router must decrement the TTL field before forwarding the packet out the designated output port.

When there are multiple actions associated with a flow entry, they appear in an action list, just like the PACKET_OUT message described above. The switch must execute actions in the order in which they appear on the action list. We walk through a detailed example of V.1.0. packet forwarding in Section 5.3.7.

5.3.5 Messaging Between Controller and Switch

The messaging between the controller and switch is transmitted over a secure channel. This secure channel is implemented via an initial TLS connection over TCP. (Subsequent versions of OpenFlow allow for multiple connections within one secure channel.) If the switch knows the IP address of the controller, the switch will initiate this connection. Each message between controller and switch starts with the OpenFlow header. This header specifies the OpenFlow version number, the message type, the length of the message, and the transaction ID of the message. The various message types in V.1.0 are listed in Table 5.1. The messages fall into three general categories: *symmetric*, *controller-switch*, and *async*. We explain the categories of messages shown in Table 5.1 in the following paragraphs. We suggest that the reader refer to Figure 5.8 as we explain each of these messages. Figure 5.8 shows the most important of these messages in a normal context and illustrates whether it normally is used during the *initialization*, *operational*, or *monitoring* phases of the controller-switch dialogue. The operational

Table 5.1 OFPT Message Types in OpenFlow 1.0

Message Type	Category	Subcategory
HELLO	Symmetric	Immutable
ECHO_REQUEST	Symmetric	Immutable
ECHO_REPLY	Symmetric	Immutable
VENDOR	Symmetric	Immutable
FEATURES_REQUEST	Controller-switch	Switch configuration
FEATURES_REPLY	Controller-switch	Switch configuration
GET_CONFIG_REQUEST	Controller-switch	Switch configuration
GET_CONFIG_REPLY	Controller-switch	Switch configuration
SET_CONFIG	Controller-switch	Switch configuration
PACKET_IN	Async	NA
FLOW_REMOVED	Async	NA
PORT_STATUS	Async	NA
ERROR	Async	NA
PACKET_OUT	Controller-switch	Cmd from controller
FLOW_MOD	Controller-switch	Cmd from controller
PORT_MOD	Controller-switch	Cmd from controller
STATS_REQUEST	Controller-switch	Statistics
STATS_REPLY	Controller-switch	Statistics
BARRIER_REQUEST	Controller-switch	Barrier
BARRIER_REPLY	Controller-switch	Barrier
QUEUE_GET_CONFIG_REQUEST	Controller-switch	Queue configuration
QUEUE_GET_CONFIG_REPLY	Controller-switch	Queue configuration

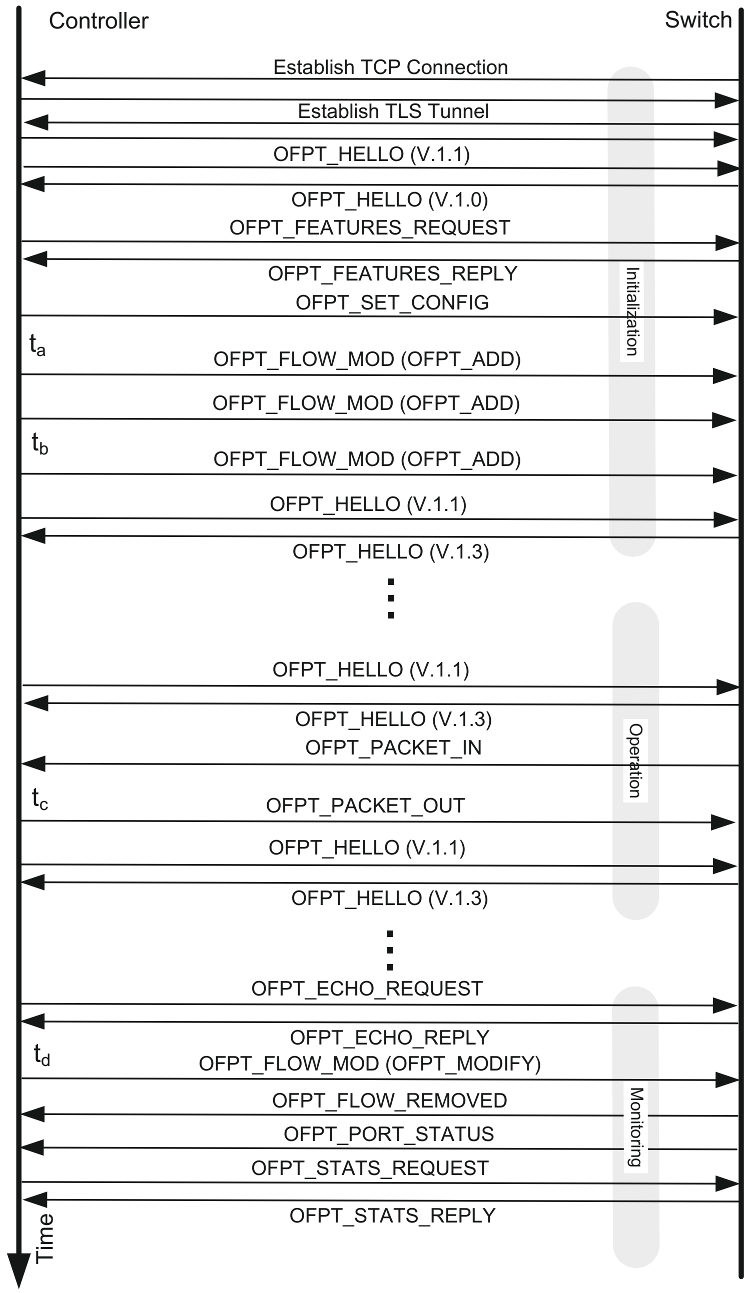


FIGURE 5.8

Controller-switch protocol session.

and monitoring phases generally overlap, but for the sake of clarity we show them as disjoint in the figure. In the interest of brevity, we have truncated the OFPT_ prefix from the message names shown in Table 5.1. We follow this convention whenever we refer to these message names in the balance of the book; however, they appear in index with their full name, including their OFPT_ prefix.

Symmetric messages may be sent by either the controller or the switch without having been solicited by the other. The HELLO messages are exchanged after the secure channel has been established to determine the highest OpenFlow version number supported by the peers. The protocol specifies that the lower of the two versions is to be used for controller-switch communication over this secure channel instance. ECHO messages are used by either side during the life of the channel to ascertain that the connection is still alive and to measure the current latency or bandwidth of the connection. The VENDOR messages are available for vendor-specific experimentation or enhancements.

Async messages are sent from the switch to the controller without having been solicited by the controller. The PACKET_IN message is the way the switch passes data packets back to the controller for exception handling. Control plane traffic will usually be sent back to the controller via this message. The switch can inform the controller that a flow entry is removed from the flow table via the FLOW_REMOVED message. PORT_STATUS is used to communicate changes in port status, whether by direct user intervention or by a physical change in the communications medium itself. Finally, the switch uses the ERROR message to notify the controller of problems.

Controller-switch is the broadest category of OpenFlow messages. In fact, as shown in Table 5.1, they can be divided into five subcategories: *switch configuration*, *command from controller*, *statistics*, *queue configuration*, and *barrier*. The switch configuration messages consist of a unidirectional configuration message and two request-reply message pairs. The controller uses the unidirectional message, SET_CONFIG to set configuration parameters in the switch. In Figure 5.8 we see the SET_CONFIG message sent during the initialization phase of the controller-switch dialogue. The controller uses the FEATURES message pair to interrogate the switch about which features it supports. Similarly, the GET_CONFIG message pair is used to retrieve a switch's configuration settings.

There are three messages comprising the *command from controller* category. PACKET_OUT is the analog of the PACKET_IN mentioned above. The controller uses PACKET_OUT to send data packets to the switch for forwarding out through the data plane. The controller modifies existing flow entries in the switch via the FLOW_MOD message. PORT_MOD is used to modify the status of an OpenFlow port.

Statistics are obtained from the switch by the controller via the STATS message pair. The BARRIER message pair is used by the controller to ensure that a particular OpenFlow command from the controller has finished executing on the switch. The switch must complete execution of all commands received prior to the BARRIER_REQUEST before executing any commands received after it, and the switch notifies the controller of having completed such preceding commands via the BARRIER_REPLY message sent back to the controller.

The queue configuration message pair is somewhat of a misnomer in that actual queue configuration is beyond the scope of the OpenFlow specification and is expected to be done by an unspecified out-of-band mechanism. The QUEUE_GET_CONFIG_REQUEST and QUEUE_GET_CONFIG_REPLY message pair is the mechanism by which the controller learns from the switch how a given queue is configured. With this information, the controller can intelligently map certain flows to specific queues to achieve desired QoS levels.

Note that *immutable* in this context means that the message types will not be changed in future releases of OpenFlow. (*Author's note:* It is somewhat remarkable that the immutable characteristic is cited as distinct in OpenFlow. In many more mature communications protocols, maintaining backward compatibility is of paramount importance. This is not the case with OpenFlow, where support for some message formats in earlier versions is dropped in later versions. This situation is handled within the OpenFlow environment by the fact that two OpenFlow implementations coordinate their version numbers via the HELLO protocol, and presumably the higher-versioned implementation reverts to the older version of the protocol for the sake of interoperability with the other device.)

In the event that the HELLO protocol detects a loss of the connection between controller and switch, the V.1.0 specification prescribes that the switch should enter *emergency mode* and reset the TCP connection. At this time all flows are to be deleted except special flows that are marked as being part of the *emergency flow cache*. The only packet matching that is allowed in this mode is against those flows in that emergency flow cache. The specification did not specify which flows that should be in this cache, and subsequent versions of OpenFlow have addressed this area differently and more thoroughly.

5.3.6 Example: Controller Programming Flow Table

In Figure 5.9 two simple OpenFlow V.1.0 flow table modifications are performed by the controller. We see in the figure that the initial flow table has three flows. In the quiescent state before time t_a, we see an exploded version of flow entry zero. It shows that the flow entry specifies that all Ethernet frames entering the switch on input port K with a destination Ethernet address of 0x000CF15698AD should be output on output port N. All other match fields have been wildcarded, indicated by the asterisks in their respective match fields in Figure 5.9. At time t_a, the controller sends a FLOW_MOD (ADD) command to the switch, adding a flow for packets entering the switch on any port, with source IP addresses 192.168.1.1 and destination IP address 209.1.2.1, source TCP port 20, and destination port 20. All other match fields have been wildcarded. The outport port is specified as P. We see that after this controller command is received and processed by the switch, the flow table contains a new flow entry F corresponding to that ADD message. At time t_b, the controller sends a FLOW_MOD (MODIFY) command for flow entry zero. The controller seeks to modify the corresponding flow entry such that there is a one-hour (3600-second) idle time on that flow. The figure shows that after the switch has processed this command, the original flow entry has been modified to reflect that new idle time. Note that idle time for a flow entry means that after that number of seconds of inactivity on that flow, the flow should be deleted by the switch. Looking back at Figure 5.8, we see an example of such a flow expiration just after time t_d. The FLOW_REMOVED message we see there indicates that the flow programmed at time t_b in our examples in Figures 5.8 and 5.9 has expired. This controller had requested to be notified of such expiration when the flow was configured, and the FLOW_REMOVED messages serves this purpose.

5.3.7 Example: Basic Packet Forwarding

We illustrate the most basic case of OpenFlow V.1.0 packet forwarding in Figure 5.10. The figure depicts a packet arriving at the switch through port 2 with source IPv4 address of 192.168.1.1 and destination IPv4 address of 209.1.2.1. The packet-matching function scans the flow table starting at flow entry 0

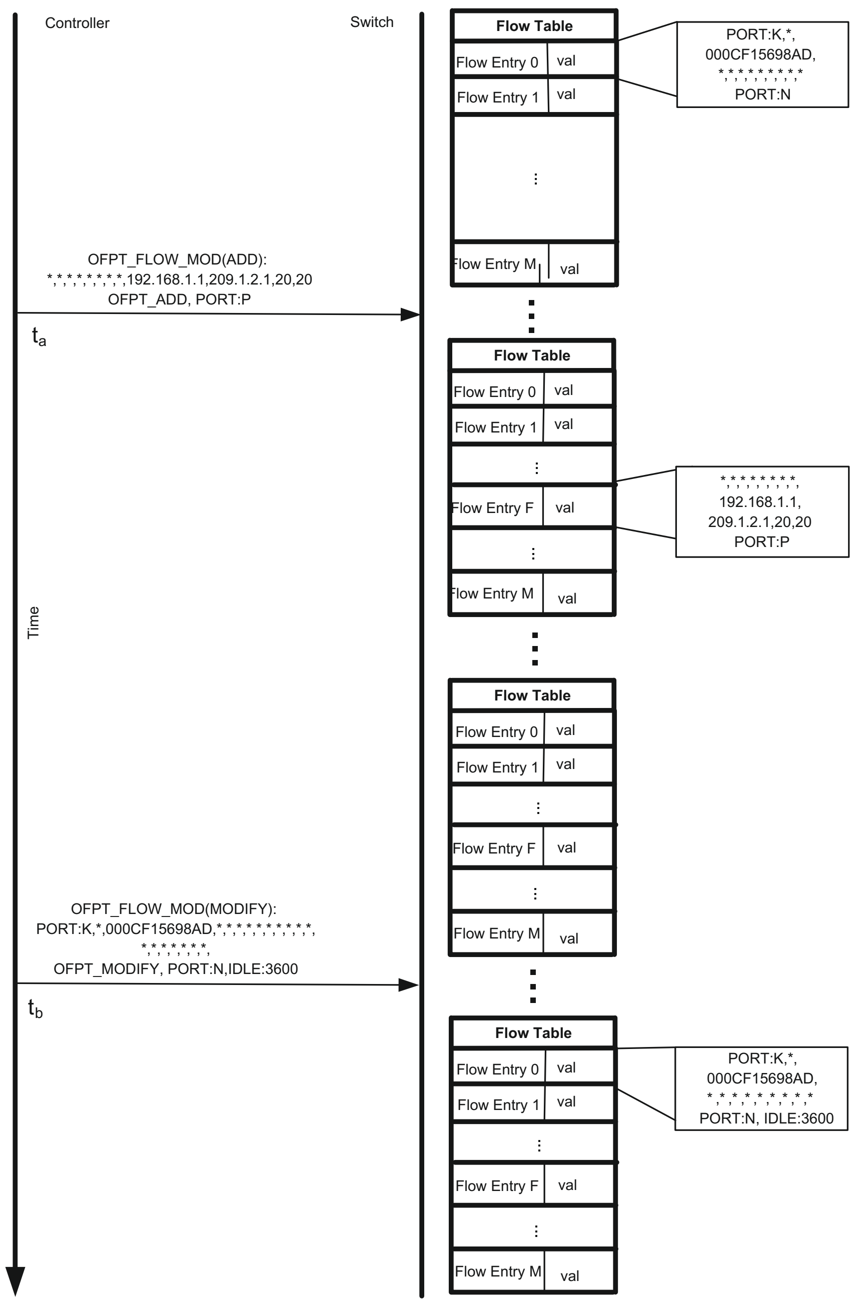

Controller
Switch
Time
Flow Table
Flow Entry 0
Flow Entry 1
Flow Entry F
Flow Entry M
val
PORT:K,*, 000CF15698AD, *,*,*,*,*,*,*,*,* PORT:N
OFPT_FLOW_MOD(ADD): *,*,*,*,*,*,*,*,192.168.1.1,209.1.2.1,20,20 OFPT_ADD, PORT:P
ta
,,*,*,*,*,*,*, 192.168.1.1, 209.1.2.1,20,20 PORT:P
OFPT_FLOW_MOD(MODIFY): PORT:K,*,000CF15698AD,*,*,*,*,*,*,*,*,*,*, *,*,*,*,*,*,*, OFPT_MODIFY, PORT:N,IDLE:3600
tb
PORT:K,*, 000CF15698AD, *,*,*,*,*,*,*,*,* PORT:N, IDLE:3600

FIGURE 5.9

Controller programming flow entries in V.1.0.

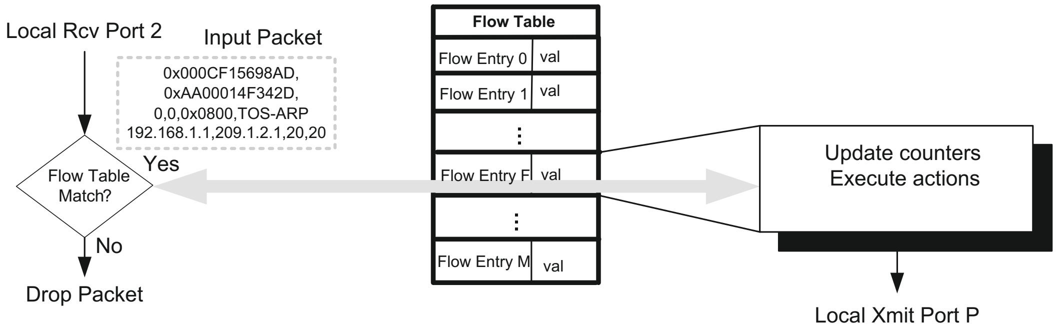


FIGURE 5.10

Packet matching function: basic packet forwarding V.1.0.

and finds a match in flow entry F. Flow entry F stipulates that a matching packet should be forwarded out port P. The switch does this, completing this simple forwarding example.

An OpenFlow switch forwards packets based on the header fields it matches. The network programmer designates layer three switch behavior by programming the flow entries to try to match layer three headers such as IPv4. If it is a layer two switch, the flow entries will dictate matching on layer two headers. The semantics of the flow entries allow matching for a wide variety of protocol headers but a given switch will be programmed only for those that correspond to the role it plays in packet forwarding. Whenever there is overlap in potential matches of flows, the priority the controller assigns the flow entry will determine which match takes precedence. For example, if a switch is both a layer two and a layer three switch, placing layer three header-matching flow entries at a higher priority would ensure that if possible, layer three switching will be done on that packet.

5.3.8 Example: Switch Forwarding Packet to Controller

We showed in the last section an example of an OpenFlow switch forwarding an incoming data packet to a specified destination port. Another fundamental action of the OpenFlow V.1.0 switch is to forward packets to the controller for exception handling. The two reasons for which the switch may forward a packet to the controller are OFPR_NO_MATCH and OFPR_ACTION. Obviously OFPR_NO_MATCH is used when no matching flow entry is found. OpenFlow retains the ability to specify that a particular matching flow entry should always be forwarded to the controller. In this case OFPR_ACTION is specified as the reason. An example is a control packet such as a routing protocol packet that always needs to be processed by the controller. In Figure 5.11 we show an example of an incoming data packet that is an OSPF routing packet. There is a matching table entry for this packet that specifies that the packet should be forwarded to the controller. We see in the figure that a PACKET_IN message is sent via the secure channel to the controller, handing off this routing protocol update to the controller for exception processing. The processing that would likely take place on the controller is that the OSPF routing protocol would be run, potentially resulting in a change to the forwarding tables in the switch. The controller could then modify the forwarding tables via the brute-force approach of sending FLOW_MOD commands to the switch modifying the output port for each flow in the switch affected by this routing

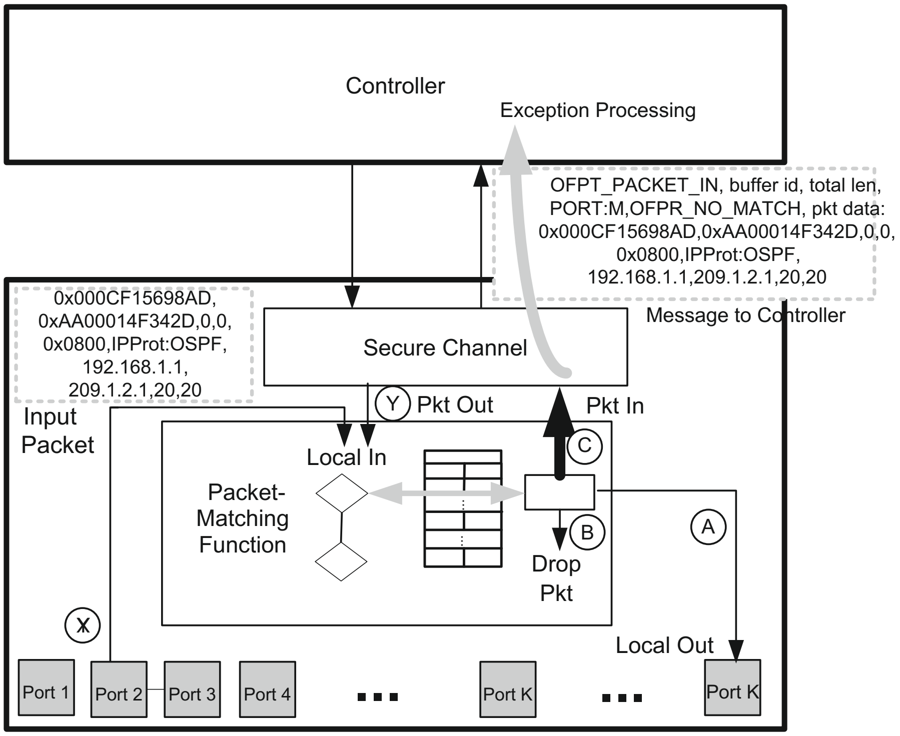

Controller
Exception Processing
OFPT_PACKET_IN, buffer id, total len, PORT:M,OFPR_NO_MATCH, pkt data: 0x000CF15698AD,0xAA00014F342D,0,0, 0x0800,IPProt:OSPF, 192.168.1.1,209.1.2.1,20,20
Message to Controller
0x000CF15698AD, 0xAA00014F342D,0,0, 0x0800,IPProt:OSPF, 192.168.1.1, 209.1.2.1,20,20
Secure Channel
Input Packet
Y
Pkt Out
Pkt In
Local In
C
Packet-Matching Function
B
A
Drop Pkt
X
Local Out
Port 1
Port 2
Port 3
Port 4
Port K
Port K

FIGURE 5.11

Switch forwarding incoming packet to controller.

table change. (Section 5.4.2 describes that there is a more efficient way for the controller to program the output port for flows in a layer three switch where multiple flows share the same next-hop IP address.)

At a minimum, the controller needs access to the packet header fields to determine its disposition of the packet. In many cases, though not all, it may need access to the entire packet. This would in fact be true in the case of the OSPF routing packet in this example. In the interest of efficiency, OpenFlow allows the optional buffering of the full packet by the switch. In the event of a large number of packets being forwarded from the switch to the controller, for which the controller only needs to examine the packet header, significant bandwidth efficiency gains are achieved by buffering the full packet in the switch and only forwarding the header fields. Since the controller will sometimes need to see the balance of the packet, a buffer ID is communicated with the PACKET_IN message. The controller may use this buffer ID to subsequently retrieve the full packet from the switch. The switch has the ability to age out old buffers that the switch has not retrieved.

There are other fundamental actions that the switch may take on an incoming packet: (1) to flood the packet out all ports except the port on which it arrived, or (2) to drop the packet. We trust that the reader's understanding of the drop and flood functions will follow naturally from the two examples just provided, so we now move onto discuss the extensions to the basic OpenFlow functions provided in V.1.1.

Table 5.2 Major New Features Added in OpenFlow 1.1

Feature Name	Description
Multiple flow tables	See Section 5.4.1
Groups	See Section 5.4.2
MPLS and VLAN tag support	See Section 5.4.3
Virtual ports	See Section 5.4.4
Controller connection failure	See Section 5.4.5

5.4 OpenFlow 1.1 Additions

OpenFlow 1.1 [2] was released on February 28, 2011. Table 5.2 lists the major new features added in this release. Multiple flow tables and group table support are the two most prominent. We highlight these new features in Figure 5.12. From a practical standpoint, V.1.1 had little impact other than as a stepping stone to V.1.2. This was because it was released just before the ONF was created, and the SDN community waited for the first version following the ONF transition (i.e., V.1.2) before creating implementations. It is important that we cover V.1.1 here, though, because the subsequent versions of OpenFlow are built on some of the V.1.1 feature set.

5.4.1 Multiple Flow Tables

V.1.1 significantly augments the sophistication of packet processing in OpenFlow. The most salient shift is due to the addition of *multiple flow tables*. The concept of a single flow table remains much as it was in V.1.0; however, in V.1.1 it is now possible to defer further packet processing to subsequent matching in other flow tables. For this reason, it was necessary to break the execution of actions from its direct association with a flow entry. With V.1.1 the new *instruction* protocol object is associated with a flow entry. The processing pipeline of V.1.1 offered much greater flexibility than was available in V.1.0. This improvement derives from the fact that flow entries can be *chained* by an instruction in one flow entry pointing to another flow table. This is called a *GOTO* instruction. When such an instruction is executed, the packet-matching function depicted earlier in Figures 5.10 and 5.11 is invoked again, this time starting the match process with the first flow entry of the new flow table. This new pipeline is reflected in the expanded packet-matching function shown in Figure 5.12. The pipeline allows all the power of an OpenFlow V.1.1 instruction to test and *modify* the contents of a packet to be applied multiple times, with different conditions used in the matching process in each flow table. This allows dramatic increases in both the complexity of the matching logic as well as the degree and nature of the packet modifications that may take place as the packet transits the OpenFlow V.1.1 switch. V.1.1 packet processing, with different instructions chained through complex logic as different flow entries are matched in a sequence of flow tables, constitutes a robust packet-processing *pipeline*.[2]

[2]This increased robustness comes at a price. It has proven challenging to adapt existing hardware switches to support multiple flow tables. We discuss this further in Section 5.6. In Sections 12.9.3 and 13.2.7, we discuss attempts to specifically design hardware with this capability in mind.

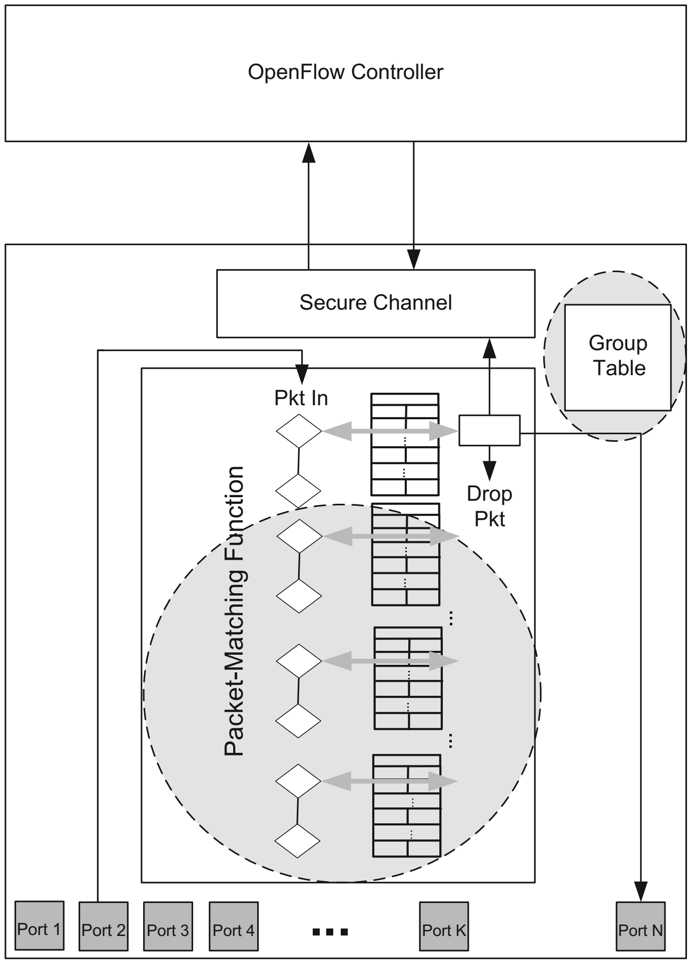


FIGURE 5.12

OpenFlow V.1.1 switch with expanded packet processing.

In Section 5.3.4 we described the actions supported in V.1.0 and the order in which they were executed. The V.1.1 instructions form the conduit that controls the actions that are taken and in what order. They can do this in two ways. First, they can add actions to an *action set*. The action set is initialized and modified by all the instructions executed during a given pass through the pipeline. Instructions delete actions from the action set or merge new actions into the action set. When the pipeline ends, the actions in the action set are executed in the following order:

1. Copy TTL inward
2. Pop

3. Push
4. Copy TTL outward
5. Decrement TTL
6. Set: Apply all set_field actions to the packet
7. QoS: Apply all QoS actions to the packet, such as set_queue
8. Group: If a group action is specified, apply the actions to the relevant action buckets (which can result in the packet being forwarded out the ports corresponding to those buckets)
9. Output: If no group action is specified, forward the packet out the specified port

The output action, if such an instruction exists (and it normally does), is executed last. This is logical because other actions often manipulate the contents of the packet, and this clearly must occur before the packet is transmitted. If there is neither a group nor output action, the packet is dropped. We explain the most important of the actions listed above in discussion and examples in the following sections.

The second way that instructions can invoke actions is via the *Apply-Actions* instruction. This is used to execute certain actions immediately between flow tables, while the pipeline is still active, rather than waiting for the pipeline to reach its end, which is the normal case. Note that the action list that is part of Apply-Actions has exactly the same semantics as the action list in the PACKET_OUT message, and both are the same as the action-list semantics of V.1.0.

In the case of the PACKET_OUT message being processed through the pipeline rather than the action list being executed upon a flow entry match, its component actions on the action list are merged into the action set being constructed for this incoming packet. If the current flow entry includes a GOTO instruction to a higher-numbered flow table, further action lists may be merged into the action set before it is ultimately executed.

When a matched flow entry does not specify a GOTO flow table, the pipeline processing completes, and whatever actions have been recorded in the action set are then executed. In the normal case, the final action executed is to forward the packet to an output port, to the controller, or to a *group* table, which we describe next.

5.4.2 Groups

V.1.1 offers the *group* abstraction as a richer extension to the FLOOD option. In V.1.1 there is the group table, which we first included in Figure 5.12 and is shown in detail in Figure 5.13. The group table consists of *group entries*, each entry consisting of one or more *action buckets*. The buckets in a group have actions associated with them that get applied before the packet is forwarded to the port defined by that bucket. Refinements on flooding, such as multicast, can be achieved in V.1.1 by defining groups as specific sets of ports. Sometimes a group may be used when there is only a single output port, as illustrated in the case of group 2 in Figure 5.13. A use case for this would be when many flows all should be directed to the same next-hop switch and it is desirable to retain the ability to change that next hop with a single configuration change. This can be achieved in V.1.1 by having all the designated flows pointing to a single group entry that forwards to the single port connected to that next hop. If the controller wants to change that next hop due to a change in IP routing tables in the controller, all the flows can be rerouted simply by reprogramming the single group entry. This provides a more efficient way of handling the routing change from the OSPF update example presented earlier, in Section 5.3.8. Clearly, changing a single group entry's action bucket is faster than updating the potentially large number of

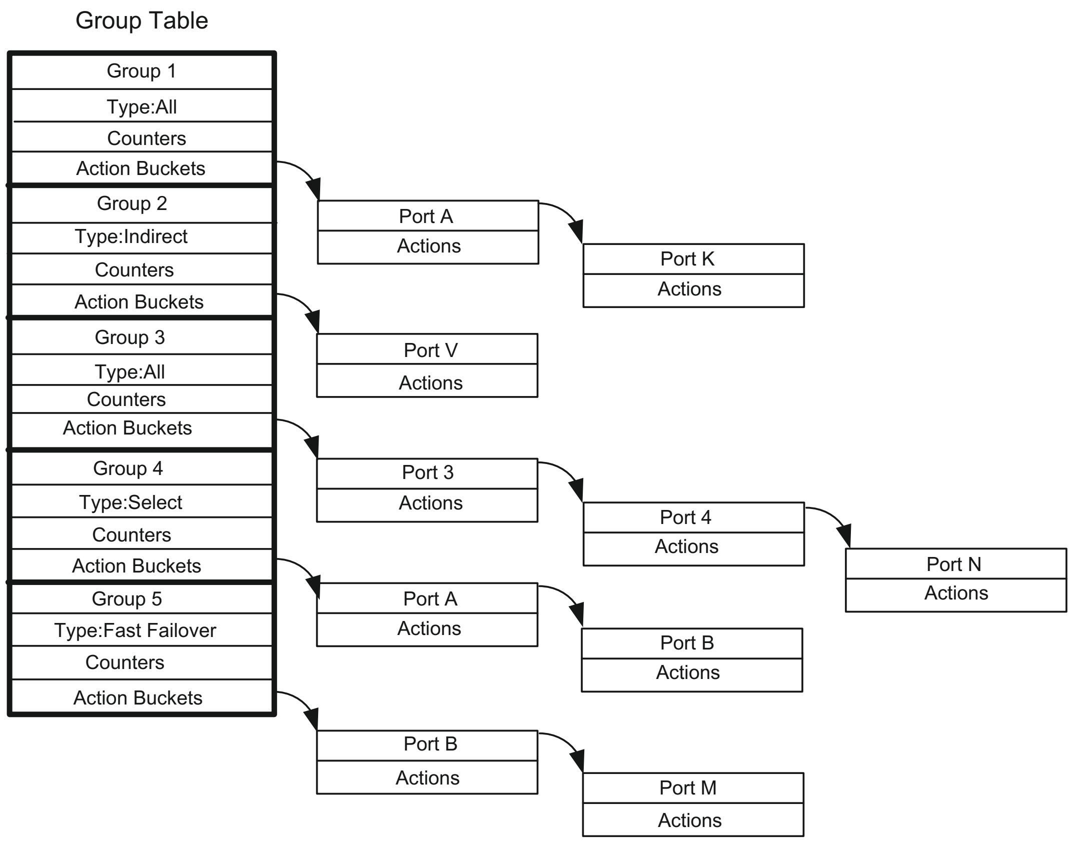


FIGURE 5.13

OpenFlow V.1.1 group table.

flow entries whose next hop changed as a result of a single routing update. Note that this is a fairly common scenario. Whenever a link or neighbor fails or is taken out of operation, *all flows* traversing that failed element need to be rerouted by the switch that detects that failure.

Note that one group's buckets may forward to other groups, providing the capability to chain groups together.

5.4.3 MPLS and VLAN Tag Support

V.1.1 is the first OpenFlow version to provide full VLAN support. Since providing complete support for multiple levels of VLAN tags required robust support for the *popping* and *pushing* of multiple levels of tags, support for MPLS tagging followed naturally and is also part of V.1.1. For the interested reader unfamiliar with MPLS technology, we provide a good reference in [5]. Both the new PUSH and POP actions as well as the chaining of flow tables were necessary to provide this generic support for VLANs and MPLS. When a PUSH action is executed, a new header of the specified type is inserted in front of the current outermost header. The field contents of this new header are initially copied from the corresponding existing fields in the current outermost header, should one exist. If it does not exist, they are initialized to zero. New header values are then assigned to this new outermost header via subsequent SET actions. Whereas the PUSH is used to add a new tag, the POP is used to remove the current outermost tag. In addition to PUSHing and POPping, V.1.1 also permits modification of the current outermost tag, whether it be a VLAN tag or an MPLS shim header.

V.1.1 correctly claims full MPLS and VLAN tag support, but that support requires specification of very complex matching logic that has to be implemented when such tags are encountered in the incoming packet. The *extensible match support* that we introduce in Section 5.5.1 provides the more generalized semantics to the controller such that this complex matching logic in the switch can be disposed of. Since this V.1.1 logic is replaced by the more general solution in Section 5.5.1, we will not confuse the reader by providing details about it here.

5.4.4 Virtual Ports

In V.1.0, the concept of an output port mapped directly to a physical port, with some limited use of *virtual ports*. Though the TABLE and other virtual ports did exist in earlier versions of OpenFlow, the concept of virtual ports was augmented in V.1.1. A V.1.1 switch classifies ports into the categories of *standard ports* and *reserved virtual ports*.

Standard ports consist of:

- **Physical ports.**
- **Switch-defined virtual ports.** In V.1.1 it is now possible for the controller to forward packets to an abstraction called *switch-defined virtual ports*. Such ports are used when more complex processing will be required on the packet than simple header field manipulation. One example is when the packet should be forwarded via a tunnel. Another use case for a virtual port is *link aggregation* (LAG). For more information about LAGs, refer to [7].

Reserved virtual ports consist of:

- **ALL.** This is the straightforward mechanism to flood packets out all standard ports except the port on which the packet arrived. This is similar to the effect provided by the optional FLOOD reserved virtual port.
- **CONTROLLER.** Forwards the packet to the controller in an OpenFlow message.
- **TABLE.** Processes the packet through the normal OpenFlow pipeline processing. This only applies to a packet that is being sent from the controller (via a PACKET_OUT message). We explained the use of the TABLE virtual port in Section 5.3.4.
- **IN_PORT.** This provides a loopback function. The packet is sent back out on the port on which it arrived.
- **LOCAL (optional).** This optional port provides a mechanism whereby the packet is forwarded to the switch's local OpenFlow control software. Because a LOCAL port may be used both as an output port and an ingress port, LOCAL can be used to implement an in-band controller connection, obviating the need for a separate control network for the controller-switch connection. Refer to our earlier discussion on LOCAL in Section 5.3.3 for an example of how LOCAL may be used.

- **NORMAL (optional).** This directs the packet to the normal non-OpenFlow pipeline of the switch. NORMAL differs from the LOCAL reserved virtual port in that it may only be used as an output port.
- **FLOOD (optional).** The general use of this port is to send the packet out all standard ports except the port on which it arrived. Refer to the OpenFlow V.1.1 specification for the specific nuances of this reserved virtual port.

5.4.5 Controller Connection Failure

Loss of connectivity between the switch and controller is a serious and real possibility, and the OpenFlow specification needs to specify how it should be handled. The *emergency flow cache* was included in V.1.0 in order to handle such a situation, but support for this cache was dropped in V.1.1 and replaced with two new mechanisms, *fail secure mode* and *fail standalone mode*. The V.1.1 switch immediately enters one of these two modes upon loss of connection to the controller. Which of these two modes is entered will depend on which is supported by the switch or, if both are supported, by user configuration. In the case of fail secure mode, the switch continues to operate as a normal V.1.1 switch except that all messages destined for the controller are dropped. In the case of fail standalone mode, the switch additionally ceases its OpenFlow pipeline processing and continues to operate in its native, underlying switch or router mode. When the connection to the controller is restored, the switch resumes its normal operation mode. The controller, having detected the loss and restoration of the connection, may choose to delete existing flow entries and begin to configure the switch anew. (*Author's note:* This is another example of the aforementioned lack of backward compatibility in the OpenFlow specifications.)

5.4.6 Example: Forwarding with Multiple Flow Tables

Figure 5.14 expands on our earlier example of V.1.0 packet forwarding that was presented in Figure 5.10. Assuming that the incoming packet is the same as in Figure 5.10, we see in Figure 5.14 that there is a match in the second flow entry in flow table 0. Unlike in V.1.0, the pipeline does not immediately execute actions associated with that flow entry. Its counters are updated, and the newly initialized action set is updated with those actions programmed in that flow entry's instructions. One of those instructions is to continue processing at table K. We see this via the jump to processing at the letter A, where we resume the pipeline process matching the packet against table K. In this case, a matching flow entry is found in flow entry F. Once again, this match results in the flow entry's counters being updated and its instructions being executed. These instructions may apply further modifications to the action set that has been carried forward from table 0. In our example, an additional action $A3$ is merged into the action set. Since there is no GOTO instruction present this time, this represents the end of the pipeline processing, and the actions in the current action set are performed in the order specified by OpenFlow V.1.1. (The difference between action set and action list was explained earlier in Section 5.4.1.) The OUTPUT action, if it is present, is the last action performed before the pipeline ends. We see that the OUTPUT action was present in our example as the packet is forwarded out port P in Figure 5.14.

5.4.7 Example: Multicast Using V.1.1 Groups

We direct the reader to Figure 5.13, discussed previously. In this figure we see that group 3 has group type ALL and has three action buckets. This group configuration prescribes that a packet sent for processing

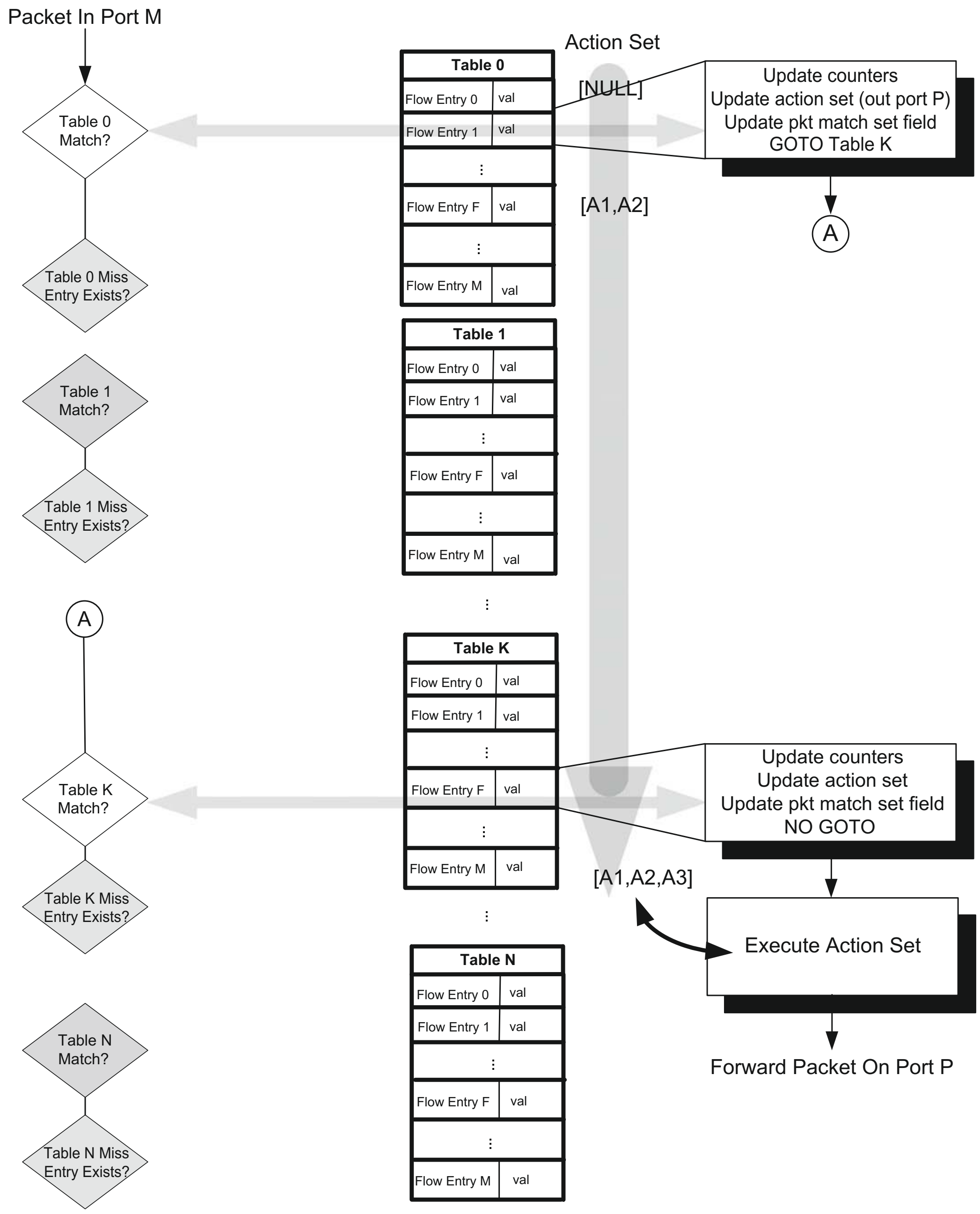


FIGURE 5.14

Packet matching function - basic packet forwarding, V.1.1.

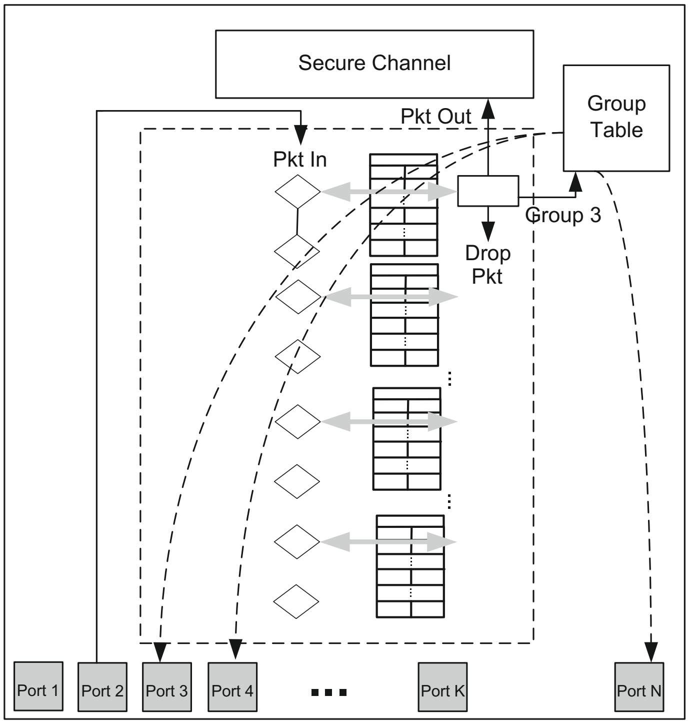


FIGURE 5.15

Multicast using group table in V.1.1.

by this group entry should have a copy sent out each of the ports in the three action buckets. Figure 5.15 shows that the packet-matching function directs the incoming packet to group 3 in the group table for processing. Assuming that the group 3 in Figure 5.15 is as shown in Figure 5.13, the packet is multicast out ports 3, 4, and N, as indicated by the dashed arrows in the figure.

5.5 OpenFlow 1.2 Additions

OpenFlow 1.2 [3] was released on December 5, 2011. Table 5.3 lists the major new features added in this release. We discuss the most important of these in the following sections.

5.5.1 Extensible Match Support

Due to the relatively narrow semantics in the packet matching prior to V.1.2, it was necessary to define complex flow diagrams that described the logic of how to perform packet parsing. The packet-matching capability provided in V.1.2 provides sufficient richness in the packet-matching descriptors that the

Table 5.3 Major New Features Added in OpenFlow 1.2

Feature Name	Description
Extensible match support	See Section 5.5.1
Extensible set_field packet-rewriting support	See Section 5.5.2
Extensible context expression in "packet-in"	See Section 5.5.3
Extensible error messages via experimenter error type	-
IPv6 support	See Section 5.5.2
Simplified behavior of flow-mod request	-
Removed packet parsing specification	See Section 5.5.1
Multiple controller enhancements	See Section 5.5.4

controller can encode the desired logic in the rules themselves. This obviates the earlier versions' requirement that the behavior be hardcoded into the switch logic.

A generic and extensible packet-matching capability has been added in V.1.2 via the *OpenFlow Extensible Match* (OXM) descriptors. OXM defines a set of *type-length-value* (TLV) pairs that can describe or define virtually any of the header fields an OpenFlow switch would need to use for matching. This list is too long to enumerate here, but in general any of the header fields that are used in matching for Ethernet, VLAN, MPLS, IPv4, and IPv6 switching and routing may be selected and a value (with bitmask wildcard capability) provided. Prior versions of OpenFlow had a much more static match descriptor, which limited flexibility in matches and made adding future extensions more difficult. An example is that in order to accommodate the earlier fixed structure, the single match field TCP Port was overloaded to also mean UDP Port or ICMP code, depending on the context. This confusion has been eliminated in V.1.2 via OXM. Because the V.1.2 controller can encode more descriptive parsing into the pipeline of flow tables, complex packet parsing does not need to be specified inside the V.1.2 switch.

This ability to match on any combination of header fields is provided within the OPENFLOW_BASIC *match class*. V.1.2 expands the possibilities for match fields by allowing for multiple match classes. Specifically, the EXPERIMENTER match class is defined, opening up the opportunity for matching on fields in the packet payload, providing a near-limitless horizon for new definitions of flows. The syntax and semantics of EXPERIMENTER are left open so that the number and nature of various fields are subject to the experimental implementation.

5.5.2 Extensible SET_FIELD Packet Rewriting Support

The V.1.2 switch continues to support the action types we saw in previous versions. A major enhancement provides the ability to set the value of *any* field in the packet header that may be used for matching. This is due to the fact that the same OXM encoding we described for enhanced packet matching is made available in V.1.2 for generalized setting of fields in the packet header. For example, IPv6 support falls out naturally from the fact that any of the fields that may be described by an OXM TLV may also be set using the set_field action. The ability to match an incoming IPv6 header against a flow entry and, when necessary, to set an IPv6 address to a new value together provide support for this major feature of V.1.2. CRC recalculation is performed automatically when a set-field action changes the packet's contents.

Since the EXPERIMENTER action type can be a modification or extension to any of the basic action types or even something totally novel, this allows for packet fields not part of the standard OXM header fields to be modified. In general, if a packet field may be used for matching in V.1.2, it may also be modified.

5.5.3 Extensible Context Expression in PACKET_IN

The OXM encoding is also used to extend the PACKET_IN message sent from the switch to the controller. In previous versions this message included the packet headers used in matching that resulted in the switch deciding to forward the packet to the controller. In addition to the packet contents, the packet-matching decision is influenced by *context* information. Formerly this consisted of the input port identifier. In V.1.2 this context information is expanded to include the input virtual port, the input physical port, and *metadata* that has been built up during packet-matching pipeline processing. Metadata semantics is not prescribed by the OpenFlow specification other than that the OXM metadata TLV may be initialized, modified, or tested at any stage of the pipeline processing. Since the OXM encoding we described contains TLV definitions for all the context fields as well as all matchable packet headers, the OXM format provides a convenient vehicle to communicate the packet-matching state when the switch decides to forward the packet to the controller. The switch may forward a packet to the controller because:

- There was no matching flow.
- An instruction was executed in the pipeline, prescribing that a matching packet be forwarded to the controller.
- The packet had an invalid TTL.

5.5.4 Multiple Controllers

In previous versions of OpenFlow, there was very limited support for backup controllers. In the event that communication with the current controller was lost, the switch entered either fail secure mode or fail standalone mode. The notion that the switch would attempt to contact previously configured backup controllers was encouraged by the specification, but it was not explicitly described. In V.1.2 the switch may be configured to maintain simultaneous connections to multiple controllers. The switch must ensure that it only sends messages to a controller pertaining to a command sent by that controller. In the event that a switch message pertains to multiple controllers, it is duplicated and a copy sent to each controller. A controller may assume one of three different roles relative to a switch:

- Equal
- Slave
- Master

These terms relate to the extent to which the controller has to change the switch configuration. The least powerful is obviously slave mode, wherein the controller may only request data from the switch, such as statistics, but may make no modifications. Both equal and master modes allow the controller the full ability to program the switch, but in the case of master mode the switch enforces that only one switch be in master mode and all others be in slave mode. The multiple-controllers feature is part of the way OpenFlow addresses the *high-availability* (HA) requirement.

5.5.5 Example: Bridging VLANs Through SP Networks

V.1.2 is the first release to provide convenient support for bridging customer VLANs through service provider (SP) networks. This feature is known by a number of different names, including *provider bridging*, *Q-in-Q*, and *stacked VLANs*. These all refer to the tunneling of an edge VLAN through a provider-based VLAN. There are a number of variants of this basic technology, but they are most often based on the nesting of multiple IEEE 802.1Q tags [10]. In this example we explain how this can be achieved by stacking one VLAN tag inside another, which involves the PUSHing and POPping of VLAN tags that was introduced in Section 5.4.3. We deferred this example until our discussion of V.1.2, since the V.1.1 packet-matching logic to support this process was very complex and not easily extended. With the extensible match support in V.1.2, the logic required to provide VLAN stacking in the OpenFlow pipeline is much more straightforward.

This feature is implemented in OpenFlow V.1.2 by encoding a flow entry to match a VLAN tag from a particular customer's layer two network. In our example, this match occurs at an OpenFlow switch serving as a service provider access switch. This access switch has been programmed to match VLAN 10 from site A of customer X and to tunnel this across the service provider core network to the same VLAN 10 at site B of this same customer X. The OpenFlow pipeline will use the extensible match support to identify frames from this VLAN. The associated action will PUSH service provider VLAN tag 40 onto the frame, and the frame will be forwarded out a port connected to the provider backbone network. When this doubly tagged frame emerges from the backbone, the OpenFlow-enabled service provider access switch matches VLAN tag 40 from that backbone interface, and the action associated with the matching flow entry POPs off the outer tag and the singly tagged frame is forwarded out a port connected to site B of customer A.

Note that although our example uses VLAN tags, similar OpenFlow processing can be used to tunnel customer MPLS connections through MPLS backbone connections. In that case, an outer MPLS label is PUSHed onto the packet and then POPped off upon exiting the backbone.

5.6 OpenFlow 1.3 Additions

OpenFlow V.1.3 [4] was released on April 13, 2012. This release was a major milestone. Many implementations are being based on V.1.3 in order to stabilize controllers around a single version. This is also true of ASICs. Since V.1.3 represents a major new leap in functionality and has not been followed quickly by another release, this provides an opportunity for ASIC designers to develop hardware support for many of the V.1.3 features, with the hope of a more stable market into which to sell their new chips. This ASIC opportunity notwithstanding, some post-V.1.0 features that are very challenging to implement in hardware. For example, the iterative matching actions we described in Section 5.4.1 are difficult to implement in hardware at line rates. It is likely that the real-life chips that support V.1.3 will have to limit the number of flow tables to a manageable number.[3] Such limitations are not imposed on software-only implementations of OpenFlow, and, indeed, such implementations will provide full V.1.3 support. Table 5.4 lists the major new features added in this release. The most prominent of these new features are discussed in the following sections.

[3]In Sections 12.9.3 and 13.2.7, we discuss the possibility of ASICs that would allow more general support of these features.

Table 5.4 Major New Features Added in OpenFlow 1.3

Feature Name	Description
Refactor capabilities negotiation	See Section 5.6.1
More flexible table-miss support	See Section 5.6.2
IPv6 extension header-handling support	-
Per-flow meters	See Section 5.6.3
Per-connection event filtering	See Section 5.6.4
Auxiliary connections	See Section 5.6.5
MPLS BoS matching	Bottom-of-stack bit (BoS) from the MPLS header may now be used as part of match criteria
Provider backbone bridging tagging	See Section 5.6.7
Rework tag order	Instruction execution order now determines tag order rather than it being statically specified
Tunnel-ID metadata	Allows for support of multiple tunnel encapsulations
Cookies in PACKET_IN	See Section 5.6.6
Duration for stats	The new *duration* field allows more accurate computation of packet and byte rate from the counters included in those statistics
On-demand flow counters	Disable/enable packet and byte counters on a per-flow basis

5.6.1 Refactor Capabilities Negotiation

There is a new MULTIPART_REQUEST/MULTIPART_REPLY message pair in V.1.3. This replaces the READ_STATE messaging in prior versions that used the STATS_REQUEST/STATS_REPLY to get this information. Rather than having this capability information embedded as though it were a table statistic, the new message request-reply pair is utilized and the information is conveyed using a standard *type-length-value* (TLV) format. Some of the capabilities that can be reported in this new manner include *next-table*, *table-miss flow entry*, and *experimenter*. This new MULTIPART_REQUEST/REPLY message pair subsumes the older STATS_REQUEST/REPLY pair and is now used for both reporting of statistics and capability information. The data formats for the capabilities are the TLV format, and this capability information has been removed from the table statistics structure.

5.6.2 More Flexible Table-Miss Support

We have explained that a table miss occurs when a packet does not match any flow entries in the current flow table in the pipeline. Formerly there were three configurable options for handling such a table miss. This included dropping the packet, forwarding it to the controller, or continuing packet matching at the next flow table. V.1.3 expands on this limited handling capability via the introduction of the *table-miss flow entry*. The controller programs a table-miss flow entry into a switch in much the same way it would program a normal flow entry. The table-miss flow entry is distinct in that it is by definition of lowest priority (zero) and all match fields are wildcards. The zero-priority characteristic guarantees that it is the last flow entry that can be matched in the table. The fact that all match fields are wildcards means that *any* packet being matched against the table miss will be a match. The upshot of these characteristics is that the table-miss entry serves as a kind of backstop for any packets that would otherwise have found no matches. The advantage of this approach is that the full semantics of the V.1.3 flow entry, including instructions and actions, may be applied to the case of a table miss. It should be obvious to the reader that the table-miss base cases of dropping the packet, forwarding it to the controller, or continuing at next flow table are easily implemented using the flow entry instructions and actions. Interestingly, though, by using the generic flow entry semantics, it is now conceivable to treat table misses in more sophisticated ways, including passing the packet that misses to a higher-numbered flow table for further processing. This capability adds more freedom to the matching "language" embodied by the flow tables, entries, and associated instructions and actions.

5.6.3 Per-Flow Meters

V.1.3 introduces a flexible meter framework. Meters are defined on a per-flow basis and reside in a *meter table*. Figure 5.16 shows the basic structure of a meter and how it is related to a specific flow. A given meter is identified by its *meter ID*. V.1.3 instructions may direct packets to a meter identified by its meter ID. The framework is designed to be extensible to support the definition of complex meters in future OpenFlow versions. Such future support may include color-aware meters that will provide DiffServ QoS capabilities. V.1.3 meters are only rate-limiting meters. As we see in Figure 5.16, there may be multiple *meter bands* attached to a given meter. Meter 3 in the example in the figure has three meter

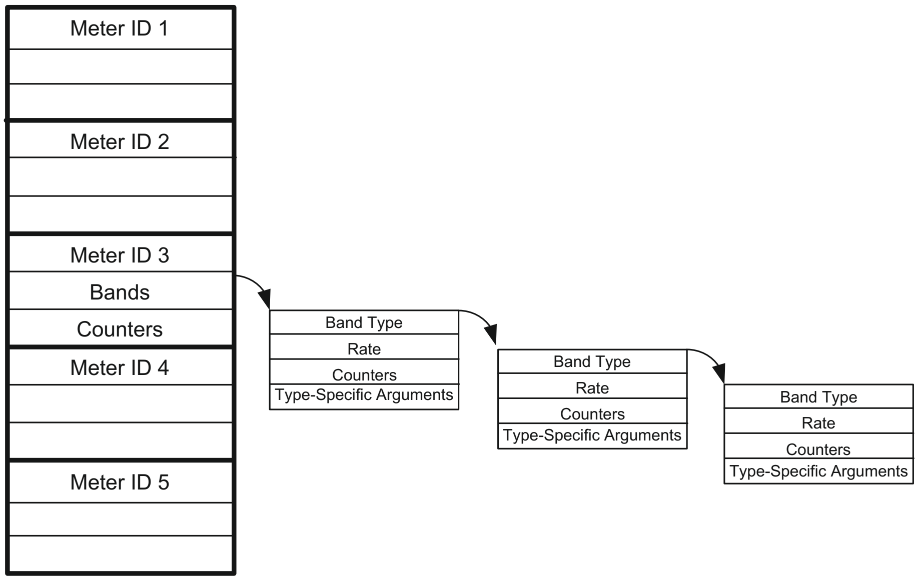


FIGURE 5.16

OpenFlow V.1.3 meter table.

bands. Each meter band has a configured bandwidth rate and a type. Note that the units of bandwidth are not specified by OpenFlow. The type determines the action to take when that meter band is processed. When a packet is processed by a meter, at most one band is used. This band is selected based on the highest bandwidth rate band that is lower than the current measured bandwidth for that flow. If the current measured rate is lower than all bands, no band is selected and no action is taken. If a band is selected, the action taken is that prescribed by the band's type field. There are no required types in V.1.3. The *optional* types described in V.1.3 consist of DROP and *DSCP remark*. DSCP remark indicates that the DiffServ drop precedence field of the *Differentiated Services Code Point* (DSCP) field should be decremented, increasing the likelihood that this packet will be dropped in the event of queue congestion. Well-known and simple rate-limiting algorithms such as the *leaky bucket* [6] may be implemented in a straightforward manner under this framework. This feature provides direct QoS control at the flow level to the OpenFlow controller by careful programming of the meters.

As shown in Figure 5.16, there are meter-level counters and meter-band-level counters. The meter counters are updated for all packets processed by the meter, and the per-meter-band counters are updated only when that particular band is selected. The dual level of counters is key in that presumably the majority of the packets processed by the meter suffer no enforcement, but their passage through the meter must be recorded in order to track the current measured rate. The reader should understand now that when a packet is processed by a band, it has exceeded a bandwidth threshold, which results in some kind of enforcement (e.g., dropping of the packet or marking it as *drop eligible* via DSCP remark).

5.6.4 Per Connection Event Filtering

We saw in V.1.2 the introduction of the notion of multiple controllers. Multiple controllers, able to take on different roles, achieve an improved level of fault tolerance and load balancing. The earlier discussion explained that the switches would only provide replies to the controller that initiated the request that elicits that reply. This, however, did not prescribe how to damp down the communication of asynchronous messages from the switch to its family of controllers. The idea that the different controllers have asymmetrical roles was not as effective as it could have been if all controllers must receive the same kind and quantity of asynchronous notifications from the switches. For example, a slave controller may not want to receive all types of asynchronous notifications from the switch. V.1.3 introduces a SET_ASYNC message that allows the controller to specify the sorts of async messages it is willing to receive from a switch. It additionally allows the controller to filter out certain reason codes that it does not want to receive. A controller may use two different filters: one for the master/equal role and another for the slave role. Note that this filter capability exists in addition to the ability to enable or disable asynchronous messages on a per-flow basis. This new capability is controller-oriented rather than flow-oriented.

5.6.5 Auxiliary Connections

We have already seen that earlier releases of OpenFlow allowed some parallelism in the switch-controller channel via the use of multiple controllers. In this way, mutiple parallel communication *channels* existed from a single switch to mutiple controllers. V.1.3 introduces an additional layer of parallelism by allowing multiple *connections* per communications channel. That is, between a single controller and switch, multiple connections may exist. We depict these multiple connections between the switch and the MASTER controller in Figure 5.17. The figure shows that there are also multiple channels

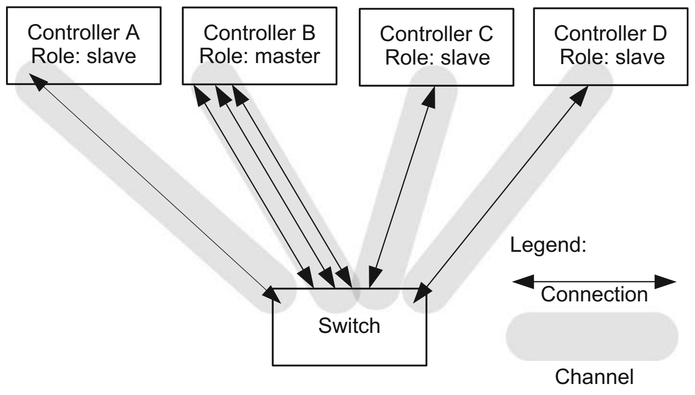


FIGURE 5.17

Multiple connections and multiple channels from a switch.

connecting the switch to different controllers. The advantage provided by the additional connections on a channel lies in achieving greater overall throughput between the switch and the controller. Because of the flow-control characteristics of a TCP connection, it is possible for the connection to be forced to quiesce (due to TCP window closing or packet loss) when there is actually bandwidth available on the physical path(s) between the switch and controller. Allowing multiple parallel connections allows the switch to take advantage of that. The first connection in the channel is specified to be a TCP connection. The specification indicates that other, unreliable connection types, such as UDP, may be used for the secondary connections, but the specification is fairly clear that the problems of potential data loss and sequencing errors that can arise from the use of such alternative transport layers is beyond the scope of the specification.

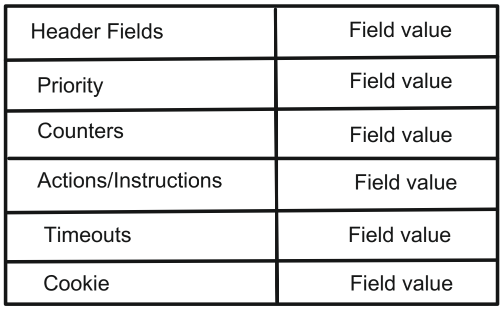
If bandwidth between the controller and switch becomes a constraint, it is most likely due to a preponderance of data packets. Some loss of data packets can be tolerated. Thus, the primary intended use of the auxiliary connections is to transmit and receive data packets between the switch and the controller. This presumes that any control messages would be sent over the reliable, primary connection.

One example of the utility of auxiliary connections is that when a switch has many PACKET_IN messages to send to the controller, this can create a bottleneck due to congestion. The delay due to this bottleneck could prevent important OpenFlow control messages, such as a BARRIER_REPLY message, from reaching the controller. Sending PACKET_IN data messages on the UDP auxiliary connection will obviate this situation. Another possible extension to this idea is that the OpenFlow pipeline could send UDP-based PACKET_IN messages directly from the ASIC to the controller without burdening the control code on the switch CPU.

5.6.6 Cookies in PACKET_IN

The straightforward handling of a PACKET_IN message by the controller entails performing a complete packet-matching function to determine the existing flow to which this packet relates, if any. As the size of commercial deployments of OpenFlow grows, performance considerations have become increasingly important. Accordingly, the evolving specification includes some features that are merely designed to increase performance in high-bandwidth situations. Multiple connections per channel, discussed above, is such an example. In the case of PACKET_IN messages, it is somewhat wasteful to require the controller to perform a complete packet match for every PACKET_IN message, considering that this lookup has already just occurred in the switch and, indeed, is the very reason the PACKET_IN message is being sent to the controller (that is, either a specific instruction to forward this packet to the controller was encountered or a table miss resulted in the packet being handed off to the controller). This is particularly true if this is likely to happen over and over for the same flow. To render this situation more efficient, V.1.3 allows the switch to pass a *cookie* with the PACKET_IN message. This cookie allows the switch to cache the flow entry pointed to by this cookie and circumvent the full packet-matching logic. Such a cookie would not provide any efficiency gain the first time it is sent by the switch for this flow, but once the controller caches the cookie and pointer to the related flow entry, considerable performance boost can be achieved. The actual savings accrued would be measured by comparing the computational cost of full packet-header matching versus performing a hash on the cookie. As of this writing, we are not aware of any study that has quantified the potential performance gain.

The switch maintains the cookie in a new field in the flow entry. Indeed, the flow entries in V.1.3 have expanded considerably compared to the basic flow entry we presented in Figure 5.6. The V.1.3.0 flow entry is depicted in Figure 5.18. The header, counter, and actions fields were present in Figure 5.6 and were covered earlier in Section 5.3.2. The priority field determines where this particular flow entry is placed in the table. A higher priority places the entry lower in the table such that it will be matched before a lower-priority entry. The timeouts field can be used to maintain two timeout clocks for this flow. Both an idle timer and a hard timeout may be set. If the controller wants the flow to be removed after a specified amount of time without matching any packets, then the idle timer is used. If the controller wants the flow to exist only for a certain amount of time, regardless of the amount of traffic, the hard timeout is used and the switch will delete this flow automatically when the timeout expires. We provide an example of such a timeout condition in Section 5.3.6. We covered the application of the cookie field earlier in this section.



Header Fields	Field value
Priority	Field value
Counters	Field value
Actions/Instructions	Field value
Timeouts	Field value
Cookie	Field value

FIGURE 5.18

V.1.3 flow entry.

5.6.7 Provider Backbone Bridging Tagging

Earlier versions of OpenFlow supported both VLAN and MPLS tagging. Another WAN/LAN technology known as *provider backbone bridging* (PBB) is also based on tags similar to VLAN and MPLS. PBB allows user LANs to be layer two bridged across provider domains, allowing complete domain separation between the user layer two domain and the provider domain via the use of *MAC-in-MAC* encapsulation. For the reader interested in learning more about PBB, we provide a reference to the related standard in [8]. This is a useful technology that's already supported by high-end commercial switches; support for this in OpenFlow falls out easily as an extension to the support already provided for VLAN tags and MPLS shim headers.

5.6.8 Example: Enforcing QoS via Meter Bands

Figure 5.19 depicts the new V.1.3 meter table as well as the group table that first appeared in V.1.1. We see in the figure that the incoming packet is matched against the second flow entry in the first flow

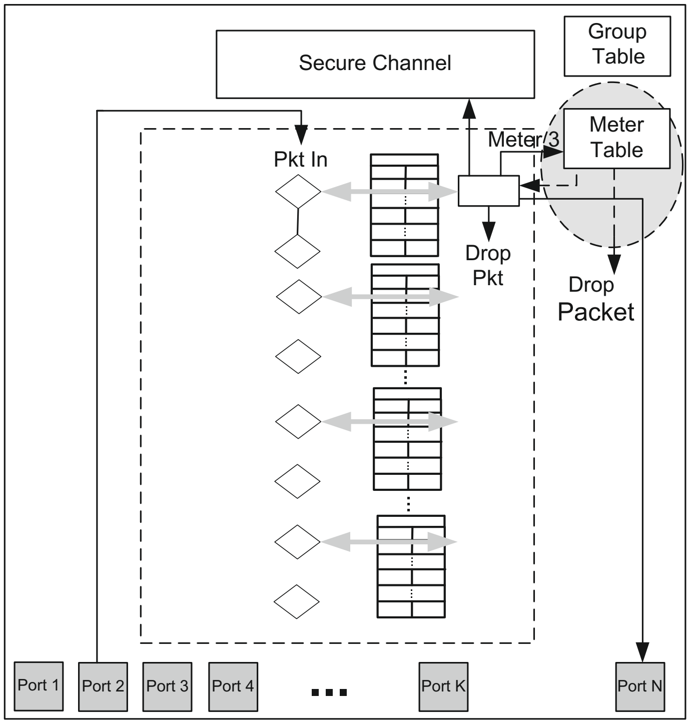


FIGURE 5.19

Use of meters for QoS control in V.1.3.

table and that the associated instructions direct the packet to meter 3. The two dashed lines emanating from the meter table show that based on the current measured bandwidth of the port selected as the output port (port N in this case), the packet may be dropped as exceeding the bandwidth limits, or it may be forwarded to port N if the meter does not indicate that bandwidth enforcement is necessary. As we explained earlier, if the packet passes the meter control without being dropped or marked as dropped-eligible, it is *not* automatically forwarded to the output port but may undergo any further processing indicated by the instructions in its packet-processing pipeline. This could entail being matched against a higher-numbered flow table with its own instructions and related actions.

5.7 OpenFlow Limitations

As OpenFlow remains in a state of rapid evolution, it is difficult to pin down precise limitations, as these may be addressed in subsequent releases. One limitation is that the currently defined match fields are limited to the packet header. Thus, *deep packet inspection* (DPI), whereby fields in the packet's payload may be used to distinguish flows, is not supported in a standard OpenFlow. Nonetheless, the EXPERIMENTER modes that are permitted within OpenFlow do open the way for such application layer flow definition in the future. Second, some OpenFlow abstractions may be too complex to implement directly in today's silicon. This is unlikely to remain an insurmountable obstacle for long, however, since the tremendous momentum behind SDN is likely to give birth to switching chips that are designed explicitly to implement even OpenFlow's most complicated features.

If we cast a wider net and consider limitations of all of Open SDN, there are a number of other areas to consider. We discuss these in Section 6.1.

5.8 Conclusion

In this chapter we attempted to provide the reader with a high-level understanding of the general OpenFlow framework. This discussion covered the protocol that an OpenFlow controller uses to configure and control an OpenFlow switch. We also presented the switch-based OpenFlow abstractions that must be implemented on a physical switch as an interface to the actual hardware tables and forwarding engine of the switch in order for it to behave as an OpenFlow switch. For the reader interested in a deeper understanding of these issues, whether out of sheer academic interest or a need to implement one of the versions of OpenFlow, there is no substitute for reading the specifications cited in [1–4]. These specifications are, however, very detailed and are not especially easy for the OpenFlow novice to follow. We believe that this chapter serves as an excellent starting point and guide for that reader who needs to delve into the lengthy specifications that comprise the OpenFlow releases covered here. In particular, these specifications contain a plethora of structure and constant names. To aid in reading the specifications, we draw your attention to Table 5.5 in this chapter as a quick reference and an aid to understanding to what part of the protocol a specific reference pertains.

Table 5.5 OpenFlow Protocol Constant Classes

Prefix	Description
OFPT	OpenFlow message type
OFPPC	Port configuration flags
OFPPS	Port state
OFPP	Port numbers
OFPPF	Port features
OFPQT	Queue properties
OFPMT	Match type
OFPXMC	OXM class identifiers
OFPXMT	OXM flow match field types for basic class
OFPIT	Instruction types
OFPAT	Action types
OFPC	Datapath capabilities and configuration
OFPTT	Flow table numbering
OFPFC	Flow modification command type
OFPFF	Flow modification flags
OFPGC	Group modification command type
OFPGT	Group type identifier
OFPM	Meter numbering
OFPMC	Meter commands
OFPMF	Meter configuration flags
OFPMBT	Meter band type
OFPMP	Multipart message types
OFPTFPT	Flow table feature property type
OFPGFC	Group capabilities flags

References

[1] OpenFlow switch specification, Version 1.0.0 (wire protocol 0x01). Open Networking Foundation; December 31, 2009. Retrieved from <www.opennetworking.org/sdn-resources/onf-specifications>.
[2] OpenFlow switch specification, Version 1.1.0 (wire protocol 0x02). Open Networking Foundation; February 28, 2011. Retrieved from <www.opennetworking.org/sdn-resources/onf-specifications>.
[3] OpenFlow switch specification, Version 1.2.0 (wire protocol 0x03). Open Networking Foundation; December 5, 2011. Retrieved from <www.opennetworking.org/sdn-resources/onf-specifications>.
[4] OpenFlow switch specification, Version 1.3.0 (wire protocol 0x04). Open Networking Foundation; June 25, 2012. Retrieved from <www.opennetworking.org/sdn-resources/onf-specifications>.
[5] Davie B, Doolan P, Rekhter Y. Switching in IP networks. San Francisco,CA, USA: Morgan Kaufmann; 1998.

[6] Shenker S, Partridge C, Guerin R. Specification of guaranteed quality of service, RFC 2212. Internet Engineering Task Force; 1997.
[7] IEEE standard for local and metropolitan area networks: link aggregation, IEEE 802.1AX. USA: IEEE Computer Society; November 3, 2008.
[8] IEEE Draft standard for local and metropolitan area networks–virtual bridged local area networks–amendment 6: provider backbone bridges, IEEE P802.1ah/D4.2. New York, NY, USA: IEEE; March 2008.
[9] IEEE standard for local and metropolitan area networks: media access control (MAC) bridges and virtual bridged local area networks, amendment 21: edge virtual bridging, IEEE 802.1Qbg. IEEE; July 2012.
[10] IEEE standard for local and metropolitan area networks: media access control (MAC) bridges and virtual bridged local area networks, IEEE 802.1Q. New York, NY, USA: IEEE; August 2011.

CHAPTER 6

Alternative Definitions of SDN

There are likely to be networking professionals who dismiss Open SDN as much ado about nothing. To be sure, some of the criticism is a natural reaction against change. Whether you are a network equipment manufacturer or a customer or any part of the vast industry gluing those endpoints together, the expectation of disruptive change such as that associated with SDN is unsettling. There are indeed valid drawbacks to the Open SDN concept. In the first part of this chapter, we inventory the drawbacks most commonly cited about Open SDN. For each, we attempt to assess whether there are reasonable solutions to these drawbacks proposed by the Open SDN community.

In some cases, the disadvantages of Open SDN have been addressed by proposed alternative SDN solutions that do not exhibit these flaws. In Chapter 4 we introduced a number of alternative SDN technologies that propose SDN solutions that differ from Open SDN. In a field with so much revenue-generating opportunity, some vendors may have a tendency to jump on the SDN bandwagon and claim that their product is an SDN solution. It is therefore important that we have a way of determining whether or not we should consider a particular alternative a true SDN solution. Recall that in Section 4.1 we said that we characterize an SDN solution as possessing all of the five following traits: *plane separation*, *a simplified device*, *centralized control*, *network automation and virtualization*, and *openness*. As we discuss each of the alternative SDN technologies in the latter part of this chapter, we will evaluate it against these five criteria. Admittedly, this process remains a bit subjective, since our litmus test is not absolute. Some of the alternatives do exhibit some of our defined SDN traits, but not all. In addition, we evaluate whether or not each alternative suffers from the same drawbacks noted early in the chapter for Open SDN. Without providing any absolute judgment about winners or losers, our hope is that this SDN report card will help others make decisions about which, if any, SDN technology is appropriate for their needs.

6.1 Potential Drawbacks of Open SDN

Many have met the advent of SDN with enthusiasm, but it is not without critics. They point to a number of shortcomings related to both the technology and its implementation. Some of the most vocal critics are the network vendors themselves. This is not surprising, since they are the most threatened by such a radical change in the landscape of networking. Some critics are skeptics who do not believe that this new technology will be successful, for various reasons that we examine in this chapter.

6.1.1 Too Much Change, Too Quickly

In Chapter 3 we introduced the creation of the *Clean Slate* project at Stanford, where researchers were encouraged to consider what networking might be like if we started anew without the baggage of our legacy networks. Out of Clean Slate came the creation of OpenFlow and the basic concepts of Open SDN as defined in this book. Some network professionals feel that this approach is not practical, since real-world customers do not have the luxury of being able to discard old technology wholesale and start fresh.

The following reasons are often cited as to why such a comprehensive change would negatively impact network equipment customers:

- **The expense of new equipment.** A complete change to the way networking is performed is too expensive because it requires a *forklift* change to the network. Large quantities of existing network equipment must be discarded as new equipment is brought en masse into customer networking environments. *Counterargument: If this is a greenfield deployment, this point is moot because new equipment is being purchased anyway. If this is an existing environment with equipment being continually upgraded, it is possible to upgrade your network to SDN incrementally using a sound migration plan.*
- **Too risky.** A complete change to the networking process is too risky because the technology is new, is possibly untested, and is not backed by years of troubleshooting and debugging experience. *Counterargument: Continued use of legacy protocols also incurs risk due to the increased costs of administering them, the inability to easily automate the network, and the inability to scale virtual networks as needed. It is unreasonable to consider only the risks of migrating to SDN without also considering the risk of maintaining the status quo.*
- **Too revolutionary.** There are tens of thousands of network engineers and IT staff personnel who have had extensive training in the networking technology that exists today. Though it may be true that Open SDN has the potential to yield benefits in terms of lowered operational expenses, in the short term these expenses will be higher. This increased expense will in large part be due to the necessary training and education for network engineers and IT personnel. *Counterargument: The scale and growth of data centers has itself been revolutionary and it is not unreasonable to think that a solution to such a problem may also need to be revolutionary.*

The risk associated with too much change too quickly can be mitigated by deploying Open SDN in accordance with a carefully considered migration plan. The network can be converted to Open SDN in stages, targeting specific network areas for conversion and rolling out the changes incrementally. Figure 6.1 shows a simple example of such phasing. First, the SDN technology is tested in a lab environment. It is subsequently rolled out to various subnets or domains incrementally, based on timing, cost, and sensitivity to change. This phased approach can help mitigate the cost issue as well as the risk. It also provides time for the retraining of skilled networking personnel. Nevertheless, for those areas

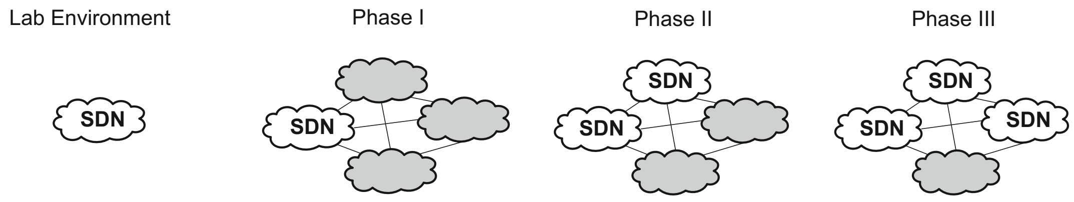


FIGURE 6.1

Phased deployment of SDN.

where Open SDN is newly deployed, the change will be radical. If this radical change brings about radical improvements in an environment previously careening out of control, this is a positive outcome. Indeed, sometimes radical change is the only escape from an out-of-control situation.

6.1.2 Single Point of Failure

One of the advantages of distributed control is that there is no single point of failure. In the current model, a distributed, resilient network is able to tolerate the loss of a single node in the infrastructure and reconfigure itself to work around the failed module. However, Open SDN is often depicted as a single controller responsible for overseeing the operation of the entire network. Since the control plane has been moved to the controller, while the SDN switch can forward packets matching currently defined flows, no changes to the flow tables are possible without that single controller. A network in this condition cannot adapt to any change. In these cases, the controller can indeed become a single point of failure.

Figure 6.2 shows a single switch failing in a traditional network. Prior to the failure, we see that two flows, $F1$ and $F2$, were routed via the switch that is about to fail. We see in the figure that upon detecting the failed node, the network automatically uses its distributed intelligence to reconfigure itself to overcome the single point of failure. The two flows are routed via the same alternate route.

Figure 6.3 shows a recovery from the failure of a single node in an SDN. In this case, it is the intelligence in the SDN controller that reconfigures the network to circumvent the failed node. One of the premises of SDN is that a central controller will be able to deterministically reconfigure the network in an optimal and efficient manner, based on global knowledge of the network. Because of other information about the overall network that the SDN controller has at its disposal, we see in the figure that it reroutes $F1$ and $F2$ over different paths. One scenario that would cause this rerouting would be that the two flows have specific QoS guarantees that cannot be met if they are both placed on the same alternate path that was chosen in Figure 6.2. Finding optimal paths for different flows is relatively straightforward within the Open SDN model, yet it is very challenging to implement in traditional networking with autonomous devices. We revisit this topic via a specific use case in Section 8.1.2.

If, however, the single point of failure is the controller itself, the network is vulnerable, as shown in Figure 6.4. As long as there is no change in the network that requires a flow table modification, the network can continue to operate without the controller. However, if there is any change to the topology,

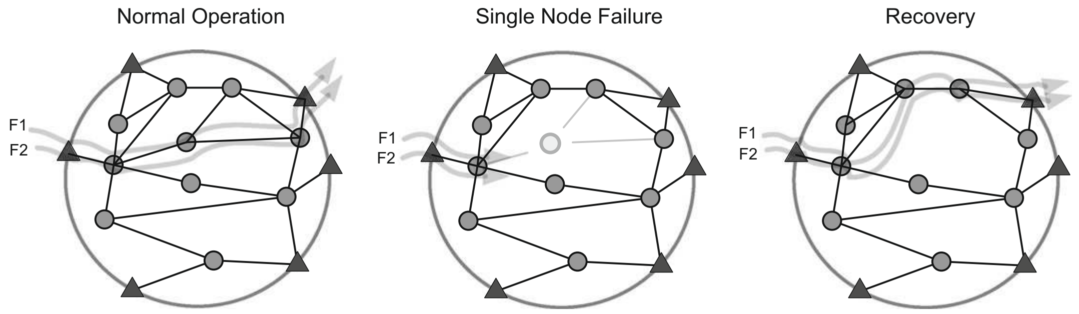


FIGURE 6.2

Traditional network failure recovery.

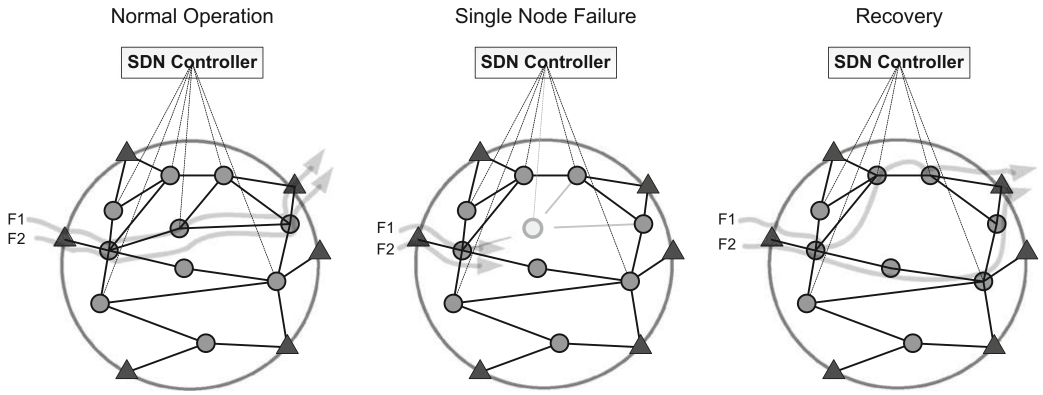


FIGURE 6.3

SDN network failure recovery.

the network is unable to adapt. The loss of the controller leaves the network in a state of reduced functionality, unable to adapt to failure of other components or even normal operational changes. So, failure to the centralized controller could potentially pose a risk to the entire network. The SDN controller is vulnerable to both hardware and software failures as well as to malicious attacks. Note that in addition to such negative stimuli such as failures or attacks, a sudden surge in flow entry modifications due to a sudden increase in network traffic could also cause a bottleneck at the controller, even if that sudden surge is attributable to an entirely positive circumstance, such as a flash mob!

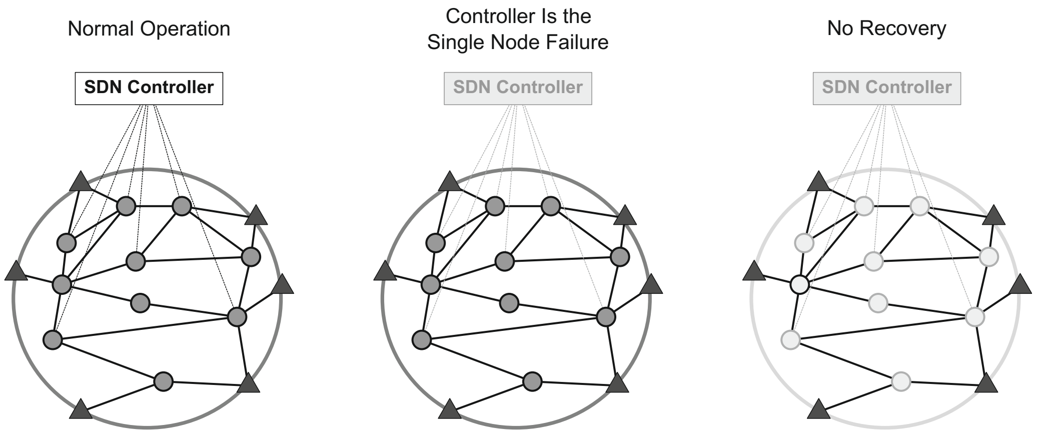


FIGURE 6.4

SDN controller as single point of failure.

The SDN controller runs on a physical compute node of some sort, probably a standard off-the-shelf server. These servers, like any other hardware components, are physical entities, frequently with moving parts; as such, they are subject to failure. Software is subject to failure due to poor design, bugs, becoming overloaded due to scale, and other typical software issues that can cause the failure of the controller. A single SDN controller is an attractive target for malicious individuals who want to attack an organization's networking infrastructure. *Denial-of-service* (DoS) attacks, worms, bots, and similar malware can wreak havoc on an SDN controller that has been compromised, just as they have wreaked havoc in legacy networks for years.

Critics of SDN point to these as evidence of the susceptibility of SDN to catastrophic failure. If indeed the SDN controller is allowed to become that single point of failure, this does represent a structural weakness. Clearly, SDN designers did not intend that operational SDN networks be vulnerable to the failure of a single controller. In fact, the Open SDN pioneers described the concept of a *logically centralized controller*. In Open SDN, one has the flexibility to decide how many control nodes (controllers) the network has. This number can range from a totally centralized situation with a single control node to one control node per device. The latter extreme may still theoretically fit the Open SDN paradigm, but it is not practical. The disparate control nodes will face an unnecessary challenge in trying to maintain a consistent global view of the network. The other extreme of the single controller is also not realistic. A reasonable level of controller redundancy should be used in any production network.

High-Availability Controller with Hardened Links

SDN controllers must use *high-availability* (HA) techniques and/or redundancy. SDN controllers are not the only systems in the broad computer industry that must be highly available. Wireless LAN controllers, critical servers, storage units, even networking devices themselves have for some time relied on HA and redundancy to ensure that there is no suspension of service should there be a failure of a hardware or software component. Figure 6.5 shows an example of a well-designed controller configuration with redundant controllers and redundant controller databases. Redundancy techniques range from $N + 1$

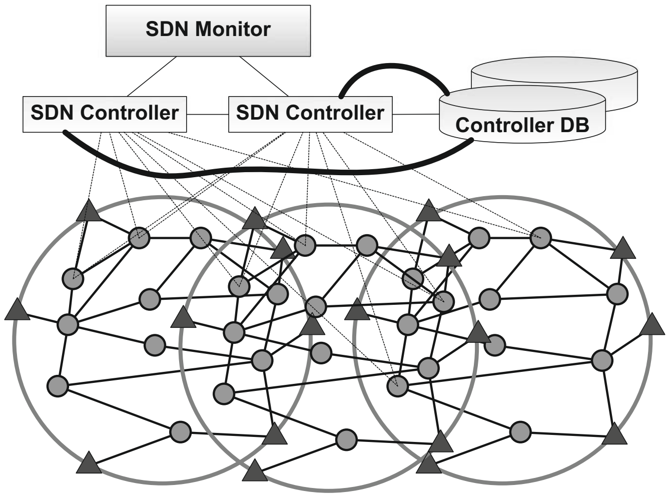


FIGURE 6.5

Controller high availability and monitoring.

approaches, where a single hot-standby stands ready to assume the load of any of the *N* active controllers, to the more brute-force approach of having a hot-standby available for each controller. A version of hot-standby for each controller is to use HA designs internal to the controller hardware such that all critical components are redundant within that server (e.g., mirrored disks, redundant power supplies). Redundancy is also reflected by allowing the devices themselves to switch to a backup controller, as we discussed in Section 5.5.4.

Detecting complete failure can be relatively simple compared to ascertaining the correct operation of a functional component. However, instrumentation exists to monitor and evaluate systems to make sure they are working as intended and to notify network administrators if they begin to deviate from the norm. Thus, SDN controllers must have monitoring and instrumentation in place to ensure correct operation. Figure 6.5 depicts a monitor that is constantly alert for changes in the behavior and responsiveness of the SDN controllers under its purview. When a problem arises, the monitor is able to take corrective action.

With respect to malicious attacks, our intuition leads us to the conclusion that by centralizing the control plane, the network now has a vulnerable central point of control that must be protected against attack or the entire network becomes vulnerable. This is in fact true, and the strongest available security measures must be used to defend against such attacks. In practice, both the controller and the links to it would be hardened to attack. For example, it is common practice to use out-of-band links or high-priority tunnels for the links between the devices and the controller. In any case, it is naïve to assume that the distributed control system such as that used in legacy networks is immune to attack. In reality, distributed systems can present a large attack surface to a malicious agent. If one device is hacked, the network can self-propagate the poison throughout the entire system. In this regard, *a single SDN controller presents a smaller attack surface.* If we consider the case of a large data center, there are thousands or tens of thousands of control nodes available to attack. This is because each network device has a locally resident control plane. In an equivalent-sized SDN, there might be 10 to 20 SDN controllers. Assuming that the compromising of any control node can bring down an entire network, that means that the SDN presents two orders of magnitude fewer control nodes susceptible to attack, compared to its traditional equivalent. Through hardening both the controllers and the secure channels to attack, network designers can focus extra protection around these few islands of intelligence in order to keep malicious attacks at bay.

In summary, the introduction of a single point of failure is indeed a vulnerability. With the right architecture and safeguards in place, an Open SDN environment may actually be a safer and more secure network than the earlier distributed model.

6.1.3 Performance and Scale

As the saying goes, “Many hands make light work.” Applied to networking, this saying would imply that the more the networking intelligence is spread across multiple devices, the easier it is to handle the overall load. In most cases this is certainly true. Having a single entity responsible for monitoring as well as switching and routing decisions for the entire network can create a processing bottleneck. The massive amount of information pertaining to all end nodes in a large network such as a data center can indeed become an issue. In spite of the fact that the controller software is running on a high-speed server with large storage capabilities, there is a limit at which even the capacity of such a server becomes strained and performance could suffer. An example of this performance sensitivity is the potential for request congestion on the controller. The greater the quantity of network devices that are dependent on the SDN controller for decisions regarding packet processing, the greater the danger of overloading the input queues of the controller such that it is unable to process incoming requests in a timely manner, again causing delays and impacting network performance.

We have dedicated a considerable part of this book to promoting the practice of moving the control plane off the switch and onto a centralized controller. The larger the network under the management of the SDN controller, the higher the probability of network delays due to physical separation or network traffic volume. As network latency increases, it becomes more difficult for the SDN controller to receive requests and respond in a timely manner.

Issues related to performance and scalability of Open SDN are real and should not be dismissed. It is worth noting, however, that although there can be performance issues related to scale in a network design involving a centralized controller, the same is true when the network intelligence is distributed. One such issue is that those distributed entities will often need to coordinate and share their decisions with one another; as the network grows, more of these distributed entities need to coordinate and share information, which obviously impacts performance and path convergence times. Indeed, there is evidence that for Open SDN the convergence times in the face of topology changes should be *as good or better* than with traditional distributed routing protocols [1]. Additionally, these distributed environments have their own frailties regarding size and scale, as we have pointed out in earlier chapters, in areas such as MAC address table size and VLAN exhaustion. Also, in large data center networks, with VMs and virtual networks constantly being ignited and extinguished, path convergence may never occur with traditional networks.

A further issue with autonomous, distributed architectures is that in real-life networks the protocols that comprise the control plane are implemented by different groups of people at different times, usually working at multiple NEMs. Although we have standards for these protocols, the actual implementations on the devices are different from one version to another and from one manufacturer to another. In addition to the vagaries that result from different groups of humans trying to do anything, even *identical* implementations will behave differently from one device to another due to different CPU speeds, differing buffering constraints, and the like. These differences can create instability within the group of devices, and there is no way to scale the computation needed to manage the topology effectively. The centralized design has fewer differences in the implementation of the control plane because there are fewer implementations running it, and it *can* be scaled, as we have explained, using traditional compute scaling approaches as well as by simply buying a bigger server. The upgrade process to higher-performing control planes is easier to address in this centralized environment. Conversely, in the embedded, distributed world of traditional networking, we are stuck with low-powered CPUs and a nonupgradable memory footprint that came with the devices when they were purchased.

There are ways to improve Open SDN's performance and scalability through appropriate controller design. In particular, it is important to employ network design where more than one controller is used to distribute the load. There are two primary ways of doing this. The first is to deploy multiple controllers in a *cluster*. As network size grows, there will be a need for multiple coordinated controllers. Note that the logistics required to coordinate a number of these controllers is nowhere near the complexity of logistics and coordination required with thousands of network devices, so the problem is easier to solve. In this type of deployment, the controllers are able to share the load of the large set of network devices

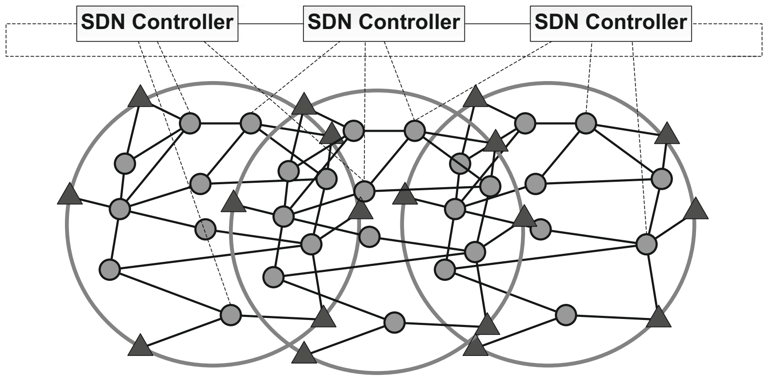


FIGURE 6.6

Controller cluster.

under their control. Figure 6.6 shows three controllers spreading the load of the management of a large number of devices. Though the details of how to implement clusters of controllers are currently outside the scope of the OpenFlow specification, the specification does state that a multiple-controller scenario is likely, and some aspects of the protocol have been extended to describe messaging to multiple controllers. This notion of clusters of controllers has been in widespread use for some time with wireless controllers. Some NEMs, notably Cisco in [3] and HP in [4], have published guidelines for the way multiple wireless controllers for the same network should interact.

A second means of utilizing multiple controllers is to organize the controllers into a *hierarchy of controllers*. As network size grows, it may be necessary to create such a hierarchy. This model allows for offloading some of the burden from the *leaf* controllers to be handled by the *root* controller. The concept of controller hierarchies for SDNs is explored in detail in [2]. Figure 6.7 shows a two-level hierarchy of controllers, with the root controller managing the coordination between the leaf controllers' peers below. In this type of deployment there can be a separation of responsibilities as well; the highest-level controller can have global responsibility for certain aspects of the network, while lower-level responsibilities that require more geographical proximity may be handled by the controllers at the lower tier. This geographical proximity of the leaf controllers can ameliorate the problems of increased latency that come with increased network size.

To summarize, performance issues can arise in large Open SDN networks, just as they can in non-SDNs. The solution lies in designing and deploying the SDN controllers and the network in such a way as to obviate those potential issues.

6.1.4 Deep Packet Inspection

A number of network applications may have requirements that exceed the capabilities of SDN or, more specifically of OpenFlow as it is defined today. In particular, there are applications that need to see more of the incoming packet in order to make matches and take actions.

One type of application that requires *deep packet inspection* (DPI) is a firewall. Simple firewalls may make decisions based on standard fields available in OpenFlow, such as destination IP address or

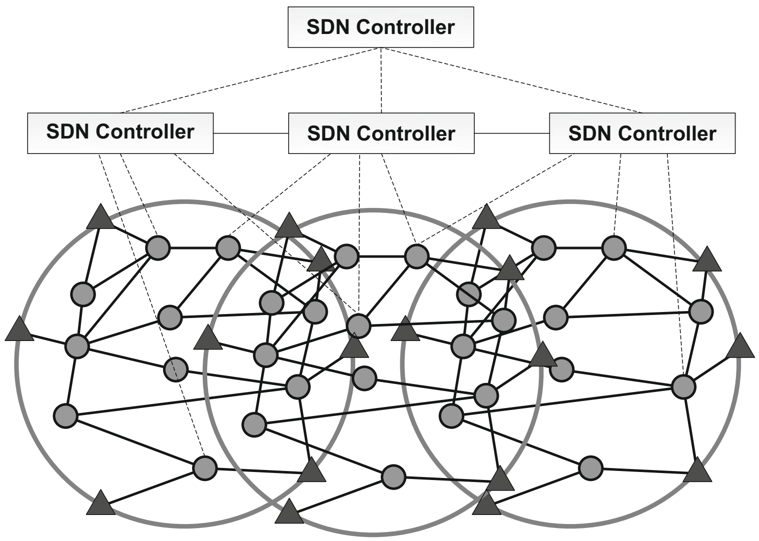


FIGURE 6.7

Controller hierarchy.

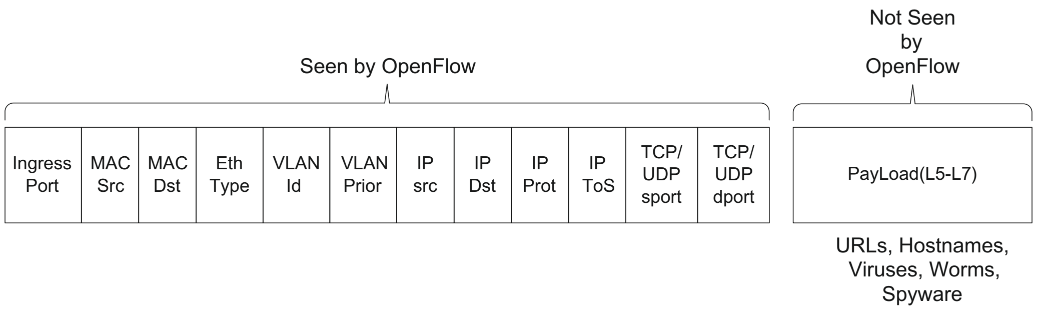


FIGURE 6.8

Deep packet inspection.

TCP/UDP port. More advanced firewalls may need to examine and act on fields that are not available to OpenFlow matches and actions. If it is impossible to set flows that match on appropriate fields, the device will be unable to perform local firewall enforcement.

Another application of this sort is a load balancer. Simple load balancers, like firewalls, make decisions based on standard fields available in OpenFlow, such as source or destination IP address or source or destination TCP/UDP port. This is sufficient in many cases, but more advanced load balancers may want to forward traffic based on fields not available in OpenFlow today, such as the specific destination URL. Figure 6.8 shows the fields that OpenFlow 1.0 devices are able to match. As the figure illustrates, OpenFlow is not able to match against fields in the payload. The payload data includes many important items such as URLs. It is possible that even simple load-balancing decisions might be made on criteria in the payload. In addition, detecting viruses, worms, and other spyware may require examining and processing such data. Since OpenFlow devices are not able to match data in the packet payload, they cannot make enforcement decisions locally.

Two distinct challenges are brought to light in this scenario. The first is to detect that a packet might need special treatment. If the criteria for that decision lie within the packet payload, an OpenFlow device cannot perform that test. Second, even if the device can determine that a packet needs to be further scrutinized, the OpenFlow device will not have the processing horsepower to perform the packet analysis that many *intrusion detection systems* (IDS) and *intrusion prevention systems* (IPS) perform. In this case, there is a solution that dovetails well with OpenFlow capabilities. When the OpenFlow pipeline determines that a packet does not need deeper inspection, perhaps because of packet size or TCP/UDP port number, it can forward the packet around the *bump in the wire* that the in-band IDS/IPS normally constitutes. This has the advantage of allowing the vast majority of the packets transiting the network to completely bypass the IDS/IPS, helping to obviate the bottlenecks that these systems frequently become.

Rather than directing selected packets around an in-line appliance, OpenFlow can sometimes be used to shunt them to a deep-packet inspection device and allow normal packets to proceed unfettered. Such a device could actually reside inside the same switch in a separate network processor card. This is the idea behind the *split SDN data plane architecture* presented in [14].

In any event, as of this writing, deep packet inspection is still absent from the latest version of the OpenFlow standard. Since this is a recognized limitation of OpenFlow, it is possible that a future version of the standard will provide support for matching on fields in the packet payload.

Limitations on Matching Resulting from Security Mechanisms

Note that the crux of the problem of deep packet inspection is the inability to make forwarding decisions based on fields that are not normally visible to the traditional packet switch. What we have presented thus far refers to fields that are in the packet payload, but the same problem can exist for certain packet header fields in the event that those fields are encrypted by some security mechanism. An example is encrypting the entire payload of a layer two frame. Obviously, this obscures the layer three header fields from inspection. This is not a problem for the traditional layer two switch, which will make its forwarding decisions solely on the basis of the layer two header fields, which are still in the clear. An Open SDN switch, on the other hand, would lose its ability to make fine-grained flow-based forwarding decisions using as criteria all of the 12-tuple of packet header fields normally available. In this sense, such encryption renders normal header fields unavailable to the OpenFlow pipeline, much like fields deep within the packet payload.

6.1.5 Stateful Flow Awareness

An OpenFlow device examines every incoming packet independently, without consideration of earlier-arriving packets that may have affected the state of either the device or the transaction taking place between the two communicating nodes. We describe this characteristic as a lack of *stateful flow awareness* or *statefulness*. An OpenFlow device attempting to perform a function that requires statefulness will not be adequate for the task. Examples of such functions include stateful firewalls or more sophisticated IPS/IDS solutions. As we noted in Section 6.1.4, in some cases OpenFlow can help the network scale by shunting packets that are candidates for stateful tracking to a device capable of such processing, diverting much of the network flows away from those processing-intensive appliances.

Most network applications use standard and well-known TCP or UDP ports. For example, HTTP traffic uses TCP port 80, *Domain Name System* (DNS) uses TCP port 53, *Dynamic Host Configuration Protocol* (DHCP) uses TCP and/or UDP ports 67 and 68, and so on. These are easy to track and to establish appropriate OpenFlow rules. However, certain protocols—for example, FTP and *Voice over IP* (VoIP)—start on a standard and fixed port (21 for FTP, 5060 for the VoIP *Session Initiation Protocol* (SIP) protocol) for the initial phase of the conversation but then set up an entirely separate connection using dynamically allocated ports for the bulk of their data transfer. Without stateful awareness, it is impossible for an OpenFlow device to have flow rules that match these dynamic and variable ports. Figure 6.9 shows an example of stateful awareness in which an FTP request is made to the FTP service on TCP port 21, which creates a new FTP instance that communicates with the FTP client on an entirely new TCP port chosen from the user space. An ability to track this state of the FTP request and subsequent user port would be helpful for some applications that want to deal with FTP traffic in a particular manner.

Some solutions, such as load balancers, require the ability to follow transactions between two end nodes to apply policy for the duration of the transaction. Without stateful awareness of the transaction, an OpenFlow device is unable to independently make decisions based on the state of ongoing transactions and to take action based on a change in that state. Figure 6.10 shows a sequence of HTTP requests that together constitute an entire transaction. The transaction is identified by the session number. Stateful awareness would allow the transaction to be dealt with as an integral unit.

It should be noted that contemporary Internet switches are generally stateless. Higher-function specialized appliances such as load balancers and firewalls do implement stateful intelligence, however. Since Open SDN is often touted as a technology that could replace purpose-built network appliances such as these, the lack of stateful awareness is a genuine drawback. Some vendors of these appliances will justifiably advertise their capability for deep packet inspection as well as their ability to track state and provide features based on these two capabilities.

As with deep packet inspection, the lack of stateful awareness can be a genuine limitation for certain applications. We are not aware of any effort by the ONF to specify such stateful behavior for flow

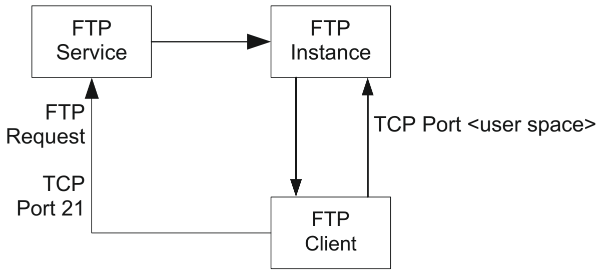


FIGURE 6.9

Stateful awareness for special protocols.

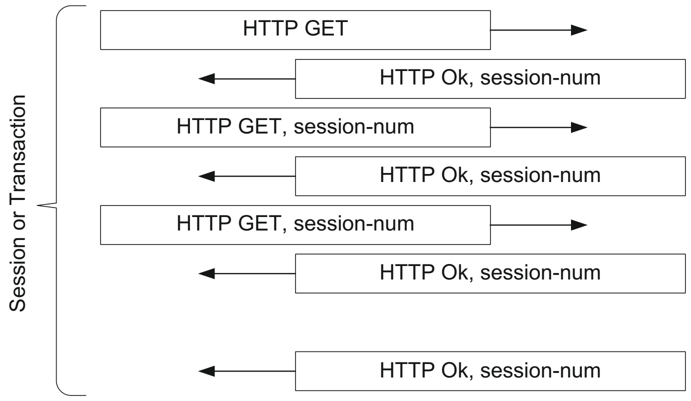


FIGURE 6.10

Stateful awareness of application-level transactions.

forwarding in OpenFlow. In those instances in which such stateful flow treatment is required, the user will likely have to continue to rely on specialized hardware or software rather than Open SDN.

6.1.6 Summary

In this section we reviewed a number of often-cited shortcomings of Open SDN. In some cases, there are documented Open SDN strategies for dealing with these issues. In other cases, workarounds outside the scope of Open SDN need to be used to address the specific application. In the balance of this chapter we examine alternative SDN technologies in greater depth than we have previously. For each, we determine whether the Open SDN drawbacks discussed in this section apply to that alternative SDN technology as well. This is important, since these alternatives are often promoted specifically as a response to the perceived limitations of Open SDN. We also assess how well each alternative meets our criteria to be considered an SDN technology.

6.2 SDN via APIs

One of the alternative SDN approaches that we introduced in Chapter 4 was *SDN via existing APIs*.[1] Superficially, one might be tempted to dismiss SDN via APIs as incumbent network vendors' attempt to protect their turf against encroachment by radical new technologies. But we would do this solution a disservice if we did not consider its positive elements. One such positive element is the movement toward controller-driven networks.

This brings us to the first of two artificial constraints we use in our definition of SDN via APIs. First, we acknowledge that there are some proponents of controller-less SDN via APIs. Juniper's *JunOS SDK* [16] provides a rich set of network device programmability and was initially released before SDN

[1]For the sake of brevity, we will henceforth use the term *SDN via APIs*.

became known to the industry. It is not controller-based, since the JunOS SDK applications actually reside in the devices themselves. Through the SDK APIs, these applications may wield control over a wide variety of device functions, including packet processing in the data plane. The details of the individual platforms' data plane hardware are abstracted so that a network programmer can affect packet processing without worrying about the vagaries of one hardware implementation versus another. Nonetheless, we consider such controller-less approaches to be outliers, and so we exclude them from consideration in this category. Looking back at our introduction to SDN via APIs in Chapter 4, Figure 4.7 portrayed a controller accessing APIs residing in the devices. This model will guide our definition of the SDN-via-APIs approach.

The second constraint is on the very definition of API. We want to delineate the OpenSDN approach from SDN via APIs. Nonetheless, OpenFlow itself is clearly the preeminent southbound API to the networking devices in Open SDN. Since we are trying to bring clarity to a confusing situation, when we say SDN via APIs, we refer to APIs that program or configure a control plane that is active in the device. Our introduction to SDN via APIs in Chapter 4 used the term *SDN via existing APIs*. Although that term suited the context of that chapter, the word *existing* is not sufficiently precise. Legacy APIs all programmed the control plane that was resident on the device. If a brand-spanking-new API continues to program a control plane residing in the device, it will still fall into this category. We contrast this with OpenFlow. As we explained in Chapter 5, OpenFlow is used to directly control the data plane. This distinction will be very important as we review both legacy and new APIs that are part of the SDN-via-APIs paradigm.

We have spent significant time in this book discussing the history behind the distributed-intelligence model of networking and the stagnation that has occurred due to the resulting closed system. SDN via APIs actually does open up the environment to a certain degree, since there are opportunities for software developed for the SDN controller to use those APIs in new and better ways and, hence, to advance networking technology.

This movement toward networking control residing in software in a controller is definitely a step forward in the evolution of networking. Another major step forward is the improvement in APIs on the devices themselves, which we discuss in the next section.

6.2.1 Legacy APIs in Network Devices

Currently, networking devices have APIs that allow for a certain amount of configuration. There are a number of network device APIs in current use:

- The *Simple Network Management Protocol* (SNMP) was first standardized in 1988 and has been the standard for management interactions with networking devices since that time. The general mechanism involves a simple protocol allowing the controlling function to GET and SET objects on a device that have been defined in a *management information base* (MIB). The functionality offered by SNMP can be extended by the definition of new and sometimes proprietary MIBs, allowing access to more information and attributes on a device. SNMP does provide both GET and SET primitives, but it is primarily used for monitoring using GET rather than for configuration, which uses SET.
- The *command-line interface* (CLI) is the most fundamental way of accessing a device and is designed with a human end user in mind. Often the networking device functionality placed into a device is accessible only via CLI (not SNMP). Consequently, management software attempting to access this

data must emulate a user logging into the CLI, executing CLI commands, and attempting to read the results. Even when the CLIs are enhanced to support scripting, this merely automates an interface designed for static configuration changes.

- The *Transaction Language 1* (TL1), developed for telecommunications equipment in the 1980s, is similar in some respects to SNMP in that its intent is to provide a language for communication between machines.
- The *Network Configuration Protocol* (NETCONF) was developed in the last decade to be a successor to protocols such as SNMP, since SNMP was mainly being used to monitor networks, not to configure them. The configuration payload utilizes an *Extensible Markup Language* (XML) format for passing data and commands to and from the networking devices.
- The *TR-069 CPE WAN Management Protocol* [15] is intended to control the communication between *customer-premises equipment* (CPE) and an *autoconfiguration server* (ACS). In our context, the CPE is the network device. The ACS is a novel contribution of TR-069. The ACS provides secure autoconfiguration of the network device and incorporates other device management functions into a unified framework.

These APIs have been used for years, primarily by network management applications. We consider these to be legacy APIs, but this should not imply that they are only for configuring old features. They have been extended to provide support for cutting-edge features such as Cisco's PBR [5]. However, these extended APIs have not scaled well into the current world of networking, especially in data centers. When there is a fully distributed control plane on hundreds or thousands of devices, this means that the configuration of the control plane on each device is very complex. Thus, programming a network of these distributed control planes requires exposing a very complex API to the network programmers, making their task much harder.

6.2.2 Appropriate APIs for SDN via APIs

SDN-appropriate APIs differ first in that they indirectly affect the behavior of the data plane in the networking device by setting parameters ranging from basic operational parameters to information about specific flows. We emphasize *indirectly* here because, by our definition of this alternative, these APIs directly interact with the control plane on the device, as was the tradition with all legacy APIs. The level of flow detail that can be specified by such APIs should be similar to the level of detail provided by OpenFlow. Although legacy APIs can be retrofitted to configure specific flows as in [5], that solution is really just extending an old interface on a legacy device to enable a new feature. SDN-appropriate APIs are more than this. First, they are dynamic, flexible, and designed to have an immediate effect on the behavior of the networking device. This is in contrast to legacy APIs, which, in many instances, are not put into effect until some process on the device has been restarted.

Second, SDN-appropriate APIs are designed to be used by a centralized controller, not just to enable a feature in a single network device in isolation but rather as part of programming a network of devices. A prime example of an SDN-appropriate API is the *Representational State Transfer* (REST) API.

6.2.3 REST: A Different API Protocol

The protocols and mechanisms for accessing devices to perform monitoring and configuration tasks, such as SNMP, CLI, TL1, NETCONF, and TR-069, have been used to a limited degree with SDN.

The new technology of choice for SDN-appropriate APIs is REST, which uses HTTP or HTTPS. APIs based on this technology are called *RESTful* interfaces.

REST technology was introduced a number of years ago and has been used primarily for access to information through a web service. The movement toward REST as the primary means for manipulating SDNs is based on a number of its perceived advantages:

- **Simplicity.** The web-based REST mechanism utilizes the simple HTTP GET, PUT, POST, and DELETE commands. Access to data involves referencing *resources* on the target device using normal and well-understood URL encoding.
- **Flexibility.** Requesting entities can access the defined configuration components on a device using REST resources, which are represented as separate URLs. There is no need for complicated schemas or MIB definitions to make the data accessible.
- **Extensibility.** Support for new resources does not require re-compilation of schemas or MIBs but only requires that the calling application access the appropriate URL.
- **Security.** It is very straightforward to secure REST communications. Simply running this web-based protocol through HTTPS adequately addresses security concerns. This has the advantage of easily penetrating firewalls, a characteristic not shared by all network security approaches.

A potential risk in using a protocol as flexible and extensible as REST is that it lacks the strict formalism and type checking of other methods. Although this may be true, in the trade-off between ease of use versus strict controls, REST appears to be gaining an ever-stronger beachhead as the API of choice for SDN via API networks.

It is important to point out that thus far we have been discussing APIs for devices such as switches. This is the context in which we have referred to REST as well as the earlier API technologies. The reader should note that REST is also frequently used as a northbound API to the controller. Such northbound APIs are used at different levels of the controller and may exist as high as the management layer that oversees a cluster of controllers. Since this kind of northbound API plays an important role in how network programmers actually program an SDN network, they certainly are SDN-appropriate APIs. Nonetheless, the reader should distinguish these from the device APIs that we refer to when discussing SDN via APIs.

6.2.4 Examples of SDN via APIs

Cisco has its own proprietary SDN program called *Cisco onePK* [7] that consists of a broad set of APIs in the devices. These APIs provide SDN functionality on legacy Cisco switches. Cisco onePK does not rely on traditional switch APIs such as the CLI but extends the switch software itself to support a richer API to support user application control of routing and traffic steering, network automation, policy control, and security features of the network, among others. Cisco onePK is used in conjunction with the Cisco ONE controller.[2]

A hybrid approach is the OpenDaylight open source project [8]. OpenDaylight supports OpenFlow as one of its family of southbound APIs for controlling network devices. Used in this way, OpenDaylight clearly fits into the Open SDN camp. The OpenDaylight controller also supports southbound APIs that

[2]The core technology of the Cisco ONE controller is the open source basis for its commercial XNC controller, which we discuss in Section 12.8.4.

program the legacy control plane on network devices. When OpenDaylight is used with those legacy southbound APIs, it falls into the controller-based SDN-via-APIs category. Due to this chameleon-like role, we do not categorize it as exclusively either SDN via APIs or Open SDN. Note that the Cisco ONE controller is the source-code basis for the OpenDaylight controller. We discuss the OpenDaylight program in more detail in Chapter 11.

Arista is another commercial example of a company that asserts that SDN is not about the separation of the control plane from the data plane and is instead about scaling the existing control and data plane with useful APIs [9]. Arista is basing its company's messaging squarely around this API-centric definition of SDN. Arista claims that it supports both centralized controllers and an alternative of distributed controllers that may reside on devices themselves.

6.2.5 Ranking SDN via APIs

Table 6.1 provides a ranking of SDN via APIs versus the other technologies discussed in this chapter. As stated in the chapter introduction, despite our best efforts there is surely some subjectivity in this ranking. Rather than assign strict letter grades for each criterion, we simply provide a high, medium, or low ranking as an assessment of how these alternatives stack up against one another. We said earlier that for the sake of clarity we have tightly constrained our definitions of each category, which may exclude an implementation that an individual reader could consider important. We also acknowledge that some criteria that are extremely important for some network professionals or users could be entirely absent from our list. For this, we apologize, and we hope that this attempt at ranking at least serves to provoke productive discussions.

The first five criteria are those that define how well a technology suits our definition of an SDN technology. Of these five, the only criterion in which SDN via APIs seems to qualify as a true SDN technology is in the area of improved network automation and virtualization. Networking devices that are changed only in that they have better APIs are certainly not simplified devices. Such devices retain the ability to operate in an autonomous and independent manner, which means that the device itself will be no simpler and no less costly than it is today. Indeed, by our very definition, these APIs still program the locally resident control plane on the device. SDN via APIs does, however, partly move toward a centralized controller model. For this reason, we give it a grade of medium for centralized controller. The APIs defined by vendors are generally unique to their own devices, and hence they are usually proprietary.[3] The end result is that SDN applications written to one vendor's set of APIs will often not work with network devices from another vendor. It might be true that the Cisco APIs have been mimicked by Cisco's competitors in order to make their devices interoperable with Cisco-based management tools, but this is not true interoperability. Cisco continually creates new APIs so as to differentiate itself from competitors, which puts its competitors in constant catch-up mode. Finally, the control plane remains in place in the device, and only a small part of forwarding control has been moved to a centralized controller.

When ranked against the drawbacks of Open SDN, however, this technology fares somewhat better. Since the network is still composed of the same autonomous legacy switches with distributed control,

[3]An important exception relevant to SDN is Nicira's *Open vSwitch Database Management Protocol* (OVSDB). OVSDB is an open interface used for the configuration of OVS virtual switches.

there is neither too much change nor a single point of failure. We rank it as medium with respect to performance and scale, since legacy networks have proven that they can handle sizable networks but are showing inability to scale to the size of mega data centers. SDN-via-APIs solutions support neither deep packet inspection nor stateful flow awareness. Nothing in these solutions directly addresses the problems of MAC forwarding table or VLAN ID exhaustion.

So, in summary, SDN via APIs provides a mechanism for moving toward a controller-based networking model. Additionally, these APIs can help alleviate some of the issues raised by SDN around the need to have a better means of automating the changing of network configurations in order to attempt to keep pace with what is possible in virtualized server environments. However, SDN via APIs falls short of meeting many other goals of SDN.

6.3 SDN via Hypervisor-Based Overlays

In Chapter 4 we introduced SDN via hypervisor-based overlay networks. This hypervisor-based overlay technology creates a completely new virtual network infrastructure that runs independently on top of the underlying physical network, as depicted in Figure 4.8. In that diagram we saw that the overlay networks exist in an abstract form *above* the actual physical network below. The overlay networks can be created without requiring reconfiguration of the underlying physical network, which is independent of the overlay virtual topology.

With these overlays it is possible to create networks that are separate and independent of each other, even when VMs are running on the same server. As shown in Figure 4.8, a single physical server hosting multiple VMs can have each VM be a member of a separate virtual network. This VM can communicate with other VMs in its virtual network, but it usually does not cross boundaries and talk to VMs that are part of a different virtual network. When it is desirable to have VMs in different networks communicate, they do so using a virtual router between them.

It is important to notice in the diagram that traffic moves from VM to VM without any dependence on the underlying physical network other than its basic operation. The physical network can use any technology and can be either a layer two or a layer three network. The only matter of importance is that the virtual switches associated with the hypervisor at each endpoint can talk to each other.

What is actually happening is that the virtual switches establish communication tunnels among themselves using general IP addressing. As packets cross the physical infrastructure, as far as the virtual network is concerned they are passed directly from one virtual switch to another via virtual links. These virtual links are the tunnels we just described. The virtual ports of these virtual switches correspond to the *virtual tunnel endpoints* (VTEPs) defined in Section 4.6.2. The actual payload of the packets between these vSwitches is the original layer two frame being sent between the VMs. In Chapter 4 we defined this as *MAC-in-IP* encapsulation, which was depicted graphically in Figure 4.9. We provide a detailed discussion about MAC-in-IP encapsulation in Chapter 7.

Much as we did for SDN via APIs, we restrict our definition of SDN via hypervisor-based overlays to those solutions that utilize a centralized controller. We acknowledge that there are some SDN overlay solutions that do not use a centralized controller (e.g., MidoNet, which has MidoNet agents located in individual hypervisors), but we consider those to be exceptions that cloud the argument that this alternative is, generally speaking, a controller-based approach.

6.3.1 Overlay Controller

By our definition, SDN via hypervisor-based overlays utilizes a central controller, as do the other SDN alternatives discussed thus far. The central controller keeps track of hosts and edge switches. The overlay controller has knowledge of all hosts in its domain, along with networking information about each of those hosts, specifically their IP and MAC addresses. These hosts will likely be virtual machines in data center networks. The controller must also be aware of the edge switches that are responsible for each host. Thus, there will be a mapping between each host and its adjacent edge switch. These edge switches will most often be virtual switches associated with a hypervisor resident in a physical server and attaching the physical server's virtual machines to the network.

In an overlay environment, these edge switches act as the endpoints for the tunnels that carry the traffic across the top of the physical network. These endpoints are the VTEPs mentioned above. The hosts themselves are unaware that anything other than normal layer two Ethernet forwarding is being used to transport packets throughout the network. They are unaware of the tunnels and operate just as though there were no overlays involved whatsoever. It is the responsibility of the overlay controller to keep track of all hosts and their connecting VTEPs so that the hosts need not concern themselves with network details.

6.3.2 Overlay Operation

The general sequence of events involved with sending a packet from host A to a remote host B in an overlay network is as follows:

1. Host A wants to send payload P to host B.
2. Host A discovers the IP address and MAC address for host B using normal methods (DNS and ARP). Host A uses its ARP broadcast mechanism to resolve host B's MAC address, but the means by which this layer two broadcast is translated to the tunneling system varies. The original VXLAN tries to map layer two broadcasts directly to the overlay model by using IP multicast to flood the layer two broadcasts. This allows the use of Ethernet-style *MAC address learning* to map between virtual network MAC addresses and the virtual network IP addresses. There are also proprietary control plane implementations whereby the tunnel endpoints exchange the VM-MAC address to VTEP-IP address mapping information. Another approach is to use MP-BGP to pass MPLS VPN membership information between controllers where MPLS is used as the tunneling mechanism. In general, this learning of the virtual MAC addresses across the virtual network is the area where virtual network overlay solutions differ most. We provide this sampling of some alternative approaches only to highlight these differences and do not attempt an authoritative review of the different approaches.
3. Host A constructs the appropriate packet, including MAC and IP addresses, and passes it upstream to the local switch for forwarding.
4. The local switch takes the incoming packet from host A, looks up the destination VTEP to which host B is attached, and constructs the encapsulating packet as follows:
   - The outer destination IP address is the destination VTEP and the outer source IP address is the local VTEP.
   - The entire layer two frame that originated from host A is encapsulated within the IP payload of the new packet.
   - The encapsulating packet is sent to the destination VTEP.

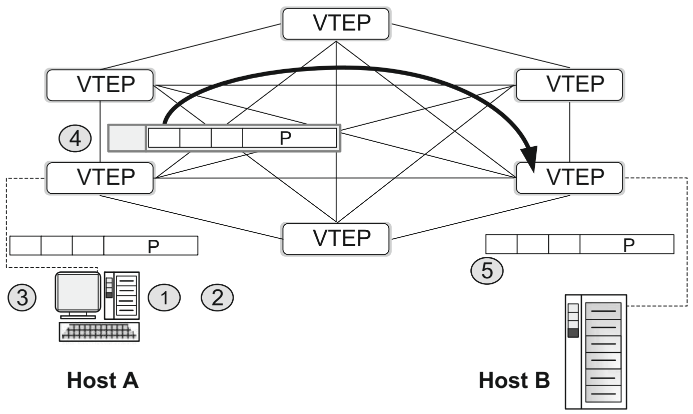


FIGURE 6.11

Overlay operation.

5. The destination VTEP receives the packet, strips off the outer encapsulation information, and forwards the original frame to Host B.

Figure 6.11 shows these five steps. From the perspective of the two hosts, the frame transmitted from one to the other is the original frame constructed by the originating host. However, as the packet traverses the physical network, it has been encapsulated by the VTEP and passed directly from one VTEP to the other, where it is finally decapsulated and presented to the destination host.

6.3.3 Examples of SDN via Hypervisor-Based Overlays

Prior to Nicira's acquisition by VMware in 2012, Nicira's *Network Virtualization Platform* (NVP) was a very popular SDN via hypervisor-based overlays offering in the market. NVP has been widely deployed in large data centers. It permits virtual networks and services to be created independently of the physical network infrastructure below. Since the acquisition, NVP is now bundled with VMware's *vCloud Network and Security* (vCNS) and is marketed as VMware NSX [10],[4] where it continues to enjoy considerable market success. The NVP system includes an open source virtual switch, *Open vSwitch*, (OVS) that works with the hypervisors in the NVP architecture. Open vSwitch has become the most popular open source virtual switch implementation, even outside of NVP implementations. Interestingly, NVP uses OpenFlow as the southbound interface from its controller to the OVS switches that form its overlay network. Thus, it is both an SDN via hypervisor-based overlays *and* an Open SDN implementation.

Juniper's JunosV Contrail [17], however, does not use OpenFlow. Contrail is a controller-based virtual overlay approach to network virtualization. It uses the *Extensible Messaging and Presence Protocol* (XMPP) to program the control plane of the virtual switches in its overlay networks. Because

[4]Since we focus on the NVP component in this work, we usually refer to that component by its original, pre-acquisition name rather than NSX.

the southbound API is XMPP, it is indeed a standards-based approach. Since the control plane still resides on those virtual switches and XMPP is not directly programming the data plane, we do not consider it Open SDN. It is clearly an SDN via hypervisor-based overlays approach, though.

Another important SDN via hypervisor-based overlays offering is IBM's *Software Defined Network for Virtual Environments* (SDN VE) [11]. SDN VE is based on IBM's established overlay technology, *Distributed Overlay Virtual Ethernet* (DOVE).

6.3.4 Ranking SDN via Hypervisor-Based Overlays

In Table 6.1 we include our ranking of SDN via hypervisor-based overlays versus the alternatives. SDN via hypervisor-based overlays is founded on the concepts of network automation and virtualization and, thus, scores high in those categories. With respect to openness, this depends very much on the particular implementation. NVP, for example, uses OpenFlow, OVSDB, and OVS, all open technologies. Other vendors use proprietary controller-based strategies. There are both closed and open subcategories within this alternative, so we have to rate it with a dual ranking [$Medium$, $High$] against this criterion, depending on whether or not the overlay is implemented using Open SDN. Because we restricted our definition of SDN via hypervisor-based overlays to controller-based strategies, it ranks high in this category. With respect to device simplification, the overlay strategy does allow the physical network to be implemented by relatively simple IP layer three switches, so we give this a ranking of medium with respect to device simplification.

A sidenote is called for here. The SDN via hypervisor-based overlays is often touted as delivering network virtualization on top of the existing physical network. Taken at face value, this does not force any device simplification, it merely allows it. A customer may be very pleased to be able to keep his very complex switches if the alternative means discarding that investment in order to migrate to simple switches. This actually holds true for OpenFlow as well. Many vendors are adding the OpenFlow protocol to existing complex switches, resulting in hybrid switches. Thus, OpenFlow itself does not mandate simple devices either, but it does allow them.

Plane separation is more complicated. In the sense that the topology of the physical network has been abstracted away and the virtual network simply layers tunnels on top of this abstraction, the control plane for the virtual network has indeed been separated. The physical switches themselves, however, still implement locally their traditional physical network control plane, so we give this a score of medium.

The criterion of too much change is also difficult to rank here. On one hand, the same physical network infrastructure can usually remain in use, so no forklift upgrade of equipment is necessarily required. On the other hand, converting the network administrators' mindset to thinking of virtual networks is a major transformation, and it would be wrong to underestimate the amount of change this represents. Thus, we rank this medium.

Since SDN via hypervisor-based overlays may or may not be based on the centralized controller concept, it gets an N/A ranking in the single-point-of-failure category. Similarly, we feel that it deserves the same medium ranking in performance and scale as Open SDN. It fares better than Open SDN with respect to deep packet inspection and stateful flow awareness because the virtual switches are free to implement it under the overlay paradigm. They may map different flows between the same hosts to different tunnels. This freedom derives from the fact that these virtual switches may implement any propriety feature, such as deep packet inspection, and are not restricted by a standard like OpenFlow.

Since these features may be supported by proprietary overlay solutions but are not currently supported by some of the largest overlay incumbents, we give these both a medium ranking.

SDN via hypervisor-based overlays receives the highest marks in the final two categories. The MAC forwarding table size problem is solved, since the physical network device deals with only the MACs of the VTEPs, which is a much smaller number than if it had to deal with all the VM MAC addresses. Similarly, since virtualization can be achieved by tunnels in this technology, the system does not need to over-rely on VLANs for virtualization.

To summarize, because SDN via hypervisor-based overlay networks was specifically designed for the data center, it was not tailored to other environments such as campus, service provider, carrier, and transport networks. For this reason, though it directly addresses some major needs in the data center, it has fewer applications in other networking environments. In the overlay alternative, the underlying physical network does not fundamentally change. As we examine SDN use cases outside the data center in Chapter 8, we will see that in many cases the solution depends on changing the underlying physical network. In such cases, the overlay approach is clearly not appropriate. SDN via hypervisor-based overlays is an important solution gaining considerable traction in data center environments, but it does not necessarily offer the device simplification or openness of Open SDN.

6.4 SDN via Opening Up the Device

There is another approach to SDN that has received considerably less attention than the other alternatives. This approach attempts to provide SDN characteristics by opening up the networking device itself for software development. We examine this alternative in the sections that follow.

We have stated that traditional networking devices themselves are closed environments. Only Cisco engineers can write networking software for Cisco devices, only Juniper engineers can write software for Juniper devices, and so on. The ramifications of this situation are that the evolution of networking devices has been stunted because so few people are enabled to innovate and develop new networking technologies.

The approaches we have discussed thus far, Open SDN, SDN via APIs, and SDN via hypervisor-based overlays, generally push networking intelligence off the switch onto a controller, which sees the entire network and on which it is possible to write networking applications. To varying degrees, these applications may be written by developers from outside the switch manufacturers. We have explained the advantages of this in our discussion of the SDN trait of openness. A fundamentally different approach to SDN is to provide networking devices that *are themselves open*, capable of running software that has been written by the open source community.

This *Open Device* concept is represented in Figure 6.12. The device itself, at the lower levels, is the same as in any other device, with L2 and L3 forwarding tables and TCAMs for the more complex matching and actions. However, above this layer is an API that allows networking software applications to be written and installed into the device. Under the Open Device approach, the current distributed, independent, and autonomous device model remains unchanged. The network intelligence is still distributed throughout the network on the devices, and there is no centralized controller. The nature of the networking devices changes from being closed and proprietary to being open and standardized, such that any outside party can write software to run on these open devices. To be successful, there needs to be

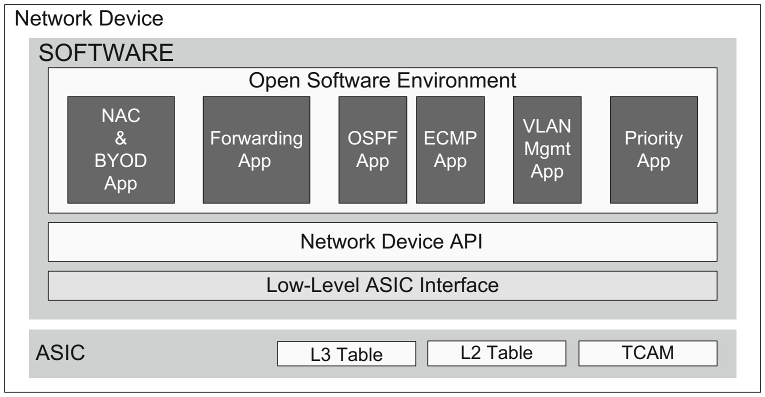


FIGURE 6.12

Opening up the device.

some level of standardization of the APIs that will be used by the networking software running on these devices. This is similar to the experience with Linux, where the operating system of the networking device provides some APIs that are relatively universal so that software can be written once and run on multiple open device platforms. It is important to recognize that the Linux experience was possible only because of the prevalence of a standardized PC hardware platform, which is a result of the Wintel de facto partnership.

This situation does not yet exist with enterprise-class switches but may begin to occur as more *original service manufacturers* (ODMs) build enterprise-class switches all based on the same reference design from the same merchant silicon vendors such as Broadcom or Intel. Since the switching chip manufacturers are few in number, the number of different reference designs will be small, and we may see de facto standardization of these switching platforms. Interestingly, this is already occurring with consumer wireless router hardware, which provides a concrete example of how *opening up the device* can work.

OpenWRT [18] is an open source implementation of a consumer wireless router. This is a Linux-based distribution that has been extended to include the IEEE 802.11, routing, and switching software necessary to support the home AP functionality. OpenWRT has enjoyed considerable success due to the fact that the consumer AP ODMs have largely followed the model we described above: A very large number of devices are manufactured from the same reference design and thus are compatible from a software point of view. This is evidenced by the very long list of supported hardware documented in [18]. This hardware compatibility allows the OpenWRT to be *reflashed* onto the hardware, overwriting the software image that was installed at the factory. Since the source code for OpenWRT is open, developers can add extensions to the basic consumer AP and bring that hybrid product to market. Although OpenWRT is not an SDN implementation, the model it uses is very much that of opening up the device. The challenge in directly translating this model to SDN is that the consumer wireless device hardware is much simpler than an enterprise-class switch. Nonetheless, the sophistication of ODMs has been increasing, as is the availability and sophistication of reference designs for enterprise-class switches.

This approach to SDN is advocated primarily by the merchant silicon vendors who develop the ASICs and hardware that exist at the lower levels of the networking device. By providing an open API to anybody who wants to write networking applications, they promote the adoption of their products. Intel has taken a step in this direction with its open source *Data Plane Development Kit* (DPDK) [6,13]. Despite this momentum, such a standardized hardware platform for enterprise-class switches does not yet exist. The success of SDN via opening up the device is predicated on this standardization becoming a reality.

6.5 Network Functions Virtualization

We cannot complete our chapter on SDN alternatives without mentioning *network functions virtualization* (NFV), since there is confusion in the industry as to whether or not NFV is a form of SDN [12]. NFV is the implementation of the functionality of network appliances and devices in software. This means implementing a generalized component of your network such as a load balancer or IDS in software. It can even be extended to the implementation of routers and switches. This has been made possible by advancements in general-purpose server hardware. Multicore processors and network interface controllers that offload network-specific tasks from the CPU are the norm in data centers. Although these hardware advances have been deployed initially to support the large numbers of VMs and virtual switches required to realize network virtualization, they have the advantage that they are fully programmable devices and can thus be reprogrammed to be many different types of network devices. This is particularly important in the service provider data center, where the flexibility to dynamically reconfigure a firewall into an IDS [19] as a function of tenant demand is of enormous value.

The reason for the confusion about whether or not NFV is a form of SDN is that the two ideas are complementary and overlapping. SDN may be used very naturally to implement certain parts of NFV. NFV may be implemented by non-SDN technologies, however, just as SDN may be used for many purposes not related to NFV. In short, NFV is not just another approach to SDN.

6.6 Alternatives Overlap and Ranking

We have tried to define a clear framework and delineated four distinct SDN alternatives; however, the lines between these alternatives are sometimes blurred. For example:

- NVP uses OpenFlow as the southbound interface from its controller to the OVS switches that form its overlay network, so it is an SDN via hypervisor-based overlays *and* an Open SDN implementation.
- The OpenDaylight controller uses several different southbound APIs to control the network devices. One of those southbound APIs is OpenFlow. Thus, depending on the exact configuration, OpenDaylight may be considered to implement SDN via APIs or OpenSDN.

In Figure 6.13 we attempt to depict the possible overlap between our four major SDN approaches. The category of SDN via opening up the device is a difficult one for comparison. As Figure 6.13 indicates, this category actually subsumes all the other categories and includes alternatives not yet contemplated. After all, if the device is truly opened up, nothing restricts that implementation from sticking with any current or future paradigm. For example, Big Switch's Indigo provides OpenFlow switch software freely to device vendors who want to use it. This should encourage ODMs to build standard hardware

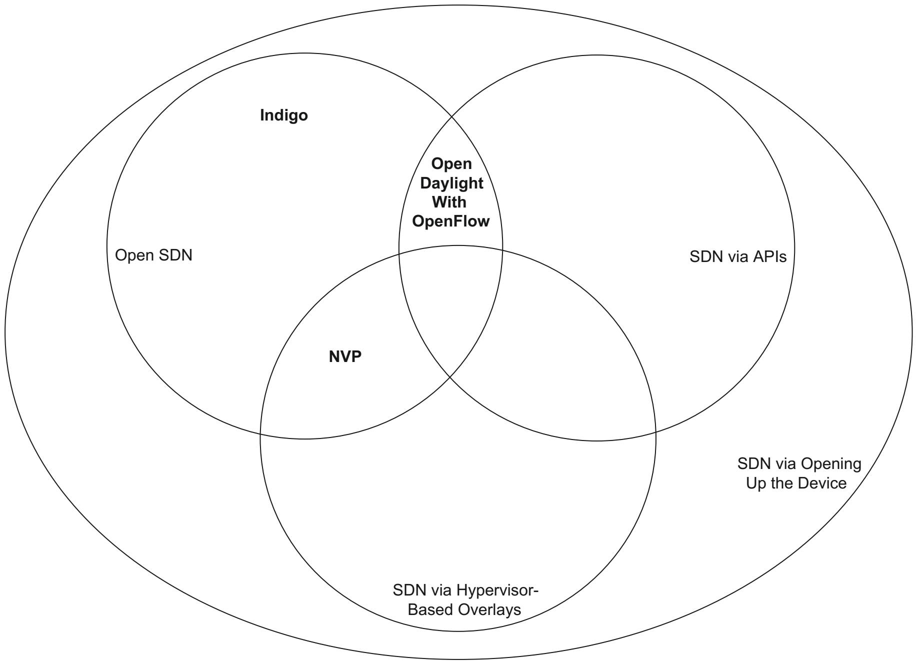

Indigo
Open Daylight With OpenFlow
Open SDN
SDN via APIs
NVP
SDN via Opening Up the Device
SDN via Hypervisor-Based Overlays

FIGURE 6.13

SDN alternatives overlap.

and software so that they can use software such as Indigo to OpenFlow-enable their switch without developing the switch-side OpenFlow software. So, one could argue that Indigo is an example of SDN via opening up the device. But Indigo is clearly an example of OpenSDN, because it is an implementation of OpenFlow. For this reason, we exclude SDN via opening up the device from Table 6.1, since the comparisons against something that can mean all things to all people are not reasonable.

Before examining Table 6.1 in detail, we remind the reader to recall certain constraints on our definitions of the alternatives:

- We restrict our definition of both SDN via APIs and SDN via hypervisor-based overlays to controller-based approaches.
- We restrict our definition of SDN via APIs to those approaches whereby the APIs reside on the devices and are used to program a locally resident control plane on those devices.
- It is possible to implement SDN via hypervisor-based overlays using Open SDN. We thus define two subcategories for this alternative. The first subcategory is not based on Open SDN, and the second uses Open SDN and, when appropriate, the ranking will reflect those two subcategories (i.e., $[not OpenSDN, OpenSDN]$).

Table 6.1 SDN Technologies Report Card

Criterion	Open SDN	SDN via APIs	SDN via Hypervisor-Based Overlays
Plane separation	High	Low	Medium
Simplified device	High	Low	Medium
Centralized control	High	Medium	High
Network automation and virtualization	High	High	High
Openness	High	Low	[Medium, high][a]
Too much change	Low	High	Medium
Single point of failure	Low	Medium	N/A
Performance and scale	Medium	Medium	Medium
Deep packet inspection	Low	Low	Medium
Stateful flow awareness	Low	Low	Medium
MAC forwarding table overflow	High	Low	High
VLAN exhaustion	High	Low	High

[a] *See Section 6.6 for an explanation of this bipartite ranking.*

6.7 Conclusion

In this chapter we examined some well-publicized drawbacks of Open SDN and looked at three alternative SDN definitions. We examined the ways in which each of these approaches to SDN meets our definition of what it means to be SDN as well as to what degree they suffer from those same well-publicized drawbacks. Each of the alternatives, SDN via APIs, SDN via hypervisor-based overlays, and SDN via open devices, has strengths and areas in which it addresses real SDN-related issues. Conversely, none of the alternatives matches our criteria for what constitutes an SDN system as well as does Open SDN. Thus, our primary focus as we study different use cases in the next two chapters will be how Open SDN can be used to address these real-life problems. We consider these alternative definitions where they have proven to be particularly effective at addressing a given use case.

References

[1] Casado M. Scalability and reliability of logically centralized controller. Stanford CIO Summit; June 2010.
[2] Yeganeh S, Ganjali Y. Kandoo: a framework for efficient and scalable offloading of control applications. In: Hot topics in software defined networking (HotSDN). Helsinki, Finland: ACM SIGCOMM; 2012.
[3] Wireless LAN controller (WLC) mobility groups FAQ. Cisco systems. Retrieved from <www.cisco.com/en/US/products/ps6366>.

[4] HP MSM7xx controllers configuration guide. Hewlett-Packard; March 2013.
[5] Configuring policy-based routing. Cisco systems. Retrieved from <www.cisco.com/en/US/docs/switches/lan/catalyst4500/12.2/20ew/configuration/guide/pbroute.html>.
[6] Intel DPDK: data plane development kit. Intel. Retrieved from <www.dpdk.org>.
[7] Cisco onePK developer program. Cisco systems. Retrieved from <developer.cisco.com/web/onepk-developer>.
[8] Open daylight technical overview. Retrieved from <www.opendaylight.org/project/technical-overview>.
[9] Bringing SDN to reality. Arista networks; March 2013. Retrieved from <www.aristanetworks.com>.
[10] NSX. VMware. Retrieved from <www.vmware.com/products/nsx>.
[11] Chua R. IBM DOVE takes flight with new SDN overlay, fixes VXLAN scaling issues, SDN Central; March 26, 2013. Retrieved from <www.sdncentral.com/news/ibm-dove-sdn-ve-vxlan-overlay/2013/03>.
[12] Pate P. NFV and SDN: what's the difference? SDN Central; March 30, 2013. Retrieved from <www.sdncentral.com/technology/nfv-and-sdn-whats-the-difference/2013/03>.
[13] McGillicuddy S. Intel DPDK, switch and server ref designs push SDN ecosystem forward, Techtarget; April 23, 2013. Retrieved from <searchsdn.techtarget.com/news/2240182264/Intel-DPDK-switch-and-server-ref-designs-push-SDN-ecosystem-forward>.
[14] Narayanan R, Kotha S, Lin G, Khan A, Rizvi S, Javed W, et al. Macroflows and microflows: enabling rapid network innovation through a split SDN data plane. European workshop on software-defined networking (EWSDN), Darmstadt, Germany; October 25–26, 2012.
[15] TR-069 CPE WAN management protocol. Issue 1, Amendment 4, Protocol version 1.3. The broadband forum technical report; July 2011.
[16] Creating innovative embedded applications in the network with the Junos SDK. Juniper networks; 2011. Retrieved from <www.juniper.net/us/en/local/pdf/whitepapers/2000378-en.pdf>.
[17] McGillicuddy S. Contrail: the Juniper SDN controller for virtual overlay network. Techtarget; May 9, 2013. Retrieved from <searchsdn.techtarget.com/news/2240183812/Contrail-The-Juniper-SDN-controller-for-virtual-overlay-network>.
[18] OpenWRT: Wireless freedom. Retrieved from <openwrt.org>.
[19] Jacobs D. Network functions virtualization primer: software devices take over. Techtarget; March 8, 2013. Retrieved from <searchsdn.techtarget.com/tip/Network-functions-virtualization-primer-Software-devices-take-over>.

CHAPTER 7

SDN in the Data Center

More than most mainstream deployments, the modern data center has stretched traditional networking to the breaking point. Previous chapters in this book highlighted the importance of data center technology in accelerating the urgency and relevance of SDN. In its various incarnations discussed thus far, SDN has evolved specifically to address the limitations of traditional networking that have presented themselves most dramatically in today's mega data center. In this chapter we delve more deeply into data center networking technology, both as it exists today and as it will adapt in the future through the benefit of SDN. Specifically, in this chapter we examine:

- **Data center needs.** What are the specific shortcomings that exist in data center networks today?
- **Data center technologies.** What technologies are employed in the data center, both now and with the advent of SDN?
- **Data center use cases.** What are some specific applications of SDN in the data center?

In answering these questions, we hope to illuminate the close relationship that exists between SDN's technical strengths and the modern data center.

7.1 Data Center Definition

Let us define what exactly a data center is so that we can better understand its needs and the technologies that have arisen to address those needs. The idea of a data center is not new. Densely packed racks of high-powered computers and storage have been around for decades, originally providing environments for mainframes. These eventually gave way to minicomputers and then to the servers that we know and deal with today. Over time the density of those servers and the arrays of storage that served them have made it possible to host a tremendous amount of compute power in a single room or pod.

Today's data centers hold thousands, even tens of thousands, of physical servers. These data centers can be segregated into the following three categories:

- **Private single-tenant.** Individual organizations that maintain their own data centers belong in this category. The data center is for the private use of the organization, and there is only the one organization or *tenant* using the data center.
- **Private multitenant.** Organizations that provide *specialized* data center services on behalf of other client organizations belong in this category. IBM and EDS (now HP) are examples of companies that host such data centers. These centers are built and maintained by the organization providing the service, and multiple clients store data there, suggesting the term *multitenant*. These data centers are private because they offer their services contractually to specific clients.

- **Public multitenant.** Organizations that provide *generalized* data center services to any individual or organization belong in this category. Examples of companies that provide these services include Google and Amazon. These data centers offer their services to the public. Anybody, whether individuals or organizations, who wants to use these services may access them via the web.

In the past, these data centers were often hosted such that they could be reached only through private communication channels. However, in recent years, these data centers have begun to be designed to be accessible through the Internet. These types of data centers are often referred to as residing in *the cloud*. Three subcategories of clouds are in common use: *public* clouds, *private* clouds, and *hybrid* clouds. In a public cloud, a service provider or other large entity makes services available to the general public over the Internet. Such services may include a wide variety of applications or storage services, among other things. Examples of public cloud offerings include *Microsoft Azure Services Platform* and *Amazon Elastic Compute Cloud*. In a private cloud, a set of server and network resources is assigned to one tenant exclusively and protected behind a firewall specific to that tenant. The physical resources of the cloud are owned and maintained by the cloud provider, which may be a distinct entity from the tenant. The physical infrastructure may be managed using a product such as VMware's vCloud. Amazon Web Services is an example of the way a private cloud may also be hosted by a third party (i.e., Amazon).

An increasingly popular model is the hybrid cloud, where part of the cloud runs on resources dedicated to a single tenant, but other parts reside on resources that are shared with other tenants. This is particularly beneficial when the shared resources may be acquired and released dynamically as demand grows and declines. Verizon's *cloud bursting*, which we present in Section 8.2.3, provides an example of how a hybrid cloud might work. Note that the ability to dynamically assign resources between tenants and to the general public is a major driver behind the interest in network virtualization and SDN.

Combining the increasing density of servers and storage and the rise in networking speed and bandwidth, the trend has been to host more and more information in ever-larger data centers. Add to that enterprises' desire to reduce operational costs, and we find that merging many data centers into a larger single data center is the natural result.

Concomitant with this increased physical density is a movement toward virtualization. Physical servers now commonly run virtualization software that allows a single server to actually run a number of virtual machines. This move toward virtualization in the compute space reaps a number of benefits, from decreasing power requirements to more efficient use of hardware and the ability to quickly create, remove, and grow or shrink applications and services.

One of the side effects of the migration to cloud computing is that it is usually associated with a usage-sensitive payment system. The three main business models of cloud computing are *Infrastructure as a Service* (IaaS), *Platform as a Service* (PaaS), and *Software as a Service* (SaaS). These three are all provided and sold on an on-demand, usage-sensitive basis, shifting CAPEX costs to OPEX. The popularity of this new business model has been a major driver for the explosion in the size and number of data centers. We discuss the business ramifications of this new model on SDN in Chapter 12.

Thus, the three trends toward cloud, increased density, and virtualization have placed demands on data center networking that did not exist before. We examine those needs in the next section.

7.2 Data Center Demands

As a result of the evolutionary trends identified in the previous section, a number of critical networking needs have arisen that must be met in order for data center technology to thrive. We mentioned some of these needs in previous chapters. We examine them in greater detail here.

7.2.1 Overcoming Current Network Limitations

The potential to easily ignite and tear down VMs is a radical departure from the physical world that network managers have traditionally faced. Networking protocols were coupled with physical ports, which is *not* the case moving forward. The dynamic nature and sheer number of VMs in the data center have placed demands on the capacity of network components that were earlier thought to be safe from such pressures. In particular, these areas include *MAC address table size*, *number of VLANs*, and *spanning tree*.

MAC Address Explosion

In switches and routers, the device uses MAC address table to quickly determine the port or interface out of which the device should forward the packet. For speed, this table is implemented in hardware. As such, it has a physical limit to its size. Naturally, device vendors will determine the maximum number of entries to hold in the MAC table based on controlling their own costs, at the same time providing an adequate number of entries for network demands. If the table is too large, the vendor will have spent too much money on creating the device. If the table is too small, the customer will experience the problems we describe here.

Networks in the past had manageable limits on the maximum number of MAC addresses that would need to be in the MAC address table at any given time. This was partly attributed to physical limitations of data centers and the number of servers that would be part of a single layer two network. It is also affected by the physical layout of the network. In the past, physically separate layer two networks remained logically separate. Now, with network virtualization and the use of Ethernet technology across WANs, the layer two networks are being stretched geographically as never before. With server virtualization, the number of servers possible in a single layer two network has increased dramatically. With numerous virtual network interface cards (NICs) on each virtual server, this problem of a skyrocketing number of MAC addresses is exacerbated. Layer two switches are designed to handle the case of a MAC table miss by flooding the frame out all ports except the one on which it arrived, as shown in Figure 7.1. This has the benefit of ensuring that the frame reaches its destination if that destination exists on the layer two network. Under normal circumstances, when the destination receives that initial frame, this prompts a response from the destination. Upon receipt of the response, the switch is able to learn the port on which that MAC address is located and populates its MAC table accordingly. This scheme works well unless the MAC table is full, in which case it cannot learn the address. Thus, frames sent to that destination continue to be flooded. This is a very inefficient use of bandwidth and can have significant negative impact on performance. This problem is exacerbated in the core of the data center network. When the traditional nonvirtualized network design is used, where all the MAC addresses are visible throughout the topology, the pressure on the MAC address tables is intense.

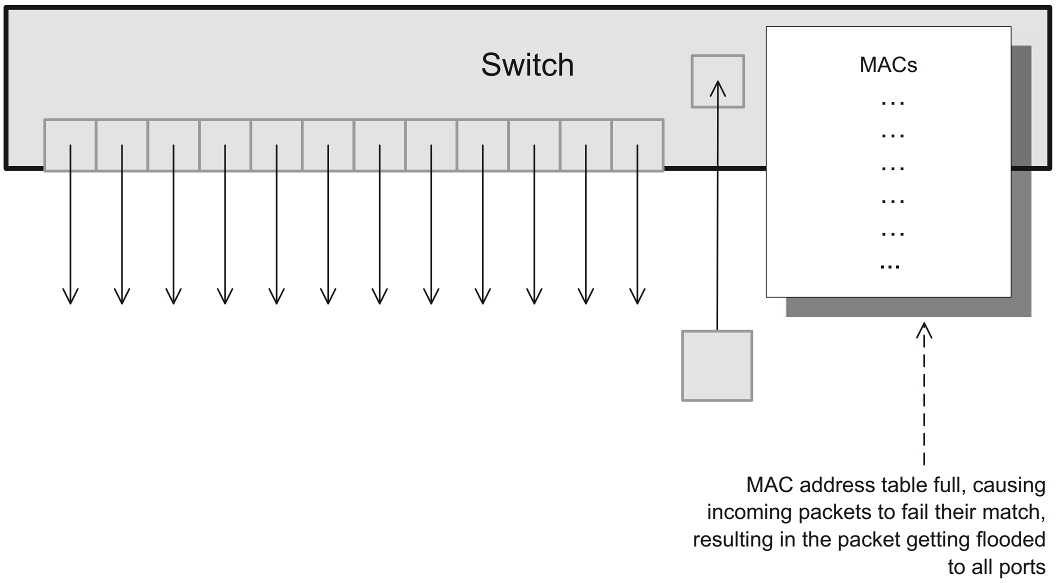

Switch
MACs
...
...
...
...
...
...
MAC address table full, causing incoming packets to fail their match, resulting in the packet getting flooded to all ports

FIGURE 7.1

MAC address table overflow.

Since VLANs are used extensively in modern layer two networks, let us take a closer VLAN-centric look at Figure 7.1. When a VLAN-tagged packet fails its match, it is flooded out only to all ports on that VLAN, somewhat mitigating the flooding inefficiency. Conversely, a host may be a member of multiple VLANs, in which case it will occupy one MAC address table entry per VLAN, further exacerbating the problem of MAC address table overflow.

Number of VLANs

When the IEEE 802.1 working group created the 802.1Q extension to the definition of local area networks, they did not anticipate that there would be a need for more than 12 bits to hold potential VLAN IDs. The IEEE 802.1Q tag for VLANs is shown in Figure 7.2. The tag depicted in Figure 7.2 supports 2^{12} (4096) VLANs. When these tags were introduced in the mid to late 1990s, networks were smaller and there was very limited need for multiple virtual domains within a single physical network.

The introduction of data centers created the need to segregate traffic so as to ensure security and separation of the various tenant networks' traffic. The technology responsible for providing this separation is VLANs. As long as data centers remained single-tenant networks, the maximum number of 4096 VLANs seemed more than sufficient. To have expanded this 12 bit field unnecessarily was not wise, since different tables in memory and in the ASICs had to be large enough to accommodate the maximum number. Thus, to have made the maximum larger than 4096 had a definite downside at the time the design work was performed.

When data centers continued to expand, however, especially with multi-tenancy and server virtualization, the number of VLANs required could easily exceed 4096. When there are no more available

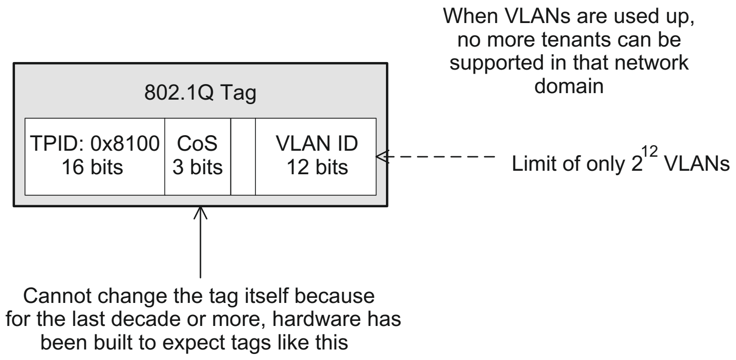

When VLANs are used up, no more tenants can be supported in that network domain
802.1Q Tag
TPID: 0x8100 16 bits
CoS 3 bits
VLAN ID 12 bits
Limit of only 2^12 VLANs
Cannot change the tag itself because for the last decade or more, hardware has been built to expect tags like this

FIGURE 7.2

VLAN exhaustion.

VLANs, the ability to share the physical resources among the tenants quickly becomes complex. Since the number of bits allocated to hold the VLAN is fixed in size and for years hardware has been built that depends on that specific size, it is nontrivial to expand the size of this field to accommodate more VLANs. Thus, some other solution is needed to overcome this limitation.

An upshot of the limit of 4096 VLANs has been an increase in the use of MPLS. MPLS does not suffer the same limitation in the number of MPLS tags as exists with VLAN IDs, and the tags need not be isolated to a single layer two network. Precisely because it is a layer three technology with the correspondingly more complex control plane, MPLS has primarily been deployed in WANs. It is likely, though, that MPLS will see more use in data centers [9]. We explore the use case of MPLS with SDN in the WAN in Section 8.1.1.

Spanning Tree

In the early years of Ethernet, bridges were built as *transparent* devices capable of forwarding packets from one segment to another without explicit configuration of forwarding tables by an operator. The bridges learned forwarding tables by observing the traffic being forwarded through them. Distributed intelligence in the devices was capable of collectively determining whether there were loops in the network and truncating those loops so as to prohibit issues such as broadcast storms from bringing down the network.

This was accomplished by the bridges or switches collectively communicating with one another to create a *spanning tree*, which enforces a loop-free hierarchical structure on the network in situations where physical loops do exist. This spanning tree was then calculated using the *Spanning Tree Protocol* (STP) we briefly reviewed in Section 1.5.1. The process of determining this spanning tree is called *convergence*, and in the early days it would take some time (dozens of seconds) for the spanning tree to converge after a physical change to the networking devices and their links had taken place.

Over time, through improvements to STP, the process of converging the spanning tree increased in speed. For example, with the improvements in recent versions of the standard, convergence times for conventionally large networks have decreased dramatically.[1] Despite these improvements, STP still leaves perfectly functional links unused and with frames being forwarded to the root of the spanning tree, which is certainly not universally the optimal path. Data centers need more *cross-sectional* bandwidth. By this we mean using the most efficient path between any two nodes without imposing an artificial hierarchy in the traffic patterns.

The fact that the fluidity of data center virtualization has increased the frequency of changes and disruptions, thus requiring reconvergence to occur more often, has only added to the inefficiency of STP in the data center. STP was simply not built for the modern data center world of virtual switches and ephemeral MAC addresses.

7.2.2 Adding, Moving, and Deleting Resources

Today's virtualized data centers are able to make changes much more quickly than in the past. Networks need to adapt in order to keep pace with the virtualization capabilities of servers and storage. Speed and automation are of critical importance when it comes to handling the rapid changes demanded by virtualized servers and storage.

It is clearly important that changes to networking infrastructure take place at the same rate as is possible with servers and storage. Legacy networks require days or weeks for significant changes to VLANs and other network plumbing needs. A large part of the reason for this time requirement is that the repercussions of a mistaken network change can easily impact all the data center resources, including not only networking but also its compute and storage components. Coordinating changes with all these disparate departments is unavoidably complex. These changes need to have the ability to be automated so that changes that must happen immediately can take place without human intervention. Legacy protocols were designed to react *after* the newly ignited service comes online. With SDN one can use the foreknowledge that a new service is about to be initiated and proactively allocate the network capacity it will require.

7.2.3 Failure Recovery

The size and scale of data centers today make recovery from failure a complex task, and the ramifications of poor recovery decisions are only magnified as scale grows. Determinism, predictability, and optimal re-configuration are among the most important recovery-related considerations.

With the distributed intelligence of today's networks, recovery from failures results in unpredictable behavior. It is desirable that the network move to a known state, given a particular failure. Distributed protocols make this difficult to do. A complete view of the network is required to make the recovery process yield the best result.

7.2.4 Multitenancy

Data center consolidation has resulted in more and more clients occupying the same set of servers, storage, and network. The challenge is to keep those individual clients separated and insulated from each other and to utilize network bandwidth efficiently.

[1]In practice, including link failure detection time, convergence times have been reduced to the order of hundreds of milliseconds.

In a large multitenant environment, it is necessary to keep separate the resources belonging to each client. For servers this could mean not mixing clients' virtual machines on the same physical server. For networks it could mean segregation of traffic using a technology that ensures that packets from two distinct tenants remain insulated from one another. This is needed not only for the obvious security reasons but also for QoS and other service guarantees.

With a small number of tenants in a data center, it was possible to arrange traffic and bandwidth allocation using scripts or some other rudimentary techniques. For example, if it was known that backups would be run at a certain time, arrangements could be made to route traffic appropriately around that heavy load. However, in an environment with hundreds (a large data center) or thousands (a cloud) of tenants with varying needs and schedules, a more fluid and robust mechanism is required to manage traffic loads. This topic is examined in more detail in the next section.

7.2.5 Traffic Engineering and Path Efficiency

As data centers have grown, so has the need to make better use of the resources being shared by all the applications and processes using the physical infrastructure. In the past this was less an issue because overprovisioning could make up for inefficiencies. But with the current scale of data centers, it is imperative to optimally utilize the capacity of the network. This entails proper use of monitoring and measurement tools for more informed calculation of the paths that network traffic takes as it makes its way through the physical network.

To understand traffic loads and take the appropriate action, the traffic data must be monitored and measured. Gathering traffic intelligence has often been a luxury for networks. But with the need to make the most efficient use of the links and bandwidth available, this task has become an imperative.

State-of-the-art methods of calculating routes use link-state technology, which provides the network topology for the surrounding network devices that are part of the same autonomous system (AS). This allows path computation to determine the *shortest path*. However, the shortest path as defined by traditional technology is not always the optimal path, since it does not take into consideration dynamic factors such as traffic load.

One of the reasons for the increasing attention on traffic engineering and path efficiency in the data center has been the rise of *East-West* traffic relative to *North-South* traffic. In Section 1.3 we defined these traffic types as follows: East-West *traffic is composed of packets sent by one host in a data center to another host in that same data center. Analogously, North-South traffic is traffic entering (leaving) the data center from (to) the outside world.*

Facebook provides a good example of what is driving the growth in East-West traffic. When you bring up your newsfeed page on Facebook, it has to pull in a multitude of data about different people and events in order to build the webpage and send it to you. That data resides throughout the data center. The server that is building and sending you your Facebook newsfeed page has to pull data from servers in other parts of the data center. That East-West traffic is at least as large as the North-South data (i.e., the final page it sends you) and in fact can be larger if you account for the fact that servers are moving data around even when no one is requesting it (via replication, staging data in different data centers, and so on).

In the older data center model, when most traffic was North-South, traffic engineering inside the data center was mostly a question of ensuring that congestion and loss would not occur as the data moved from the servers back toward the outside world. With so much East-West traffic, traffic engineering in the data center has become much more complex. These new traffic patterns are a large part of the reason for the interest in the *Ethernet fabrics* we discuss in Section 7.5.

We have now described some of the most critical needs that exist in data center networks as a result of server virtualization and the massive scale that occurs due to data center consolidation and the advent of the cloud. In response to these needs, the networking industry has developed new technologies to mitigate some of these issues. Some of these technologies are directly related to SDN, and some are not. We examine a number of these technologies in the sections that follow.

7.3 Tunneling Technologies for the Data Center

Previously in this chapter we have noted the impact of server virtualization. One of the responses to server virtualization has been the birth of the concept of *network virtualization*. We defined network virtualization in Section 3.7. We also discussed one of the alternative definitions of SDN, which we referred to as *SDN via hypervisor-based overlays*. At that time we showed how hypervisor-based tunneling technologies are employed to achieve this virtualization of the network. There are a number of tunneling technologies used in the data center, some of which predate SDN. Three main technologies are being used today to address many of the data center needs we presented in Section 7.2. These tunneling methods are *Virtual eXtensible Local Area Network* (VXLAN) [3], *Network Virtualization using Generic Routing Encapsulation* (NVGRE) [4], and *Stateless Transport Tunneling* (STT) [5].

All these tunneling protocols are based on the notion of encapsulating an entire layer two MAC frame inside an IP packet. This is known as *MAC-in-IP tunneling*. The hosts involved in communicating with each other are unaware that there is anything other than a traditional physical network between them. The hosts construct and send packets in exactly the same manner as they would had there been no network virtualization involved. In this way, network virtualization resembles server virtualization, where hosts are unaware that they are actually running in a virtual machine environment. Admittedly, the fact that the packets are encapsulated as they transit the physical network does reduce the *maximum transmission unit* (MTU) size, and the host software must be reconfigured to account for that detail.

All three of these tunneling methods may be used to form the tunnels that knit together the *virtual tunnel endpoints* (VTEPs) in an SDN via hypervisor-based overlays system.[2] Given this background and common behavior of these tunneling technologies, the next sections take a deeper look at these technologies and examine how they differ from one another.

7.3.1 Virtual eXtensible Local Area Network

The VXLAN technology was developed primarily by VMware and Cisco as a means to mitigate the inflexibility and limitations of networking technology. Some of the main characteristics of VXLAN are:

- VXLAN utilizes MAC-in-IP tunneling.
- Each virtual network or *overlay* is called a VXLAN segment.

[2]These are sometimes called *VXLAN tunnel endpoints*, which essentially means the same thing as the more generic and commonly used *virtual tunnel endpoint*.

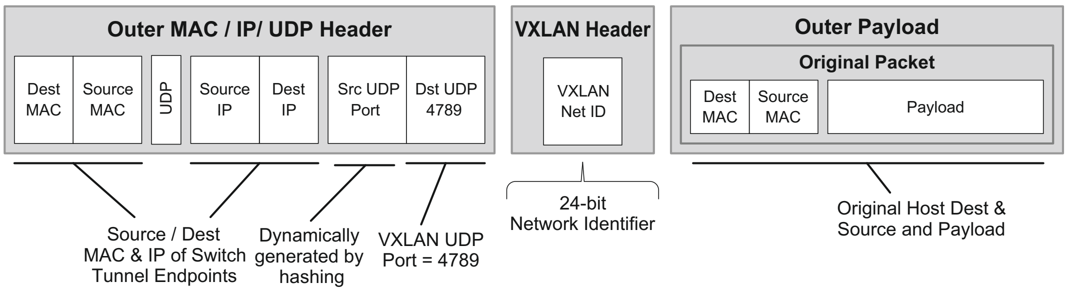

Outer MAC / IP/ UDP Header
Dest MAC
Source MAC
UDP
Source IP
Dest IP
Src UDP Port
Dst UDP 4789
VXLAN Header
VXLAN Net ID
Outer Payload
Original Packet
Dest MAC
Source MAC
Payload
Source / Dest MAC & IP of Switch Tunnel Endpoints
Dynamically generated by hashing
VXLAN UDP Port = 4789
24-bit Network Identifier
Original Host Dest & Source and Payload

FIGURE 7.3

VXLAN packet format.

- VXLAN segments are identified by a 24-bit segment ID, allowing for up to 2^{24} (approximately 16 million) segments.
- VXLAN tunnels are stateless.
- VXLAN segment endpoints are the switches that perform the encapsulation and are called *virtual tunnel endpoints* (VTEPs).
- VXLAN packets are unicast between the two VTEPs and use UDP-over-IP packet formats.
- It is UDP based. The UDP port number for VXLAN is 4789.

Figure 7.3 illustrates the format of a VXLAN packet. As you can see in the diagram, the outer header contains the MAC and IP addresses appropriate for sending a unicast packet to the destination switch, acting as a virtual tunnel endpoint. The VXLAN header follows the outer header and contains a VXLAN network identifier of 24 bits in length, sufficient for about 16 million networks.

The proponents of VXLAN technology argue that it assists load balancing within the network. Much of the load balancing within the data center network is based on the values of various packet fields, including the destination and source port numbers. The fact that the source port of the UDP session is derived by hashing other fields of the flow results in a smooth distribution function for load balancing with the network.

7.3.2 Network Virtualization Using GRE

The NVGRE technology was developed primarily by Microsoft, with other contributors including Intel, Dell, and Hewlett-Packard. Some of the main characteristics of NVGRE are:

- NVGRE utilizes MAC-in-IP tunneling.
- Each virtual network or *overlay* is called a virtual layer two network.
- NVGRE virtual networks are identified by a 24-bit virtual subnet identifier, allowing for up to 2^{24} (16 million) networks.
- NVGRE tunnels, like GRE tunnels, are stateless.
- NVGRE packets are unicast between the two NVGRE end points, each running on a switch. NVGRE utilizes the header format specified by the GRE standard [11,12].

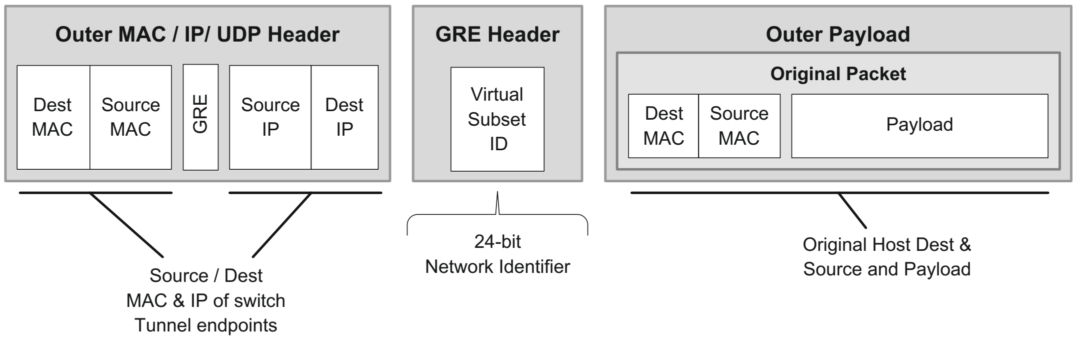

Outer MAC / IP/ UDP Header
Dest MAC
Source MAC
GRE
Source IP
Dest IP
GRE Header
Virtual Subset ID
Outer Payload
Original Packet
Dest MAC
Source MAC
Payload
Source / Dest MAC & IP of switch Tunnel endpoints
24-bit Network Identifier
Original Host Dest & Source and Payload

FIGURE 7.4

NVGRE packet format.

Figure 7.4 shows the format of an NVGRE packet. The outer header contains the MAC and IP addresses appropriate for sending a unicast packet to the destination switch, acting as a virtual tunnel endpoint, just like VXLAN. Recall that for VXLAN the IP protocol value was UDP. For NVGRE the IP protocol value is 0x2F, which means GRE. GRE is a separate and independent IP protocol in the same class as TCP or UDP. Consequently, as you can see in the diagram, there are no source and destination TCP or UDP ports. The NVGRE header follows the outer header and contains a NVGRE subnet identifier of 24 bits in length, sufficient for about 16 million networks.

7.3.3 Stateless Transport Tunneling

Stateless Transport Tunneling (STT) is a recent entry into the field of tunneling technologies used for network virtualization. Its major sponsor was originally Nicira. Some of the main characteristics of STT are:

- STT utilizes MAC-in-IP tunneling.
- The general idea of a virtual network exists in STT but is enclosed in a more general identifier called a context ID.
- STT context IDs are 64 bits, allowing for a much larger number of virtual networks and a broader range of service models.
- STT attempts to achieve performance gains over NVGRE and VXLAN by leveraging the *TCP Segmentation Offload* (TSO) found in the *network interface cards* (NICs) of many servers. TSO is a mechanism implemented on server NICs that allows large packets of data to be sent from the server to the NIC in a single send request, thus reducing the overhead associated with multiple smaller requests.
- STT, as the name implies, is stateless.
- STT packets are unicast between tunnel end points, utilizing TCP in the stateless manner associated with TSO. This means that it does not use the typical TCP windowing scheme, which requires state for TCP synchronization and flow control.

Figure 7.5 shows the format of an STT packet. Like the other tunneling protocols discussed here, the outer header contains the MAC and IP addresses appropriate for sending a unicast packet to the destination switch, acting as a VTEP. For VXLAN, the IP protocol value was UDP, and for NVGRE

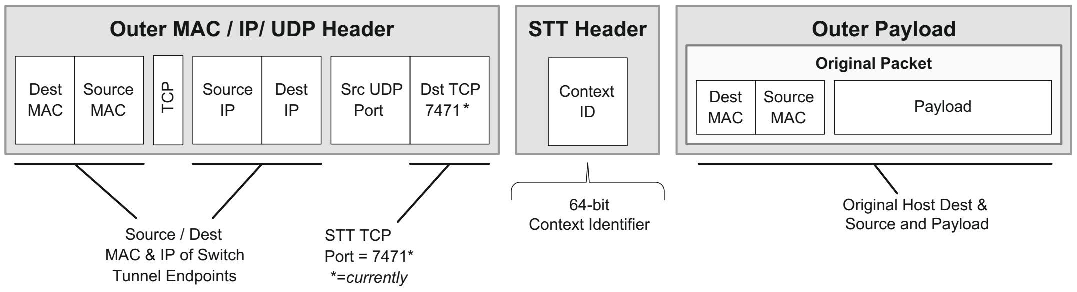

Outer MAC / IP/ UDP Header
Dest MAC
Source MAC
TCP
Source IP
Dest IP
Src UDP Port
Dst TCP 7471*
STT Header
Context ID
Outer Payload
Original Packet
Dest MAC
Source MAC
Payload
Source / Dest MAC & IP of Switch Tunnel Endpoints
STT TCP Port = 7471* *=currently
64-bit Context Identifier
Original Host Dest & Source and Payload

FIGURE 7.5

STT packet format.

the IP protocol value was GRE. For STT, the IP protocol is TCP. The TCP port for STT has yet to be ratified, but, tentatively, the value 7471 has been used. The STT header follows the outer header and contains an STT context identifier of 64 bits in length, which can be subdivided and used for multiple purposes; however that is done, there is ample space for as many virtual networks as required.

7.4 Path Technologies in the Data Center

One of the critical issues in current data centers is the efficient use of the physical network links that connect network devices forming the data center's network infrastructure. With the size and demands of data centers, it is imperative that all links be utilized to their full capacity.

As we explained in Section 7.2.1, one of the shortcomings of layer two network technologies such as spanning tree is that it will intentionally block certain links in order to ensure a hierarchical network without loops. Indeed, in the data center, this drawback has become more important than the problem of slow convergence times, which formed the bulk of the early critiques of STP.

Layer three networks also require intelligent routing of packets as they traverse the physical network. This section explores some path-related technologies that are intended to provide some of the intelligence required to make the most efficient use of the network and its interconnecting links.

7.4.1 General Multipath Routing Issues

In a typical network of layer three subnets, there will be multiple routers connected in some manner as to provide redundant links for failover. These multiple links have the potential to be used for load balancing. Multipath routing is the general category for techniques that make use of multiple routes in order to balance traffic across a number of potential paths. Issues that must be considered in any multipath scheme are:

- The potential for *out-of-order delivery* (OOOD) of packets that take different paths and must be reordered by the recipient, and
- The potential for maximum packet size differences on links within the different paths, causing issues for certain protocols such as TCP and its path MTU discovery.

7.4.2 Multiple Spanning Tree Protocol

The *Multiple Spanning Tree Protocol* (MSTP) was introduced to achieve better network link utilization with spanning tree technology when there are multiple VLANs present. Each VLAN would operate under its own spanning tree. The improved use of the links was to have one VLAN's spanning tree use unused links from another VLAN, when reasonable to do so. MSTP was originally introduced as IEEE 802.1s. It is now part of IEEE 802.1Q-2005 [1]. It was necessary to have a large number of VLANs in order to achieve a well-distributed utilization level across the network links, because the only distribution was at VLAN-granularity. MSTP predates the Shortest Path Bridging protocol discussed below, which does not suffer this limitation.

7.4.3 Shortest Path Bridging

IEEE 802.1aq is the *Shortest Path Bridging* (SPB) standard, and its goal is to enable the use of multiple paths within a layer two network. Thus, SPB allows all links in the layer two domain to be active.

SPB is a link state protocol, which means that devices have awareness of the topology around them and are able to make forwarding decisions by calculating the best path to the destination. It uses the *Intermediate System to Intermediate System* (IS-IS) routing protocol [6] to discover and advertise network topology and to determine the paths to all other bridges in its domain.

SPB accomplishes its goals using encapsulation at the edge of the network. This encapsulation can be either MAC-in-MAC (IEEE 802.1ah) or Q-in-Q (IEEE 802.1ad). We discussed Q-in-Q encapsulation briefly in Section 5.5.5.

Figure 7.6 shows the frame format for Q-in-Q. As shown in the figure, Q-in-Q means pushing VLAN tags into the packet such that the inner tag becomes the original VLAN, also called the *customer VLAN* or *C-VID*. The outer tag has multiple names, depending on the context of its use. They are *metro tag* for its use in Metro Ethernet backbones, *PE-VLAN* for its use in provider networks, and *S-VID* for its use in service provider VLANs. *C-VID* represents the original customer VLAN tag.

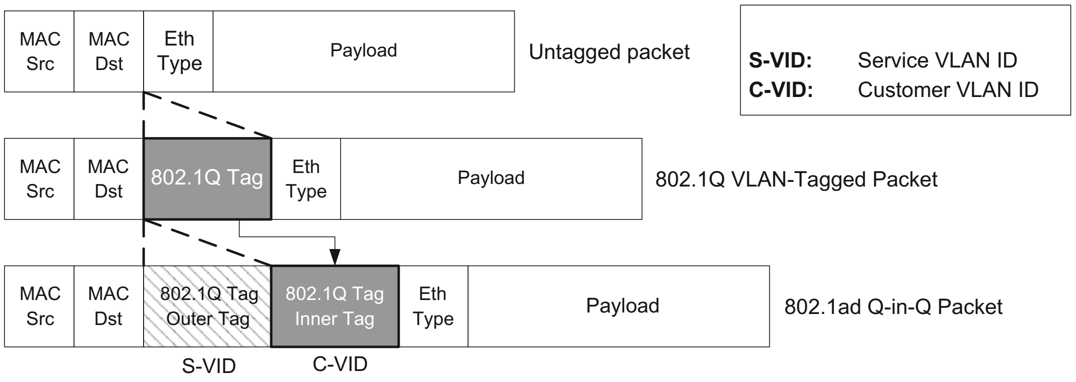


FIGURE 7.6

Shortest path bridging: Q-in-Q (VLAN).

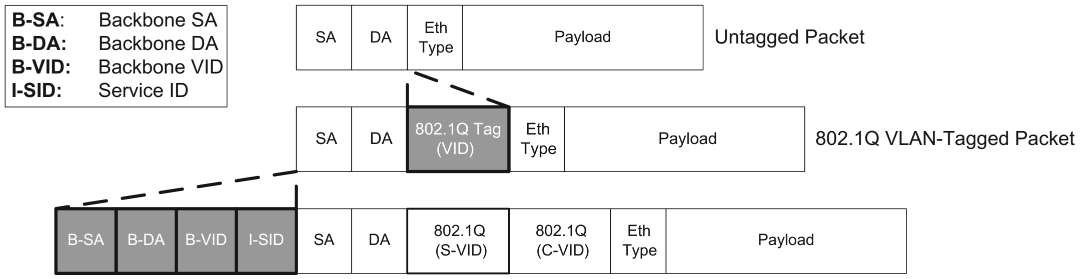


FIGURE 7.7

Shortest path bridging: MAC-in-MAC.

When the packet arrives at the edge of the provider's interior network, a new VLAN tag is pushed into the packet, identifying the VLAN on which the packet will travel when it is inside the provider's network. When it exits, that VLAN tag is removed.

Figure 7.7 shows the frame format for MAC-in-MAC. The new MAC header includes the backbone's source (SA) and destination (DA) addresses. Also part of the header is the VLAN identifier for the backbone provider's network (B-VID) as well as the backbone's service instance ID (I-SID), which allows the provider to organize customers into specific services and provide appropriate levels of QoS and throughput to each service. When a packet arrives at the provider edge, the new header gets attached to the beginning of the packet and is sent through the provider's network. When it leaves the provider's domain, the header is stripped.

7.4.4 Equal-Cost Multipath

One of the most important operations that must be undertaken in large networks is optimal path computation. Merely getting a packet to its destination is far simpler than getting it there in the most efficient way possible. In Section 1.5.2 we introduced OSPF and IS-IS as modern link-state protocols used for calculating routes in complex layer three networks. We just explained in Section 7.4.3 that IS-IS is used as the basis for calculating routes in the layer two SPB protocol. There is a general routing strategy called *equal-cost multipath* (ECMP) that is applicable in the data center. Multipath routing is a feature that is explicitly allowed in both OSPF and IS-IS. The notion is that when more than one equal-cost path exists to a destination, these multiple paths can be computed by a shortest-path algorithm and exposed to the packet-forwarding logic. At that point some load-balancing scheme must be used to choose between the multiple available paths. Because several routing protocols can derive the multiple paths and there are many ways in which to load-balance across the multiple paths, ECMP is more of a routing strategy than a specific technology. A discussion of some of the issues with ECMP can be found in [10]. An analysis of a specific ECMP algorithm is found in [8].

Another sophisticated path computation technology is the *Path Computation Element (PCE)-based architecture*. It has been used primarily in the WAN and not seen much use in the data center, so we defer this topic to Section 8.1.1.

7.4.5 SDN and Shortest-Path Complexity

Throughout this chapter we have made frequent reference to the importance of calculating optimal paths. We have implied that a centralized controller can perform this task better than the traditional distributed algorithm approach because (1) it has a more stable, global view of the network, (2) it can take more factors into consideration, including current bandwidth loads, than current distributed approaches do, and (3) the computation can be performed on the higher-capacity memory and processor of a server rather than the more limited memory and processor available on the network devices. Though all this is true, we should point out that the fundamental computational complexity of shortest-path calculation is not changed by any of these factors. Shortest-path remains a fundamentally difficult problem that grows harder very fast as the number of switches and links scales. The well-known Dijkstra's algorithm [2] for shortest-path remains in wide use (it is used in both OSPF and IS-IS), and SDN is unlikely to alter that. Having a faster machine and a more stable database of the network graph helps compute optimal paths more quickly; they do not affect the complexity of the fundamental problem.

7.5 Ethernet Fabrics in the Data Center

Traffic engineering in the data center is challenging using traditional Ethernet switches arranged in a typical hierarchy of increasingly powerful switches as one moves up the hierarchy closer to the core. Although the interconnecting links increase in bandwidth as we move closer to the core, these links are normally heavily oversubscribed, and blocking can easily occur. By oversubscription we mean that the aggregate potential bandwidth entering one tier of the hierarchy is greater than the aggregate bandwidth going to the next tier.

An alternative to a network of these traditional switches is the Ethernet fabric [7]. Ethernet fabrics are based on a nonblocking architecture whereby every tier is connected to the next tier with equal or higher aggregate bandwidth. This is facilitated by a topology referred to as *fat-tree* topology. We depict such a topology in Figure 7.8. In a fat-tree topology, the number of links entering one switch in a given tier of the topology is equal to the number of links leaving that switch toward the next tier. This implies that as we approach the core, the number of links entering the switches is much greater than those toward the leaves of the tree. The root of the tree will have more links than any switch lower than it. This topology is also often referred to as a *Clos* architecture. Clos switching architecture dates from the early days of crossbar telephony and is an effective nonblocking switching architecture. Note that the Clos architecture is not only used in constructing the network topology but may also be used in the internal architecture of the Ethernet switches themselves.

For Ethernet fabrics, the key characteristic is that the aggregate bandwidth not decrease from one tier to the next as we approach the core. Whether this is achieved by a larger number of low-bandwidth links or a lower number of high-bandwidth links does not fundamentally alter the fat-tree premise.

As we approach the core of the data center network, these interconnecting links are increasingly built from 10 Gbps links, and it is costly to simply try to overprovision the network with extra links. Thus, part of the Ethernet fabric solution is to combine it with ECMP technology so that *all* of the potential bandwidth connecting one tier to another is available to the fabric.

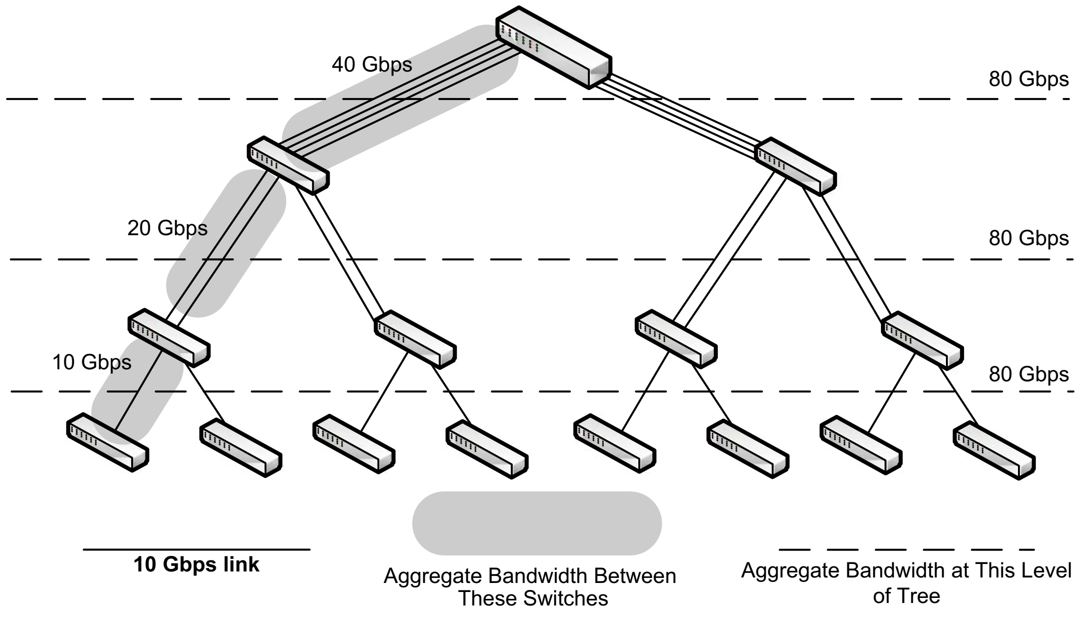

40 Gbps
80 Gbps
20 Gbps
80 Gbps
10 Gbps
80 Gbps
10 Gbps link
Aggregate Bandwidth Between These Switches
Aggregate Bandwidth at This Level of Tree

FIGURE 7.8

Fat-tree topology.

7.6 SDN Use Cases in the Data Center

We have described several definitions of Software Defined Networking: the original SDN, which we call Open SDN; SDN via APIs; and SDN via hypervisor-based overlays. We additionally described SDN via opening up the device, but here we focus on the first three, since they are the only ones to have gained significant commercial traction as of this writing. How do these versions of SDN address the needs of today's data center as we have described them in this chapter? The following sections answer that question.

For each data center requirement we presented in Section 7.2, we consider each of the three SDN technologies. We present them in the order SDN via overlays first, followed by Open SDN, and finally SDN via APIs. The reader should note that in this book, when we use the expression SDN via overlays, we mean SDN via *hypervisor-based* overlays. This distinction is important because there are other overlay mechanisms that are not hypervisor-based. The reason for this ordering is that SDN via overlays is a technology specifically addressing the data center, so one would expect the overlay solution to directly address most of these issues.

Of the three alternative SDN technologies, the one we consider least likely to address data center needs is SDN via APIs. Our rationale relies on the definition of SDN via APIs in Section 6.2, where we said that such APIs programmed a control plane local to the device. We believe that since the control

Table 7.1 Comparison of Alternatives in Addressing Data Center Needs

Need	SDN via Overlays	Open SDN	SDN via APIs
Overcoming network limitations	Yes	Yes	No
Adds, moves, deletes	Yes	Yes	Yes
Failure recovery	No	Yes	No
Multitenancy	Yes	Yes	No
Traffic engineering and path efficiency	No	Yes	*Some*

plane is locally resident in the network device, this approach to SDN is at a disadvantage with respect to the other two in its ability to bring the radical change needed to address data center needs. As such, this type of SDN is considered last.

Note that Table 7.1 briefly summarizes the ability of the various SDN definitions to address the data center needs that we have mentioned in this chapter. We discuss this table in the sections that follow.

7.6.1 Overcoming Current Network Limitations

In this section we examine how SDN via overlays, Open SDN, and SDN via APIs address the limitations of MAC address table size and maximum number of VLANs.

SDN via Overlays

The simplest and most readily available solution to these problems involves tunneling, so *SDN via overlays* is an obvious choice here. The only MAC addresses visible through the physical network are the MAC addresses of the tunnel endpoints, which are at the hypervisors. As a trivial example, if there are eight VMs per hypervisor, you have reduced the total number of MAC addresses by a factor of eight. If the tunnel endpoints are further upstream or the number of VMs per hypervisor is higher, the MAC address savings are even greater.

For the issue of VLAN exhaustion, i.e., exceeding the limit of 4096, this solution is superior because the new mechanism for multitenancy is tunnels, not VLANs. As we explained in Section 7.3, the number of tunneled networks or *segments* can be 16 million or greater using VXLAN, NVGRE, or STT tunneling technologies.

As for the issue of spanning tree convergence and using all links in the physical infrastructure, SDN via overlays does not address issues related to the physical infrastructure. The network designer would have to use current non-SDN technologies to mitigate these types of physical limitations.

Open SDN

Open SDN has the capability of addressing these network limitations as well. However, it does not *inherently* resolve these limitations in the same way as does SDN via overlays due to that alternative's basic nature of using tunnels. Moving control functionality off the device to a centralized controller does not directly address limitations such as MAC address table size and the maximum number of VLANs.

However, Open SDN is well suited to creating a solution that is an instance of SDN via overlays. The SDN controller can create tunnels as required at what will become the tunnel endpoints, and then OpenFlow rules are used to push traffic from hosts into the appropriate tunnel. Since hardware exists that has built-in tunneling support, SDN devices can be built that derive these benefits from tunneling but with the performance gain of hardware. Thus, Open SDN can solve these network limitations similarly to the overlay alternative.

SDN via APIs

Adding SDN APIs to networking devices does not directly address network limitations as we have discussed them in this chapter.

7.6.2 Adding, Moving, and Changing Resources

Agility, automation, fast and efficient adds, moves, and changes—this type of functionality is critical in data centers in order to keep pace with speeds brought about by automation of servers and storage. We now describe the ways in which SDN technology can address these issues.

SDN via Overlays

The primary attribute of SDN via overlays as it addresses adds, moves, and changes is that the technology revolves around virtualization. It does not deal with the physical infrastructure at all. The networking devices that it manipulates are most often the virtual switches that run in the hypervisors. Furthermore, the network changes required to accomplish the task are simple and confined to the construction and deletion of virtual networks, which are carried within tunnels that are created expressly for that purpose. These virtual networks are easily manipulated via software.

Consequently, the task of making adds, moves, deletes, and changes in an overlay network is quite straightforward and is easily automated. Because the task is isolated and constrained to tunnels, problems of complexity are less prevalent compared to what would be the case if the changes needed to be applied and replicated on all the physical devices in the network. Thus, many would argue that overlays are the simplest way to provide the automation and agility required to support frequent adds, moves, deletes, and changes.

A downside of this agility is that since the virtual networks are not tightly coupled with the physical network, it is easy to make these adds, moves, and changes without being certain that the underlying physical network has the capacity to handle them. The obvious solution is to over-engineer the physical network with great excess capacity, but this is not an efficient solution to the problem.

Open SDN

As we discussed in the previous section, if Open SDN is being used to create tunnels and virtual networks, it is straightforward for Open SDN to achieve the same results as discussed earlier. The task is to create the overlay tunnels as required and to use OpenFlow rules to push packets into the appropriate tunnels.

In addition to the advantages of virtual networks via tunnels, Open SDN offers the ability to change the configuration and operation of the physical network below—what some refer to as the *underlay*. The real advantage of this capability is described in Section 7.6.5.

SDN via APIs

APIs provide a programmatic framework for automating tasks that would otherwise require manual intervention. The ability to have a controller that is aware of server virtualization changes and can make changes to the network in response is a definite advantage. The downside of this solution revolves around the fact that the fundamental capabilities of the network have not changed. Neither network virtualization (overlays and Open SDN) nor fundamental improvement of the actual infrastructure (Open SDN) is a necessary component of the API solution.

7.6.3 Failure Recovery

Data centers today have mechanisms for achieving high availability, redundancy, and recovery in the case of failure. However, as mentioned earlier, these mechanisms have deficiencies in terms of the predictability and optimization of those recovery methods. Consequently, we would expect SDN to provide some assistance in meeting those specific needs.

Note that failures on the compute side, i.e., physical servers or virtual machines, are really just a subset of the previous section on adds, moves, and changes and have been addressed in that context.

SDN via Overlays

Because it does not deal at all with the physical network below it, overlay technology offers little in terms of improving the failure recovery methods in the data center. If there are failures in the physical infrastructure, those must be dealt with via the mechanisms already in place, apart from overlays. In addition, the interplay between the virtual and physical topologies can sometimes be difficult to diagnose when there are problems.

Open SDN

One of the stated benefits of Open SDN is that with a centralized controller the whole network topology is known and routing (or, in this case, rerouting) decisions can be made that are consistent and predictable. Furthermore, those decisions can incorporate other sources of data related to the routing decisions, such as traffic loads, time of day, even scheduled or observed loads over time. Creating a solution such as this is not trivial, but the SDN application responsible for such failure recovery functionality can leverage existing and emerging technologies, such as IS-IS and PCE.

SDN via APIs

The presence of improved APIs on network devices and a controller to use those APIs to automatically update the device provides some improvement in the realm of failure recovery. However, at the heart of the issue is whether the network devices are performing their own route and path management or whether that management is under the control of a centralized controller. If the APIs are giving only a slightly improved access to traditional configuration parameters, clearly SDN via APIs provides little value in this area. However, if those APIs are accompanied by the devices ceding their path decision-making functionality to the SDN controller, the APIs can furnish more value. This, however, is not the typical case for SDN via APIs.

7.6.4 Multitenancy

With large data centers and cloud computing, it is frequently the case that more and more tenants are passing traffic along the same physical network infrastructure and therefore sharing the same physical communications links. The traditional way of achieving the separation required by this sharing of network bandwidth has been through the use of VLANs. We have shown how the maximum number of VLANs is no longer adequate for today's data centers.

SDN via Overlays

Overlay technology resolves the multitenancy issue by its very nature through the creation of virtual networks that run on top of the physical network. These virtual networks substitute for VLANs as the means of providing traffic separation and isolation. In overlay technologies, VLANs are only relevant within a single tenant. For each tenant, there is still the 4096 VLAN limit, but that seems to currently suffice for a single tenant's traffic.

Open SDN

Open SDN can implement network virtualization using layer three tunnel-based overlays in a manner very similar to SDN via Overlays. Open SDN offers other alternatives as well, however. Other types of encapsulation (e.g., MAC-in-MAC, Q-in-Q) can also be employed to provide layer two tunneling, which can provide multiplicative increases in the number of tenants. For example, using Q-in-Q can theoretically result in 4096 times 4096 VLANs, or roughly 16 million, the same value as is possible with the Overlay solutions utilizing VXLAN or NVGRE.

SDN via APIs

Without some complementary changes to devices, such as providing virtual networks, SDN via APIs does not address the issue of multitenancy. Recall that in the SDN-via-APIs solution, much or all of the control functionality still resides on the device, and the main use of those APIs is to improve the speed and accuracy of configuration changes.

7.6.5 Traffic Engineering and Path Efficiency

In the past, networking hardware and software vendors created applications that measured traffic loads on various links in the network infrastructure. Some even recommended changes to the infrastructure in order to make more efficient use of network bandwidth. As good as these applications may have been, the harsh reality was that the easiest way to ensure optimal response times and minimal latency was to overprovision the network. In other words, the simplest and cheapest solution was almost always to purchase more and newer networking gear to make certain that links were uncongested and throughput was maximized.

However, with the massive expansion of data centers due to consolidation and virtualization, it has become imperative to make the most efficient use of networking bandwidth. *Service-level agreements* (SLAs) with clients and tenants must be met; at the same time, costs need to be cut in order for the data center service to be competitive. This has placed an added importance on traffic load measurements and in using that information to make critical-path decisions for packets as they traverse the network.

SDN via Overlays

In a similar manner to the issue of failure recovery, SDN via overlays does not have much to contribute in this area due to the fact that it does not attempt to affect the physical network infrastructure. Traffic loads as they pass from link to link across the network are not a part of the discussion concerning traffic that is tunneled across the top of the physical devices and interfaces. Thus, SDN via overlays is dependent on existing underlying network technology to address these types of issues.

Open SDN

Open SDN has major advantages here in the areas of centralized control and having complete control of the network devices. Open SDN has centralized control with a view of the full network and can make decisions predictably and optimally. It also controls the individual forwarding tables in the devices and, as such, maintains direct control over routing and forwarding decisions.

Two other major ways that Open SDN can impact the traffic moving through the network are (1) path selection and (2) bandwidth management on a per-flow basis. Open SDN can make optimal path decisions for each flow. It can also prioritize traffic down to the flow level, allowing for granular control of queuing, using methods such as IEEE 802.1 *Class of Service* (CoS), IP-level *Type of Service* (ToS), and *Differentiated Services Code Point* (DSCP).

Figure 7.9 provides an illustration of the difference between shortest path via the current node-by-node technology and optimal path, which is possible with an SDN controller that can take into consideration more network parameters such as traffic data in making its decisions.

Open SDN is explicitly designed to be able to control directly the switching hardware network traffic down to the flow level. We contrast this with the overlay alternative, where network traffic does exist at the flow level in the virtual links but not natively in the switching hardware. Therefore, Open SDN is particularly well suited to addressing traffic engineering and path efficiency needs.

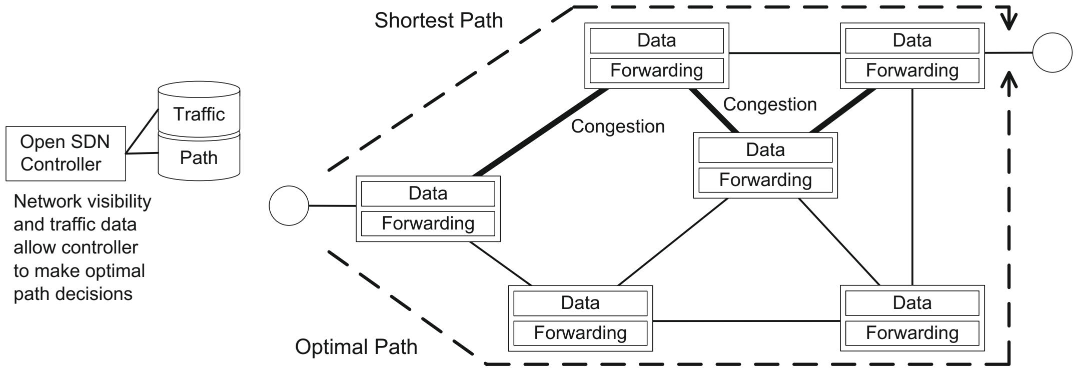


FIGURE 7.9

Optimal path selection by SDN controller.

SDN via APIs

The general capability of SDN via APIs is to set configuration of networking devices in a simpler and more efficient manner, allowing for automation and a certain amount of agility. However, although these APIs were generally not designed to handle the immediate and highly granular needs presented in traffic engineering challenges, it should be possible to support better traffic engineering with extended APIs, even in legacy devices with local control planes.

It is worth noting that *policy-based routing* (PBR) has the ability to direct packets across paths at a fairly granular level. In theory one could combine current traffic-monitoring tools with PBR and use current SNMP or CLI APIs to accomplish the sort of traffic engineering we discuss here. RSVP and MPLS-TE are examples of traffic engineering protocols that may be configured via API calls to the device's control plane. However, PBR and those APIs are not designed for regular and dynamic changes to paths and cannot react as quickly as OpenFlow in the face of change. Therefore, this type of solution is not ideal for traffic engineering and path efficiency.

7.7 Open SDN versus Overlays in the Data Center

We saw in Table 7.1 that SDN via overlays and Open SDN seem to have an advantage over SDN via APIs. For the moment, let's focus on these two technologies for a more fine-grained comparison.

7.7.1 SDN via Overlays

- The technology is designed for solving data center issues.
- The technology utilizes (for the most part) contemporary data center technologies, specifically VXLAN and NVGRE, and thus the technology is not entirely new.
- The technology creates a virtual overlay network and thus does a very good job of overcoming networking limitations and improving agility.
- The technology does not address any issues related to improving the behavior of the physical infrastructure. However, this has the advantage of incurring little or no costs related to new networking equipment.
- It is sometimes difficult to diagnose problems and determine whether they relate to the virtual network or the physical one.

7.7.2 Open SDN

- The technology is designed for solving networking issues in a wide range of domains, not just data centers.
- Open SDN can be implemented using networking devices designed for Open SDN. Legacy switching hardware can often be enhanced by adding OpenFlow to function as Open SDN devices.
- The technology, used in conjunction with virtual networks (e.g., VXLAN and NVGRE), does a very good job of addressing all the networking issues we have discussed.
- The technology is broader in scope; hence, it is more disruptive and may pose more transitional issues than does SDN via overlays.

- Although both Open SDN and SDN via overlays allow significant innovation, the Open SDN use cases discussed in Chapter 8 will demonstrate how implementors have *radically* redefined how the network is addressed and structured. In so doing, they abandon legacy constructs and structure the network to behave in exact alignment with application requirements.

These points indicate that SDN via overlays is a good solution and may be an easier transitional phase in the near term, whereas Open SDN holds the most promise in a broader and more comprehensive sense. Table 7.1 illustrated that SDN via APIs is unlikely to be a good long-term solution for the data center, but we should point out that *some* data center problems may be addressed in this way. This may in fact represent the easiest transition plan for network programmers in the short term because less radical change is needed than that required moving to either the overlay or Open SDN approach.

7.8 Real-World Data Center Implementations

In this chapter we have spent time defining the needs of modern data centers, and we have investigated the ways that each SDN technology has addressed those needs. One may wonder if anybody is actually implementing these SDN technologies in the data center. The answer is that, although it is still a work in progress, there are organizations that are running SDN pilots and in some cases placing SDN into production. In Table 7.2 we list a number of well-known enterprises and their early-adopter implementations of SDN in their data centers. The table shows a mix of Open SDN and SDN via overlays but no examples of SDN via APIs. The bulk of the API-oriented efforts are not really what we have defined as SDN. They use technologies similar to those we described in Sections 3.2.3 and 3.2.4—that is, orchestration tools and VMware plugins to manipulate networks using SNMP and CLI. Some experimentation with newer SDN APIs is presumably underway, but as of this writing production data center SDN networks tend more to be hypervisor-based overlays, with a few examples using Open SDN. Between Open SDN and the overlay approach, our observation is that the extremely large greenfield

Table 7.2 Data Center SDN Implementations

Enterprise	SDN Type	Description
Google	Open SDN	Has implemented lightweight OpenFlow switches, an OpenFlow controller, and SDN applications for managing the WAN connections between their data centers. Google is moving that Open SDN technology into the data centers
Microsoft Azure	Overlays	Implementing overlay technology with NVGRE in vSwitches, communicating via enhanced OpenFlow, creating tens of thousands of virtual networks
eBay	Overlays	Creating public cloud virtual networks using VMware's Nicira solution
Goldman Sachs	Open SDN	Using Floodlight-based application and OpenFlow-supporting commodity switches. Also replacing firewalls by augmenting OpenFlow-supporting switches for firewalling functionality
Rackspace	Overlays	Creating large multitenant public clouds using VMware's Nicira solution

data center examples gravitate toward Open SDN, whereas hypervisor-based overlays have seen greater adoption in the more typical-scale data centers.

7.9 Conclusion

This chapter described the many significant and urgent needs that have caused data center networking to become a stumbling block to technological advancement in that domain. We examined new technologies and standards that are attempting to at least partially address those needs. Then we presented a description of how the three main SDN alternatives use those technologies, as well as new ideas introduced by SDN to address those needs. Finally, we listed how some major enterprises are using SDN in their data center environments today.

In Chapter 8 we consider other use cases for SDN, such as WANs and backbones, carrier networks, enterprise networks, and network functions virtualization.

References

[1] IEEE standards for local and metropolitan area networks: virtual bridged local area networks. IEEE 802.1Q-2005, New York: IEEE; 2005.

[2] Dijkstra EW. A note on two problems in connexion with graphs. Numer Math 1959;1:269–71.

[3] Mahalingam M, Dutt D, Duda K, Agarwal P, Kreeger L, Sridhar T, et al. VXLAN: a framework for overlaying virtualized layer 2 networks over layer 3 networks. Internet Engineering Task Force; August 26, 2011 [internet draft].

[4] Sridharan M, Duda K, Ganga I, Greenberg A, Lin G, Pearson M, et al. NVGRE: network virtualization using generic routing encapsulation. Internet Engineering Task Force; September 2011 [internet draft].

[5] Davie B, Gross J. STT: a stateless transport tunneling protocol for network virtualization (STT). Internet Engineering Task Force; March 2012 [internet draft].

[6] Information technology, telecommunications and information exchange between systems, intermediate system to intermediate system intradomain routing information exchange protocol for use in conjunction with the protocol for providing the connectionless-mode network service (ISO 8473). ISO/IEC 10589:2002(E), Switzerland; November 2002.

[7] Ammirato J. Ethernet fabric switching for next-generation data centers. Network World; November 3, 2008, Retrieved from <www.networkworld.com/news/tech/2008/110308-tech-update.html>.

[8] Hopps C. Analysis of an equal-cost multi-path algorithm, RFC 2992. Internet Engineering Task Force; November 2000.

[9] Kompella K. New take on SDN: does mpls make sense in cloud data centers? SDN Central; December 11, 2012, Retrieved from <www.sdncentral.com/use-cases/does-mpls-make-sense-in-cloud-data-centers/2012/12>.

[10] Thaler D. Hopps C. Multipath Issues in Unicast and Multicast Next-Hop Selection, RFC 2991. Internet Engineering Task Force; November 2000.

[11] Farinacci D, Li T, Hanks S, Meyer D, Traina P. Generic Routing encapsulation (GRE), RFC 2784. Internet Engineering Task Force; March 2000.

[12] Dommety G. Key and sequence number extensions to GRE, RFC 2890. Internet Engineering Task Force; September 2000.

This page is intentionally left blank

CHAPTER

SDN in Other Environments

8

The urgency associated with the growth of data centers has caused SDN to leap to the forefront of solutions considered by IT professionals and CIOs alike. But the data center is not the only environment in which SDN is relevant. This chapter looks at other domains in which SDN will play a major role.

The following environments and some accompanying use cases are examined in this chapter:

- Wide area networks (Section 8.1)
- Service provider and carrier networks (Section 8.2)
- Campus networks (Section 8.3)
- Hospitality networks (Section 8.4)
- Mobile networks (Section 8.5)
- In-line network functions (Section 8.6)
- Optical networks (Section 8.7)

As we examine each of these domains, we explain the most pressing issues and then consider the ways in which SDN can play a part in helping to resolve them. We discuss current implementations and *proofs of concept* (PoCs) that have been developed to address needs in these areas. Before we look into each environment in detail, we review some advantages that accompany an SDN solution that are particular to these environments.

Consistent Policy Configuration

One of the major issues currently facing IT departments is the problem of scale related to having consistent configuration across very large populations of networking devices. In the past this task has fallen to manual configuration or network management, but with Open SDN there is the promise of great simplification of these tasks, with the associated reduction in operational costs for the enterprise. Some specifics of SDN with respect to configuration are (1) dealing with large numbers of devices, (2) consistency of configuration, (3) central policy storage, (4) common policy application, and (5) granularity of control.

The sheer number of devices operating in networks today has grown to the point of becoming nearly unmanageable. In the largest data centers, a few thousand physical switches might be deployed. IT personnel today are required to maintain such a large number of devices using traditional tools such as CLI, web interfaces, SNMP, and other manual, labor-intensive methods. SDN promises to remove many of these tasks. In order for SDN to work and scale, the controllers will be built from the ground up to comprehend and manage a huge number of devices. In addition, this centralization drives more commonality between the devices. The task of managing large numbers of devices is simplified as

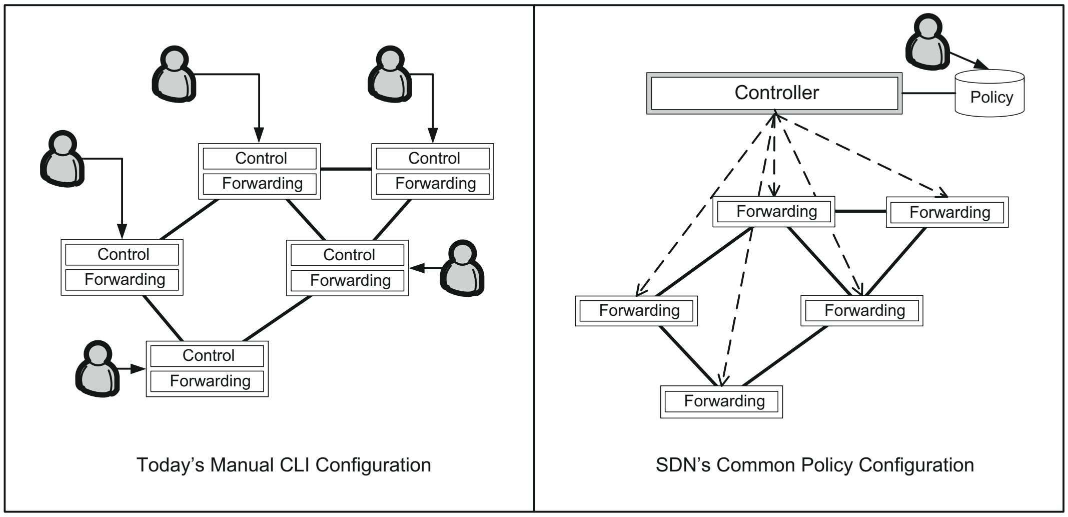

Control
Forwarding
Controller
Policy
Today's Manual CLI Configuration
SDN's Common Policy Configuration

FIGURE 8.1

Ensuring consistent policy configuration.

a direct result of this homogeneity. This simplification of network configuration is one of the major operational cost savings brought about by SDN.

One common issue with large networks today is the difficulty in maintaining consistency across all the devices that are part of the infrastructure. Figure 8.1 gives a simple example of the comparison of autonomous devices that have been individually and manually configured versus common policy distributed via flow rules from a central controller. The trivial case depicted in the diagram is compounded by orders of magnitude in real-life networks. Keeping configuration of policy consistent simply cannot be done by purely manual means. For some time now, sophisticated network management tools have existed that purport to provide centralized configuration and policy management. However, as we have described in previous chapters, these tools have only realized marginal success. They are generally based on a model of creating a *driver* that abstracts away vendor and device differences. These tools attempt to standardize at such a high level that the actual low-level vendor and device differences pose a constant challenge. This model has not been able to scale with the amount of change that is endemic in networking. SDN approaches configuration in a radically different way. Configuration is performed in a standard but entirely new and more fine-grained fashion at the flow level. The configuration is based on the fundamental construct of the flow, which is shared by all networking devices. This permits configuration by a common protocol such as OpenFlow. Rather than trying to bandage over a configuration morass with a system of the aforementioned drivers, this SDN approach has the ability to scale with the evolution of the network and to implement consistent network-wide configuration and policy. Central policy storage falls out naturally from centralized controller architecture of SDN. These attributes stem from SDN's very roots. We recall from Section 3.2.7 that Ethane, the direct precursor to OpenFlow, was specifically targeted at the distribution of complex policy on a network.

Global Network View

In previous chapters we mentioned the importance of network-wide visibility, noting this as a real value of controller-based networking. This is true for any type of system that relies on a central management and control system, such as phone systems, power grids, and the like. Having complete visibility allows the controlling system to make optimal decisions, as we discuss here.

The control plane of today's network devices has evolved to the point where each device holds the network topology represented as a graph. This solution is imperfect because the individual graphs held in each device are constructed by propagating changes across the network from neighbor to neighbor. This can result in relatively long periods before the devices converge on a consistent view of the network topology. During these periods of convergence, routing loops can occur. In such networks, it is cumbersome to understand exactly how the network will behave at any point in time. This legacy technology was geared toward maintaining basic connectivity, not the use cases we presented in Chapter 7 and continue to discuss here. In addition, the traditional networks do not have real-time data about traffic loads. Even maximum bandwidth information is not typically shared. A centralized control system is able to gather as much data (traffic loads, device loads, bandwidth limits) as is necessary to make the decisions required at any moment.

Having the aforementioned information makes it possible for the controller to make optimal routing and path decisions. Running on a processor that is many times more powerful than that present on a networking device increases the probability that the controller will be able to perform faster and higher-quality analysis in these decisions. Additionally, these decisions can take into account traffic patterns and baselines stored in secondary storage systems, which would be prohibitively expensive for networking devices. The distributed counterpart to this centralized control suffers the additional disadvantage of running a variety of protocol implementations on a selection of weaker processors, which creates a more unpredictable environment. Finally, if the capabilities of the physical server on which the controller is running are exceeded, the controller can use horizontal scaling approaches, something not possible on a networking device. In this context, horizontal scaling means adding more controllers and distributing the load across them. Horizontal scaling is not possible on the networking device, because such scaling would change the network topology. Changing the network topology does not provide more power to control the same problem; rather, it alters the fundamental problem and creates new ones.

A network-wide view takes the guesswork and randomness out of path decisions and allows the controller to make decisions that are reliable and repeatable. This deterministic behavior is critical in environments that are highly sensitive to traffic patterns and bandwidth efficiency, in which suboptimal behavior based on arbitrary forwarding assignments may cause unacceptable packet delays and congestion.

These common, general-use case values of SDN-based networks are relevant to just about every one of the environments described in this chapter. In certain cases they are of particular importance and are mentioned as such in those sections.

8.1 Wide Area Networks

Wide area networks have historically been used to connect remote islands of networks, such as connecting geographically dispersed LANs at different offices that are part of the same enterprise or organization.

A number of non-Ethernet technologies have been used to facilitate this WAN connectivity, including these:

- **Leased line.** These point-to-point connections were used in the early days of networking and were very expensive to operate.
- **Circuit switching.** Dialup connections are an example of this type of connectivity, which had limited bandwidth but was also very inexpensive.
- **Packet switching.** Carrier networks at the center carried traffic from one endpoint to another using technologies such as X.25 and Frame Relay.
- **Cell relay.** Technologies such as ATM, featuring fixed cell sizes and virtual circuits, are still used today.
- **Cable and *digital subscriber loop* (DSL).** These technologies play a key role in bringing WAN connectivity to the home.

However, recently these technologies have significantly given way to Ethernet-based WAN links, as we describe in greater detail in Section 8.2. Nowadays, companies and institutions are using Ethernet to connect geographically dispersed networks using either private backbones or, in certain cases, the Internet. When we use the term Ethernet in this context, we are referring to the use of Ethernet framing over the WAN optical links. We do not imply that these links operate as broadcast, CSMA/CD media, like the original Ethernet.

Ethernet has proven to be an excellent solution in areas not envisioned in the past. However, the move toward Ethernet brings with it not only the advantages of that technology but also many of those challenges that we have described in detail in this book. The following section addresses some of the major issues that are relevant for WAN environments.

The key issues that face network designers when it comes to WANs are reliability and making the most efficient use of available bandwidth. Due to the cost of long-haul links in the WAN, bandwidth is a scarce commodity compared to LANs. In environments such as a data center, it is possible to create redundant links with little concern for cost, since cables are relatively inexpensive. Furthermore, bandwidth needs can be met by adding more ports and links. Not so with WAN, where bandwidth costs are orders of magnitude greater over distance than they are within a data center or campus. Adding redundant ports just exacerbates this problem. As a result, it is important to drive higher utilization and efficiency in WAN links.

8.1.1 SDN Applied to the WAN

Overcoming the loss of connectivity on a link using redundancy and failover is common in all forms of networking, including WANs. This technology exists in devices today and has existed for many years. However, routing decisions made during a failover are not predictable during the period of convergence and often are not optimal even after convergence. The suboptimal paths are due to the lack of a central view that sees all paths as well as bandwidth capabilities and other criteria. Such global knowledge permits repeatable and optimal decisions. This aspect of reliability is keenly desired by organizations that rely on wide-area connections to operate their businesses and institutions. As we discussed in Section 7.4, the SDN controller can have access to all this information, and since it presumably runs on a high-performance server, it can compute optimal paths more quickly and more reliably than the traditional distributed approach.

A number of pre-SDN technologies attempt to enable optimal routing decisions. These include:

- Network monitoring and traffic management applications—for example, sFlow and NetFlow—collect the necessary information so that a network management system can make optimal routing decisions.
- MPLS-TE (see Section 8.2.2) uses traffic engineering mechanisms to choose optimal paths for MPLS *label switched paths* (LSPs) An LSP is a path through an MPLS network. Packets are forwarded along this path by intermediate routers that route the packets by a preassigned mapping of ingress label-port pairs to egress label-port pairs. An LSP is sometimes referred to as an MPLS tunnel.
- The *path computation element* (PCE)-based architecture [13] determines optimal paths between nodes based on a larger set of network-loading conditions than may be considered in simple best-path analysis such as IS-IS.

Even when these pre-SDN technologies attempt to make optimal path decisions based on a global view of the network, they are thwarted by devices that retain their autonomous control planes. We consider an example in the following section.

8.1.2 Example: MPLS LSPs in the Google WAN

In [9] Google describes how its pre-SDN WAN solution handles the case of a failed link. In this particular scenario, the problem revolves around rerouting MPLS LSPs in the event of a link failure.

We depict the pre-SDN situation in Figure 8.2. Say that the link labeled *Best route* on the right in the figure goes down. When a link goes down and the network needs to reconverge, all the LSPs try to reroute. This process is done autonomously by the routers along the path using the *Resource Reservation Protocol* (RSVP) to establish the path and reserve the bandwidth. When the first LSP is established and reserves the remaining bandwidth on a link, the others have to retry over the remaining paths. Because the process is performed autonomously and *ad hoc*, it can be repeated multiple times with a single winner declared each time. This process can iterate for some time until the last LSP is established. It is not deterministic in that it is not known which LSP will be the last in any given scenario. This is a real-life problem, and this pre-OpenFlow solution is clearly lacking.

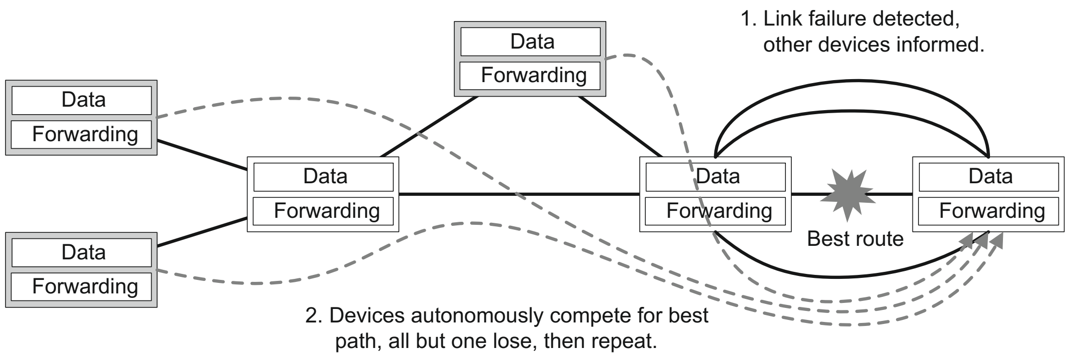

1. Link failure detected, other devices informed.
Data
Forwarding
Best route
2. Devices autonomously compete for best path, all but one lose, then repeat.

FIGURE 8.2

Google without OpenFlow.

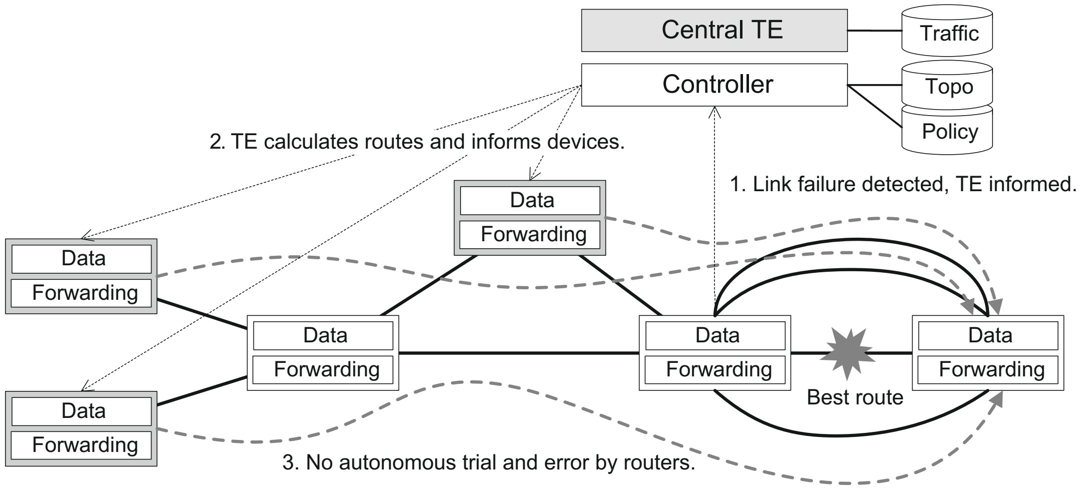

Central TE
Controller
Traffic
Topo
Policy
2. TE calculates routes and informs devices.
1. Link failure detected, TE informed.
Data
Forwarding
Best route
3. No autonomous trial and error by routers.

FIGURE 8.3

Google with OpenFlow.

We contrast this with Figure 8.3, which shows how an SDN system with a centralized controller containing knowledge of all possible routes and current available bandwidth can address this problem much more efficiently. The computation of an optimal solution for mapping the LSPs to the remaining network can be made once and then programmed into the devices. This SDN-based solution is deterministic in that the same result will occur each time.

Google has been at the forefront of SDN technology since its outset, and the company has made use of the technology to its advantage in managing its WAN networks. Google has connected its data centers using OpenFlow switches connected to a controller designed with the features we just described in mind. The values that Google has realized from its conversion to OpenFlow in the WAN are:

- Lower cost of managing devices
- Predictable and deterministic routing decisions in the case of failover
- Optimal routing decisions based on bandwidth and load

8.2 Service Provider and Carrier Networks

Service provider (SP) and carrier networks are wide area in nature and are sometimes referred to as *backhaul* networks, since they aggregate edge networks, carrying huge amounts of data and voice traffic across geographies, often on behalf of telecommunications and Internet service providers (ISPs). Examples of service providers and carriers are Verizon, AT&T, Sprint, Vodafone, and China Mobile.

A *network service provider* (NSP) sells networking bandwidth to subscribers, who are often ISPs. A *carrier* is typically associated with the telecommunications and mobility industries, which in the past have been concerned primarily with voice traffic. Carriers and SPs now carry a much more diverse traffic mix, including *Voice over IP* (VoIP) and video. In addition to a greater variety of traffic types, the total volume of traffic grows at an ever-increasing pace. Smartphones with data plans have played a large role in increasing both the total amount of traffic as well as its diversity.

Without taking a new approach to managing bandwidth, this growth in diversity and volume of traffic results in costs spiraling out of control. Over-provisioning links to accommodate changing traffic patterns is too costly for the WAN links used by the SPs. Traditionally, bandwidth management is provided via network management platforms that can certainly measure traffic utilization and can perhaps even recommend changes that could be made manually in order to respond to traffic patterns.

Not only does network bandwidth need to be utilized most efficiently, it is also necessary to respond immediately to changes in requirements brought about by service contract upgrades and downgrades. If a customer wants to have their traffic travel across the network at a higher priority or at greater speeds, SPs would like to be able to implement changes to their service policies immediately, without disruption to existing flows. Being able to provide *bandwidth on demand* is a selling point for SP. Other requirements include the ability to dynamically change paths to higher-bandwidth, lower latency paths and to reprioritize packets so that they take precedence in queues as they pass through networking devices.

Another aspect of reducing costs involves the *simplification of devices*. In the large core networks operated by carriers, the OPEX costs of managing a complex device outweigh the increased CAPEX outlay for that device. Thus, a simpler device provides cost savings in two ways. This is true as well not only for network devices but also for network appliances that today require specialized hardware, such as load balancers, firewalls, and security systems.

SPs are responsible for taking network traffic from one source, passing it throughout the SP's network, and forwarding it out the remote network edge to the destination. Thus, the packets themselves must cross at least two *boundaries*. When more than one SP must be traversed, the number of boundaries crossed increases. Traffic coming into the service provider's network is typically marked with specific tags for VLAN and priority. The SP has needs regarding routing traffic as well, which may require the packet to be tagged again. Furthermore, the routing mechanism may be different within the SP network. For example, the SP may use MPLS or VLAN tagging for internal routing. Using these additional tagging mechanisms entails another layer of encapsulation of the customer data packet. The boundaries that traffic must cross are often referred to as *customer edge* (CE) and *provider edge* (PE). The *network-to-network interface* (NNI) is the boundary between two SPs. Since the NNI is an important point for policy enforcement, it is important that policy be easily configurable at these boundaries. Technologies supporting SPs in this way must support these boundary-crossing requirements.

Figure 8.4 shows a simplified version of a network with customers and a pair of SPs. The two endpoint hosts are attempting to communicate by passing through the CE and then the PE. After traversing SP1's network, the packet will egress that network at the PE on the other side. After the packet passes through the CE, the destination receives the packet.

The figure shows a packet traversing the network from the endpoint on the left to the endpoint on the right. The original packet as it emanates from the source device is unencapsulated. The packet may acquire a VLAN tag as it passes through the CE (probably provided by the ISP). When the packet passes through the PE, it may be encapsulated using a technology such as PBB that we discussed in Section 5.6.7, or it may be tagged with another VLAN or MPLS tag. When the packet exits the provider network and passes through the other PE on the right, it is correspondingly either decapsulated or the tag is popped and it is then passed into the destination CE network.

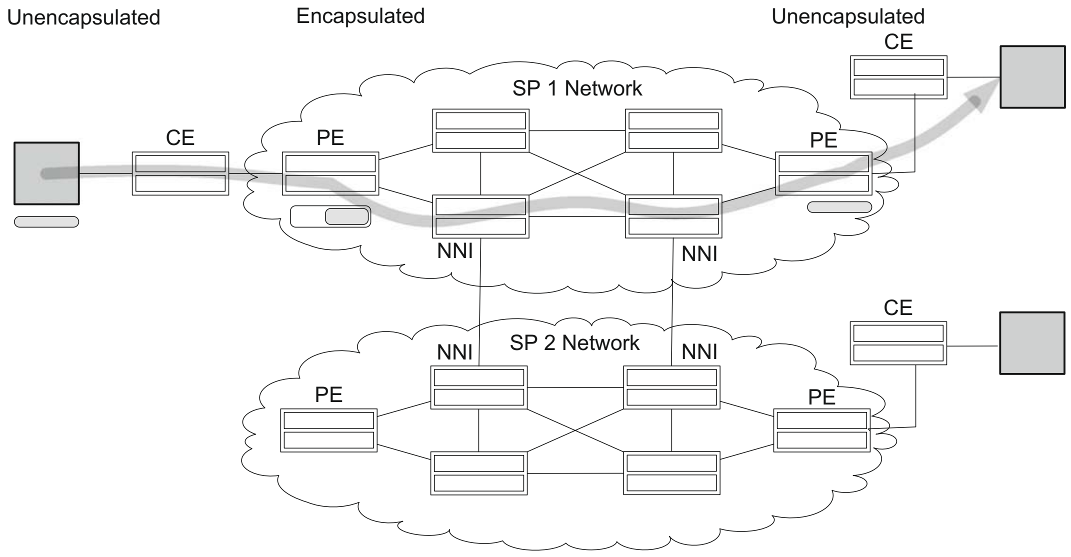


FIGURE 8.4

Service provider environment.

Figure 8.4 additionally depicts the NNI between SP1 and SP2. If the customer packets were directed to a destination on the SP2 network, they would traverse such an NNI boundary. Policies related to business agreements between the SP1 and SP2 service providers would be enforced at that interface.

8.2.1 SDN Applied to SP and Carrier Networks

A key focus of SPs when considering SDN is *monetization*. This refers to the ability to make or save money by using specific techniques and tools. SDN is promoted as a way for providers and carriers to monetize their investments in networking equipment by increasing efficiency, reducing the overhead of management, and rapidly adapting to changes in business policy and relationships.

Some of the ways in which SDN can help improve monetization for providers and carriers are (1) bandwidth management, (2) CAPEX and OPEX savings, and (3) policy enforcement at the PE and NNI boundaries.

SDN exhibits great agility and the ability to maximize the use of existing links using traffic engineering and centralized, network-wide awareness of state. The granular and pliable nature of SDN allows changes to be made easily and with improved ability to do so with minimal service interruption. This facilitates the profitable use of bandwidth as well as the ability to adapt the network to changing requirements related to customer needs and *service-level agreements* (SLAs).

SDN can reduce costs in a couple of ways. First, there are CAPEX savings. The cost of white-box SDN devices is appreciably lower than the cost of comparable non-SDN equipment. This may be due to *bill-of-materials* (BOM) reduction as well as the simple fact that the white-box vendors are accustomed to a lower-margin business model. The BOM reductions derive from savings such as reduced memory and CPU costs due to the removal from the device of so much CPU- and memory-intensive control software. Second, there are reduced OPEX costs, which come in the form of reduced administrative loads related to the management and configuration of the devices.

As described, packets will travel from the customer network across the PE into the provider's network, exiting at the remote PE. OpenFlow supports multiple methods of encapsulating traffic as is required in these circumstances. OpenFlow 1.1 supports the pushing and popping of MPLS and VLAN tags. OpenFlow 1.3 added support for PBB encapsulation. This makes it possible to conform to standard interfaces with Open SDN technology. Additionally, Open SDN provides a versatile mechanism for policy enforcement on the aggregated traffic that traverses NNI boundaries. Since these policies are often tied to fluid business relationships between the SPs, it is essential that the technology supporting policy enforcement be easily manipulated itself. With the additional bandwidth, cost-cutting, visibility, and policy enforcement advantages of SDN, an SDN solution, becomes even more compelling.

8.2.2 Example: MPLS-TE and MPLS VPNs

A research project at Stanford [6] demonstrated *MPLS Traffic Engineering* (MPLS-TE) using OpenFlow. The authors stated that “while the MPLS data plane is fairly simple, the control planes associated with MPLS-TE and MPLS-VPM are complicated. For instance, in a typical traffic engineered MPLS network, one needs to run OSPF, LDP, RSVP-TE, I-BGP and MP-BGP.” They argue that the control plane for a traditional MPLS network that determines routes using traffic information is unnecessarily complex. The amount and complexity of control signaling in the traditional approach is such that when there are frequent changes in the network, there are so many control packets exchanged that some can readily be lost. This results in a cascade of instability in the MPLS-TE control plane. The authors demonstrate that it is simple to build an equivalent network using OpenFlow to push and pop MPLS tags according to the topology and traffic statistics available at the NOX OpenFlow controller. In addition to providing a more stable and predictable alternative to the traditional approach, the OpenFlow-based solution was able to be implemented in less than 2,000 lines of code, which is far more simple than the current systems.

Figure 8.5 shows the architecture of the research project, replacing many routing and traffic engineering protocols on the devices with an MPLS-TE application running on the OpenFlow controller, setting

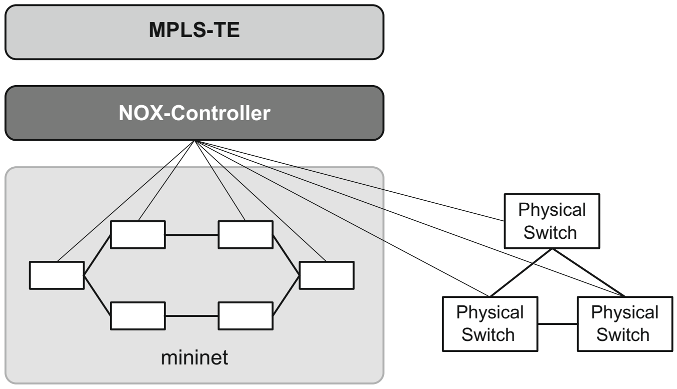


FIGURE 8.5

Service provider and SDN: MPLS (Courtesy [6]).

flows consistent with the connectivity and QoS requirements. This OpenFlow-based MPLS solution obviates the need for a number of other protocols and functions that would otherwise have had to be implemented by the individual devices.

MPLS-TE remains an active area of research for the application of SDN. For example, Ericsson has published related research in using OpenFlow for MPLS-TE in [7].

8.2.3 Example: Cloud Bursting with Service Providers

Cloud bursting allows a service provider to expand an enterprise's private cloud (data center) capacity on demand by dynamically allocating the compute and storage resources in the SP's data center to that enterprise. This additionally entails allocating the network resources to allow the free flow of data between the original private cloud and its dynamic extension in the SP. In [12], the author proposes to accomplish this cloud bursting via an SDN network. Controllers in the enterprise's private cloud make requests to the SP's SDN controller for the needed facilities, and the SP SDN controller allocates the needed networking, compute, and storage components to that enterprise. This model facilitates a major business opportunity for SPs. Verizon, in collaboration with HP and Intel, describes an SDN-based proof-of-concept (PoC) project [10,11] to implement a solution for cloud bursting. In this PoC, an Intel private cloud in Oregon is dynamically augmented by the capacity in a Verizon public cloud in Massachusetts. Without manual intervention, network bandwidth is increased to handle the surge in data between the two data centers. The security that Intel requires on the Verizon VMs can be implemented without the need for physical appliances, using SDN techniques such as those that we discuss in Section 8.6. This PoC demonstrates a method whereby an SP can obtain improved CAPEX *return on investment* (ROI), lower operating expenses, and improved business agility through the use of SDN.

8.3 Campus Networks

Campus networks are a collection of LANs in a concentrated geographical area. Usually the networking equipment and communications links belong to the owner of the campus. This may be a university, a private enterprise, or a government office, among other entities. Campus end users can connect through wireless access points (APs) or through wired links. They can connect using desktop computers, laptop computers, shared computers, or mobile devices such as tablets and smartphones. The devices with which they connect to the network may be owned by their organization or by individuals. Furthermore, those individually owned devices may be running some form of access software from the IT department, or the devices may be completely independent.

There are a number of networking requirements that pertain specifically to campus networks. These include (1) differentiated levels of access, (2) bring your own device (BYOD), (3) access control and security, (4) service discovery, and (5) end-user firewalls.

Various types of users in the campus will require different levels of access. Day guests should have access to a limited set of services, such as the Internet. More permanent guests may obtain access to more services. Employees should receive access based on the category into which they fall, such as executives or IT staff. These differentiated levels of access can be in the form of access control (i.e., what they can and cannot have access to) as well as their quality of service, such as traffic prioritization and bandwidth limits.

BYOD is a phenomenon that has arisen from the exponential increase in functionality available in smartphones and tablet computers. Campus network users want to access the network using the devices with which they are familiar with rather than campus-issued devices. In addition to mobile devices, some individuals may prefer Apple or Linux laptops over the corporate-issued Windows systems.

In years past, access to networks was based on physical proximity. If you were in the building and could plug your device into the network, you were granted access. With the popularity of wireless connectivity and heightened security, it is now more common for employees and guests alike to have to overcome some security hurdle in order to be granted access. That security may be in a more secure form, such as IEEE 802.1X [1], or it may be some more limited form of security, such as authentication based on MAC address or a captive portal web login.

Campus end users want access to services such as printers or file shares or perhaps even TVs. Service discovery makes this possible through simple interfaces and automatic discovery of devices and systems that provide these services. To achieve this level of simplicity, however, there is a cost in terms of network traffic load associated with protocols providing these services.

One of the dangers of campus networks is the possibility of infected devices introducing unwanted threats to the network. This threat is magnified by the presence of BYOD devices on the network. These may take the form of malicious applications such as port scanners, which probe the network looking for vulnerable devices. End-user firewalls are needed to protect against such threats.

8.3.1 SDN on Campus: Application of Policy

At the beginning of this chapter we described the value of centralized policy management and deployment. Campus networks are one area in which this capability is readily realized. SDN's flexibility to manipulate flow rules based on policy definitions makes it well suited to the application of policy in the campus. The general idea of policy application is as follows:

- The user connects to and attempts to send traffic into the network.
- No policy is in place for this user, so the user's initial packets are forwarded to the controller. At this point, the user has either no access or only limited access to the network.
- The controller consults the policy database to determine the appropriate prioritization and access rights for this user.
- The controller downloads the appropriate flow rules for this user.
- The user now has access that is appropriate for the group to which he or she belongs as well as other inputs such as the user's location, time of day, etc.

This system provides the differentiated levels of access that are needed in campus networks.

Clearly, as we explained in Section 3.2.2, this functionality is similar to what *network access control* (NAC) products provide today. However, this SDN-based application can be simple, open, software-based, and flexible in that policies can be expressed in the fine-grained and standard language of the flow. We discuss programming an SDN NAC application in greater detail in Section 10.13.

Note that this type of access control using SDN should be applied at the edge of the network, where the number of users and thus the number of flow entries will not exceed the limits of the actual hardware tables of the edge networking device. We introduced the topic of scaling the number of flow entries in Section 4.3.5. After the edge, the next layer of network concentration is called the *distribution* or *aggregation* layer. Application of policy at this layer of the campus network would take another form. At this level of the network, it is not appropriate or even practical to apply end-user rules that should be applied at the edge, as described above. Rather, SDN policy in these devices should be related to classes of traffic or traffic that has been aggregated in some other way, such as a tunnel or MPLS LSP.

Flows can also be established for various types of traffic that set priorities appropriately for each traffic type. Examples of different traffic types are HTTP versus email. For example, HTTP traffic might be set at a lower priority during office hours, assuming that the organization's work-based traffic was not HTTP-based.

8.3.2 SDN on Campus: Device and User Security

Technological trends such as BYOD, access control, and security can also be addressed by SDN technology. For example, one of the requirements for registering users' BYOD systems and guests involves the use of a *captive portal*. This is the mechanism by which a user's browser request gets redirected to another destination website. This other website can be for the purpose of device registration or guest access. Captive portals are traditionally implemented by encoding the redirection logic into the switch firmware or by in-line appliances. To implement this in a pre-SDN environment could entail upgrading to switches that had this capability, implying more complex devices, or by installing specialized in-line appliances, adding to network complexity. Configuring which users need to be authenticated via the captive portal would entail configuring all these devices where a user could enter the network. Here we describe a simpler SDN solution for a captive portal:

- The user connects to the network and attempts to establish network connectivity.
- No access rules are in place for this user, so the SDN controller is notified.
- The SDN controller programs flows in the edge device, which will cause the user's HTTP traffic to be redirected to a captive portal.
- The user is redirected to the captive portal and engages in the appropriate exchange to gain access to the network.
- Once the captive portal exchange is complete, the SDN controller is notified to set up the user's access rules appropriately.
- The user and/or BYOD device now has the appropriate level of access.

Figure 8.6 illustrates an SDN-based captive portal-based application. The network edge device is initially programmed to route ARP, DNS, and DHCP requests to the appropriate server. In the figure, the end user connects to the network and makes a DHCP request to obtain an IP address. When the DHCP reply is returned to the user, a copy is sent to the SDN controller. Using the end-user MAC address as a lookup key, the controller consults the database of users. If the user device is currently registered, it is allowed into the network. If it is not, then OpenFlow rules are programmed to forward that user's HTTP traffic to the controller. When the unauthenticated user's HTTP traffic is received at the controller, that web session is redirected to the captive portal web server. After completing the user authentication or device registration, the controller updates the user's flow(s) so that the packets will be allowed into the network. Note that this is also the appropriate time to configure rules related to the user's policy, which can include levels of access and priority, among other items.

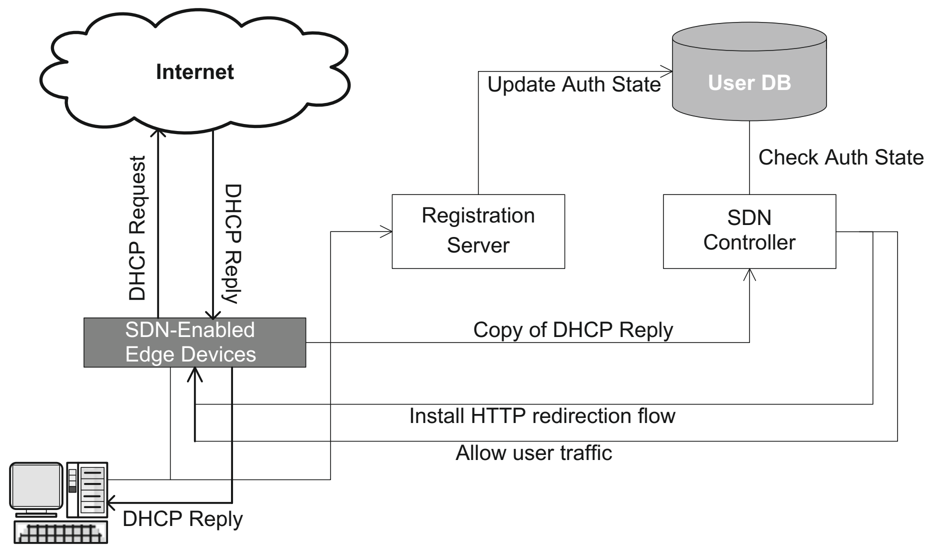


FIGURE 8.6

NAC captive portal application.

This type of functionality, like policy, is appropriately applied at the edge of the network. The further into the network this type of application is attempted, the more problematic it becomes, with flow table overflow becoming a real issue.

HP currently offers a commercial SDN application called Sentinel [8] that provides secure BYOD access. The approach of securing the network from end-user infection by installing anti-malware software on end-user devices is not viable in the BYOD case since they are not under the control of the network administration. The Sentinel SDN application provides a solution by turning the entire network infrastructure into a security-enforcement device using mechanisms that are hardened and more sophisticated versions of the basic methods for campus security we described above. Sentinel is an SDN application that works with HP's TippingPoint threat database to secure and apply policy to the intranet as follows:

1. Identify *botnets* on the intranet and neutralize them by blocking DNS requests and IP connections to their *botnet masters* in the Internet.
2. Identify and prevent users or machines from accessing infected or prohibited sites on the Internet by monitoring and blocking DNS requests and IP connections (see Section 8.3.3).
3. Identify and quarantine infected machines on the intranet so that they can be remediated.

While traditional NAC solutions provide similar functionality, as an SDN application Sentinel can be deployed without installing additional hardware, and it can provide protection at the edge of the network. Sentinel is a sophisticated example of the *blacklist* technology discussed in the next section.

8.3.3 SDN on Campus: Traffic Suppression

Having flow table rules at the edge of the network carries the additional benefit of being able to suppress unwanted traffic. That unwanted traffic could be benign, such as service discovery multicasts, or it could be malicious, such as infected devices bringing viruses into the network. With flow tables at the edge of the network, service discovery traffic can be captured by the edge device and forwarded to the controller. The controller can then keep a repository of services and service requestors, which can be used to satisfy requests without having to forward all those multicasts upstream to flood the possibly overloaded network.

The ability to set policy and security for end users with highly granular flow-based mechanisms facilitates effective per-user firewalls. Only certain types of traffic (e.g., traffic destined for certain UDP or TCP ports) will be allowed to pass the edge device. All other types of traffic will be dropped at the very edge of the network. This is an effective means of blocking certain classes of malicious traffic. Note that the implementation of per-user firewalls requires that the controller know who the user is. If a NAC solution is running in the SDN controller, this is not a problem. If the NAC solution, such as RADIUS, is implemented outside of SDN, there is a challenge because the relationship between that external NAC solution and OpenFlow is not standardized nor well understood.

One facet of a per-user firewall is an application known as a *blacklist*. Blacklist technology blocks attempts by end users to access known malicious or harmful hostnames and IP addresses. Traditional blacklist solutions rely on in-line appliances that *snoop* all traffic and attempt to trap and block attempts to reach these bad destinations. SDN is able to implement a blacklist solution without the insertion of additional appliances into the network. This provides both CAPEX savings due to less network equipment and OPEX savings due to a network that is easier to administer. An SDN application for blocking attempts to reach specific hostnames entails setting up flow table rules in edge devices to capture DNS requests and send them to the controller. The SDN controller consults a database of undesirable hostnames and blocks the DNS request from being sent out when a blacklisted hostname is encountered.

Figure 8.7 shows the simple manner in which a DNS blacklist could be implemented. The controller sets flows in the edge devices, directing them to forward all DNS requests to the controller. When DNS requests arrive at the controller, the controller consults a local or remote database of known malicious sites. If the result is that the hostname is clean, the controller returns the DNS request with instructions to the edge device to forward the packet as it normally would. If the hostname is deemed unsafe, the controller instructs the edge device to drop the packet, denying the user access to that host.

A savvy user can circumvent this first blacklist application if he/she knows the IP address of the destination host. By specifying the IP address directly instead of the hostname, no DNS request is sent. Figure 8.8 shows a simple SDN implementation for this variant of the blacklist problem. In this second solution, packets with destination IP addresses of unknown virtue are forwarded to the controller, which inspects the packets and either allows the IP address (so that packets to and from that IP address will now automatically be allowed) or else the device is programmed to drop this and future packets sent to that bad address. If the packet is allowed, the application will install a transient flow allowing requests to that destination IP address.

Note with this latter solution, application design is critical. If destination IP address *allow* rules have aging timers that are too short, they will age out in between requests, and consequently, every time a destination is visited, there will be delay and overhead associated with verifying and allowing the

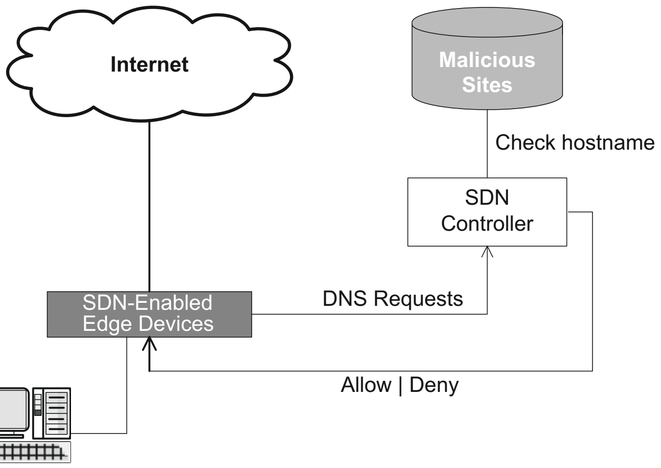


FIGURE 8.7

DNS blacklist application.

packets to that destination. If those rules have aging timers that are too long, the device runs the risk of being overloaded with flow entries, possibly even exceeding the maximum size of the flow table.

We provide a detailed analysis with accompanying source code for an Open SDN blacklist application in Section 10.4.

Blacklist is really a simple host-level firewall. HP's Sentinel application, described earlier, provides a blacklist feature as a component of its commercial SDN security application.

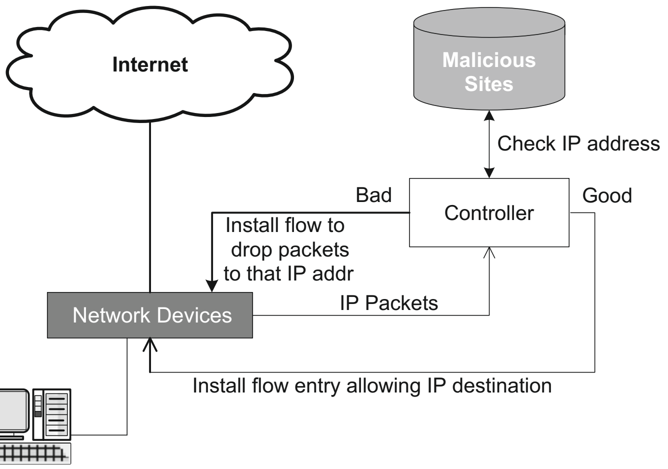

Internet
Malicious Sites
Check IP address
Bad
Controller
Good
Install flow to drop packets to that IP addr
Network Devices
IP Packets
Install flow entry allowing IP destination

FIGURE 8.8

IP address blacklist application.

8.4 Hospitality Networks

Hospitality networks are found in hotels, airports, coffee shops, and fast-food restaurants among other places. There is a fair degree of overlap between the user requirements in campus networks and those of hospitality networks. The primary end user in a hospitality network is a guest who is using the network either through the largesse of the establishment or through the purchase of a certain amount of time. One class of the for-purchase situation is end users who have purchased connectivity for a certain duration, such as a month or a year, and who are thus able to connect to the network without a direct financial exchange each time they connect.

The application of SDN for captive portals in the campus discussed in Section 8.3.2 applies similarly to hospitality networks. The difference is that in the case of hospitality networks, the user is paying directly or indirectly for the access, so a corresponding form of identification will likely be involved in authenticating with the registration server. These could include providing hotel room or credit card information.

Hospitality networks frequently offer WiFi access. We describe a more general application of SDN to WiFi access in the next section on mobile networks.

8.5 Mobile Networks

Mobile networking vendors, such as AT&T, Verizon, and Sprint, compete for customers to attach to their networks. The customers use smartphones or tablets to connect using the available cellular service, whether that is 3G, 4G, LTE, or another cellular technology.

When mobile customers use traditional WiFi hotspots to connect to the Internet, those mobile vendors effectively lose control of their customers. This is because the users' traffic enters the Internet directly from the hotspot. Since this completely circumvents the mobile vendor's network, the vendor is not even aware of the volume of traffic that the user sends and receives and certainly cannot enforce any policy on that connection. When customer traffic circumvents the mobile provider's network, the provider loses a revenue-generating opportunity. Nevertheless, since their cellular capacities are continually being stretched, from that perspective it is advantageous for the mobile vendors to offload traffic to WiFi networks when possible. Thus, the mobile provider is interested in a solution that allows its customers to access its networks via public WiFi hotspots *without the provider losing control of and visibility to its customers' traffic*. The owner of such hotspots may want for multiple vendors to share the WiFi resource offered by the hotspot. The multitenant hotspot we describe here is somewhat analogous to network virtualization in the data center. Just as SDN shines in that data center environment, so can it play a pivotal role in implementing such multitenant hotspots.

8.5.1 SDN Applied to Mobile Networks

Mobile vendors interested in gaining access to users who are attaching to the Internet via WiFi hotspots require a mechanism to control their users' traffic. Control in this context may simply mean being able to measure how much traffic that user generates. It may mean the application of some policy regarding

QoS. It may mean diverting the user traffic before it enters the public Internet and redirecting that traffic through its own network. SDN technology can play a role in such a scheme in the following ways:

- Captive portals
- Tunneling back to the mobile network
- Application of policy

We discussed captive portals and access control in Section 8.3.1. This functionality can be applied to mobile networks as well. It requires allowing users to register for access based on their mobile credentials. Once valid credentials are processed, the user is granted appropriate levels of access.

One of the mechanisms for capturing and managing mobile user traffic is through the establishment of tunnels from the user's location back to the mobile vendor's network. The tunnel would be established using one of several available tunneling mechanisms. By programming SDN flows appropriately, that user's traffic would be forwarded into a tunnel and diverted to the mobile vendor's network. Usage charges could be applied by the mobile provider. In addition to charging for this traffic, other user-specific policies could be enforced. Such policies could be applied at WiFi hotspots where the user attaches to the network. SDN-enabled access points can receive policy, either from the controller of the mobile vendor or from a controller on the premises.

As an example of the utilization of SDN and OpenFlow technology as it relates to the needs of mobile networks and their service providers, consider Figure 8.9.

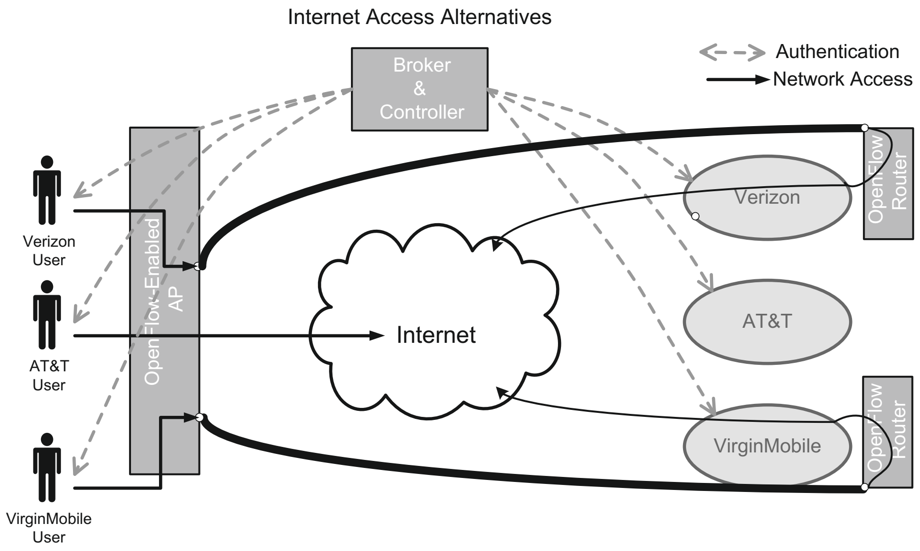


FIGURE 8.9

Mobile service providers.

The example depicted in Figure 8.9 shows the basic means by which SDN and OpenFlow can be used to grant carrier-specific access and private or public access from mobile devices to the Internet. In Figure 8.9 the customers on the left want to access the Internet, and each set has a different carrier through which they gain network access. Connecting through an OpenFlow-enabled wireless hotspot, they are directed through a broker that acts as an OpenFlow controller. Based on their carrier, they are directed to the Internet in various ways, depending on the level of service and the mechanism set up by the carrier. In the example, AT&T users gain access directly to the Internet, whereas Verizon and Virgin Mobile users access the Internet through being directed by OpenFlow through a tunnel to the carrier's network. Both tunnels start in the OpenFlow-enabled AP. One tunnel ends in an OpenFlow-enabled router belonging to Verizon, the other in an OpenFlow-enabled router belonging to Virgin Mobile. These two routers then redirect their respective customers' traffic back into the public Internet. In so doing, the customers gain the Internet access they desire, and two of the mobile providers achieve the WiFi offload they need while maintaining visibility and control over their users' traffic. To facilitate such a system, there is presumably a business relationship between the providers and the hotspot owner whereby the hotspot owner is compensated for allowing the three providers' users to access the hotspot. There would likely be a higher fee charged Verizon and Virgin Mobile for the service that allows them to retain control of their customers' traffic.

The ONF's Wireless and Mobile Working Group has published a number of OpenFlow-based use cases. These include:

- Flexible and scalable packet core
- Dynamic resource management for wireless backhaul
- Mobile traffic management
- Management of secured flows in LTE
- Media-independent handover
- Energy efficiency in mobile backhaul networks
- Security and backhaul optimization
- Unified equipment management and control
- Network-based mobility management
- SDN-based mobility management in LTE
- Unified access network for enterprise and large campus

These use cases are very detailed and specific applications relevant to mobile operators. In a number of cases, implementing them would require extensions to OpenFlow. The mobile use cases listed here as well as others are described in [5].

8.6 In-Line Network Functions

In-line network functions typically include functionality such as load balancers, firewalls, *intrusion detection systems* (IDS), and *intrusion prevention systems* (IPS). These services must be able to inspect the packets passing through them. These functions are often delivered in specialized appliances that can be very expensive. These appliances are *bumps in the wire*, meaning that they are inserted into the data plane. In some cases, the *bump* shunts traffic to an out-of-band processor, which can result in cost savings. In other cases, the value-added service performed by the appliance is performed at line rate, and the increased cost of such devices is necessary if performance is of paramount concern. Data centers and SPs that must employ these types of services welcome any novel means to drive down the costs associated with these devices.

There are many schemes for load balancing traffic, but all involve some level of packet inspection, choosing the destination in order to evenly distribute the load across the entire set of servers.

Firewalls play the role of admitting or denying incoming packets. The decision criteria involve packet inspection and can be based on factors such as destination or source IP address, destination or source TCP or UDP port, or something deeper into the payload, such as the destination URL.

IDS and IPS solutions analyze packets for malicious content. This entails performing process-intensive analysis of the packet contents to determine whether a threat is present. Although an IPS is a bump in the wire, an IDS can run in parallel with the data plane by looking at mirrored traffic. Such systems require some traffic-mirroring mechanism. A hybrid system can be implemented in the SDN paradigm by shunting only those packets requiring deeper inspection to an appliance that is off the data path. This approach has the advantage of removing the bump in the wire, since packets that are not subject to deeper inspection suffer no additional latency at all. Furthermore, since not all packets need to be processed by the appliance, fewer or less powerful appliances are needed in the network. This can be realized in SDN by simply programming flow rules, which is much simpler than the special *Switch Port ANalyzer* (SPAN) ports that are in use on contemporary traditional switches. We discuss SPAN ports in Section 8.6.4.

8.6.1 NFV vs. SDN

SDN has been proposed to address network service needs through *network functions virtualization* (NFV), introduced in Section 6.5. The idea of NFV is to move service functionality (load balancers, firewalls, etc.) off specialized appliances and implement it as software that runs on common server platforms. This has been made possible by advances in server and network interface technology. Applications that once required specialized hardware for both CPU and NICs can now be performed by industry-standard servers. Arguments in favor of this shift include the lower cost of servers compared to these specialized appliances as well as the ability to easily upgrade the hardware when necessary.

Active standards efforts are under way for NFV. The *European Telecommunications Standards Institute* (ETSI) has the *Industry Specification Group* (ISG) working on NFV standardization, with 127 companies participating. There is also the CloudNFV industry consortium led by the CIMI Corporation that is building NFV prototypes. A number of NEMs are bringing NFV products to the market. These include Juniper, Cisco, Intel, VMware, and the startup Embrane.

NFV is a logical next step in the migration toward the server virtualization in data centers that we have discussed previously. Part of the motivation is to lower equipment costs and power consumption by reducing the amount of physical equipment. This reduction in equipment results in a more simplified network topology, which is easier to manage and permits faster rollout of new services. New services can be enabled by merely loading and configuring software. The fact that these services are software-enabled means that it is easier to maintain test beds for services being evaluated and trial rollouts alongside the operational services.

Migrating to NFV is not without its challenges. Performance is a key concern. Different functions have very different performance requirements, and some functions need to be performed at line rate on specialized hardware. Extra effort is necessary to ensure that the NFV implementations work well on the many different servers and hypervisors that come from different vendors. Because the migration to NFV will take place over many years, and indeed there will always be the need for some specialized hardware appliances, it will be necessary to manage the coexistence of NFV solutions with traditional appliances. Architecting for resiliency to hardware and software failures in an NFV world will be very different than with traditional networking. Finally, traffic flows must be directed to these virtual appliances. The way this is done will depend on where in the network the NFV functionality is hosted. There are three basic methods:

- Standard servers running NFV functionality in the same topological network location as the appliance. This saves on hardware costs but does not reduce network complexity.
- Much of the network functionality, including switches and appliances, can be virtualized in a hypervisor. This is the approach taken by Nicira and others. This may be appropriate in data center environments.
- OpenFlow-capable switches can selectively shunt traffic to standard servers running NFV functionality off the main data path. This is the area of overlap between SDN and NFV of greatest interest in this book. We provide an example of SDN working with NFV in this way in Section 8.6.4.

NFV is an extensive field that merits deeper treatment than the scope of this book allows. We refer the interested reader to [2] for an excellent introduction to this topic. To be clear, NFV is not the same as SDN, although there is some overlap. Running network services on standard servers is not the same as moving control planes off networking devices, having a controller, and opening the entire networking domain to innovation.

8.6.2 SDN Applied to Server Load Balancing

Load balancers must take incoming packets and forward them to the appropriate servers. Using SDN and OpenFlow technology, it is straightforward to imagine a switch or other networking device having rules in place that would cause it to act as a load-balancer appliance at a fraction of the cost of purchasing additional, specialized hardware. Figure 8.10 shows a switch acting as a load balancer distributing load across a set of servers.

OpenFlow 1.0 supports matching against the basic 12-tuple of input fields, as described in Section 5.3.3. A load balancer has a rich set of input options for determining how to forward traffic. For example, as shown in the figure, the forwarding decision could be based on the source IP address. This would afford server continuity for the duration of time that that particular endpoint was attempting to communicate with the service behind the load balancer. This would be necessary if the user's transaction with the service required multiple packet exchanges and if the server needed to maintain state across those transactions.

The fact that the controller will have the benefit of understanding the load on each of the links to the servers, as well as information about the load on the various servers behind the firewall, opens up other possibilities. The controller could make load-balancing decisions based on a broader set of criteria than would be possible for a single network appliance.

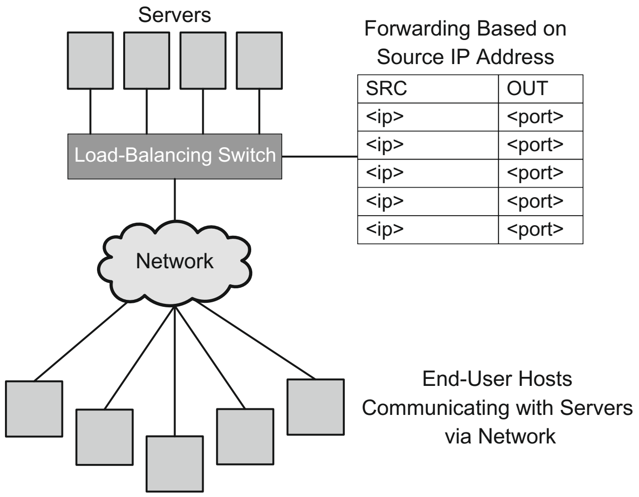


FIGURE 8.10

Load balancers using OpenFlow.

8.6.3 SDN Applied to Firewalls

Firewalls take incoming packets and forward them, drop them, or take some other action such as forwarding the packet to another destination or to an IDS for further analysis. Like load balancers, firewalls using SDN and OpenFlow have the ability to use any of the standard 12 match fields for making these forward or drop decisions. Figure 8.11 shows a switch behaving as an inexpensive firewall. Packets matching certain protocol types (e.g., HTTPS, HTTP, Microsoft Exchange) are forwarded to the destination, and those that do not match are dropped. Simple firewalls that are ideally suited to an SDN-based solution include firewalls that are based on blocking or allowing specific IP addresses or specific TCP/UDP ports. As with load balancers, SDN-based firewalls may be limited by a lack of statefulness and inability for deeper packet inspection.

One SDN use case applied to firewalls is to insert an OpenFlow-enabled switch in front of a battery of firewalls. One benefit is that the SDN switch can load balance between the firewall devices, permitting greater scale of firewall processing. More important, the SDN switch can shunt *known-safe* traffic around the firewalls. This is possible because OpenFlow rules can be programmed such that we offload from the firewalls traffic that is known to be problem-free. The benefit of doing so is that less firewall equipment and/or power is needed and less traffic is subjected to the latency imposed by firewall processing.

8.6.4 SDN Applied to Intrusion Detection

An IDS is intended to observe network traffic to detect network intrusions. An IPS additionally attempts to prevent such intrusions. The range of intrusions that these systems are intended to recognize is very broad, and we do not attempt to enumerate them here. We do, however, consider two particular IDS functions that lend themselves well to SDN-based solutions.

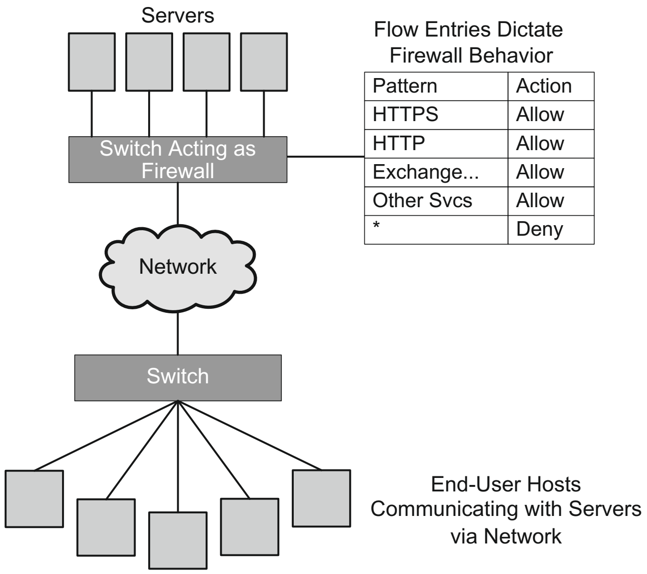

Servers
Flow Entries Dictate Firewall Behavior
Pattern | Action
HTTPS | Allow
HTTP | Allow
Exchange... | Allow
Other Svcs | Allow
* | Deny
Switch Acting as Firewall
Network
Switch
End-User Hosts Communicating with Servers via Network

FIGURE 8.11

Firewalls using OpenFlow.

The first type of intrusion detection we consider requires deep inspection of packet payloads, searching for signs of viruses, trojans, worms, and other malware. Since this inspection is processing-intensive, it must take place somewhere outside the data plane of the network. Typical solutions utilize a *network tap*, which is configured to capture traffic from a particular source, such as a VLAN or physical port, and copy that traffic and forward it to an IDS for analysis. Physical taps such as this are mostly port-based.

One pre-SDN solution is based on a network patch-panel device that can be configured to take traffic from an arbitrary device port and forward it to the analysis system. This requires patch panel technology capable of copying traffic from any relevant network port. This technology is available from companies such as *Gigamon* and *cPacket Networks*. In large networks the cost of such approaches may be prohibitive. Forwarding traffic on uplinks is also sensitive to the volume of traffic, which may exceed the capacity of the uplink data path.

Another pre-SDN solution for IDS is to use configuration tools on the network devices to configure SPAN ports. This allows the switch to operate as a virtual tap, where it copies and forwards traffic to the IDS for analysis. Such taps can be based on port or VLAN or even some other packet-based filter, such as device address.

OpenFlow is well suited to this application, since it is founded on the idea of creating flow entries that match certain criteria (e.g., device address, VLAN, ingress port) and performing some action. A tap system using OpenFlow can be programmed to set up actions on the device to forward packets of interest either out through a monitor port or to the IP address of an IDS.

Note that the use case cited at the end of Section 8.6.3, where an SDN switch is used to front-end a battery of firewalls, can also be applied to a battery of IPS appliances. The same two rationales of shunting safe traffic around the IPS appliances while allowing the system to scale by using multiple IPS units in parallel apply in this case as well.

The FlowScale open source project done at InCNTRE is an example of this kind of front-end load balancing with the SPAN concept. FlowScale sends *mirrored* traffic to a battery of IDS hosts, load balancing the traffic across them.

8.7 Optical Networks

An *optical transport network* (OTN) is an interconnection of optical switches and optical fiber links. The optical switches are layer one devices. They transmit bits using various encoding and multiplexing techniques. The fact that such optical networks transmit data over a lightwave-based *channel* as opposed to treating each packet as an individually routeable entity lends itself naturally to the SDN concept of a flow. In the past, data traffic was transported over optical fiber using protocols such as *Synchronous Optical Networking* (SONET) and *Synchronous Digital Hierarchy* (SDH). More recently, however, OTN has become a replacement for those technologies. Some companies involved with both OTN and SDN are Ciena, Cyan, and Infinera. Some vendors are creating optical devices tailored for use in data centers. Calient, for example, uses optical technology for fast links between racks of servers.

As was shown in Table 3.2, the ONF has a working group dedicated to studying use cases for SDN applications for optical transport, underscoring the importance of this field of application for SDN. Next we examine a simple example of such a use case.

8.7.1 SDN Applied to Optical Networks

In data center networks, there often arise certain traffic flows that make intense use of network bandwidth, sometimes to the point of starving other traffic flows. These are often called *elephant flows* due to their sizable nature. The flows are characterized by being of relatively long duration yet having a discrete beginning and end. They may occur due to bulk data transfer, such as backups that happen between the same two endpoints at regular intervals. These characteristics can make it possible to predict or schedule these flows. Once detected, the goal is to reroute that traffic onto some type of equipment, such as an all-optical network that is provisioned specifically for such large data offloads. OTNs are tailor-made for these huge volumes of packets traveling from one endpoint to another. Packet switches' ability to route such elephant flows at packet-level granularity is of no benefit, yet the burden an elephant flow places on the packet-switching network's links is intense. Combining a packet-switching network with an OTN into the kind of *hybrid* network shown in Figure 8.12 provides an effective mechanism for handling elephant flows.

In Figure 8.12 we depict normal endpoints ($A1$, $A2$) connected through *top-of-rack* (ToR) switches $ToR-1$ and $ToR-2$, communicating through the normal path, which traverses the packet-based network fabric. The other elephant devices ($B1$, $B2$) are transferring huge amounts of data from one to the other; hence, they have been shunted over to the optical circuit switch, thus protecting the bulk of the users from such a large consumer of bandwidth.

The mechanism for this shunting or *offload* entails the following steps:

1. The elephant flow is detected between endpoints in the network. Note that, depending on the flow, detecting the presence of an elephant flow is itself a difficult problem. Simply observing a sudden

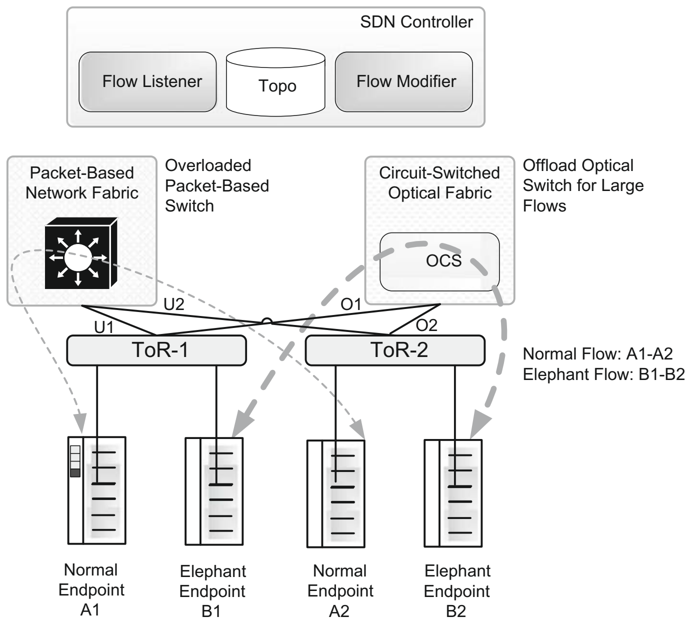


FIGURE 8.12

Optical offload application overview.

surge in the data flow between two endpoints in no way serves to predict the longevity of that flow. If the flow is going to end in 500 ms, this is not an elephant flow and we would not want to incur any additional overhead to set up special processing for it. This is not trivial to know or predict. Normally, some additional contextual information is required to know that an elephant flow has begun. An obvious example is the case of a regularly scheduled backup that occurs across the network. This topic is beyond the scope of this book; we direct the interested reader to [3,4].

2. The information regarding the endpoints' attaching network devices is noted, including the uplinks ($U1$, $U2$), which pass traffic from the endpoints up into the overloaded network core.
3. The SDN controller program flows in $ToR-1$ and $ToR-2$ to forward traffic to and from the endpoints ($B1$, $B2$) out an appropriate offload port ($O1$, $O2$) rather than the normal port ($U1$, $U2$). Those offload ports are connected to the optical offload fabric.
4. The SDN controller programs flows on the SDN-enabled optical offload fabric to patch traffic between $B1$ and $B2$ on the two offload links ($O1$, $O2$). At this point, the rerouting is complete and the offload path has been established between the two endpoints through the OTN.
5. The elephant flow eventually returns to normal and the offload path is removed from both connecting network devices and from the optical offload device. Subsequent packets from $B1$ to $B2$ traverse the packet-based network fabric.

We discuss details of programming an offload application in Section 10.12.

8.8 SDN vs. P2P/Overlay Networks

At a conceptual level, *P2P/overlay networks* resemble the overlay networks presented in detail throughout this book. Just as the data center virtual networks are overlaid on a physical infrastructure, the details of which are masked from the virtual network, so also is the P2P/overlay network overlaid over the public Internet without concern or knowledge of the underlying network topology. Such networks comprise a usually ad hoc collection of host computers in diverse locations owned and operated by separate entities, with each host connected to the Internet in either permanent or temporary fashion. The peer-to-peer (hence the name P2P) connections between these hosts are usually TCP connections. Thus, all of the hosts in the network are directly connected. Napster is the earliest well-known example of a PTP/overlay network.

We introduce P2P/overlay networks here primarily to distinguish them from the overlay networks we describe in SDN via hypervisor-based overlays. Although the nature of the overlay itself is different, it is interesting to consider where there might be some overlap between the two technologies. Just as scaling SDN will ultimately require coordination of controllers across controlled environments, there is a need for coordination between P2P/overlay devices. SDN helps move up the abstraction of network control, but there will never be a single controller for the entire universe, and thus there will still need to be coordination between controllers and controlled environments. P2P/overlay peers also must coordinate among each other, but they do so in a topology-independent way by creating an overlay network. A big distinction is that Open SDN can also control the underlay network. The only real parallel between these two technologies is that at some scaling point, they must coordinate and control in a distributed fashion. The layers at which this is applied are totally different, however.

8.9 Conclusion

Through examples, we have illustrated in this and the preceding chapter that SDN can be applied to a very wide range of networking problems. We remind the reader of the points made in Section 4.1.3: that SDN provides a high-level abstraction for detailed and explicit programming of network behavior. It is this ease of programmability that allows SDN to address these varied use cases. One of the most exciting aspects of SDN is that the very flexibility that has rendered it capable of crisply addressing traditional network problems will also make it adaptable to solve yet-to-be conceived networking challenges. Addressing the different use cases that we have described in these two chapters requires a wide variety of controller applications. In Chapter 10 we examine some sample SDN applications in detail and see how domain-specific problems are addressed within the general network programming paradigm of SDN. First, though, we should acknowledge that this important movement could only take place because of key players that have pushed it forward. For that reason, in Chapter 9 we take a step back from technology and review the individuals, institutions, and enterprises that have had the greatest influence on SDN since its inception.

References

[1] IEEE standard for local and metropolitan area networks: port-based network access control. IEEE 802.1X-2010. New York, NY, USA: IEEE; February 2010.

[2] Chiosi M, Clarke D, Feger J, Chi C, Michel U, Fukui M, et al. Network functions virtualisation: an introduction, benefits, enablers, challenges and call for action. SDN and OpenFlow world congress, Darmstadt, Germany; October 2012.
[3] Platenkamp R. Early identification of elephant flows in internet traffic. In: Sixth twente student conference on IT, University of Twente; February 2007. Retrieved from <referaat.cs.utwente.nl/conference/6/paper/6797/early-identification-of-elephant-flows-in-internet-traffic.pdf>.
[4] Rivillo J, Hernandez J, Phillips I. On the efficient detection of elephant flows in aggregated network traffic. Networks and control group, Research school of informatics, Loughborough University. Retrieved from <www.ee.ucl.ac.uk/lcs/previous/LCS2005/49.pdf>.
[5] Wireless and mobile working group charter application: use cases. Open Networking Foundation; October 2013.
[6] Sharafat A, Das S, Parulkar G, McKeown N. MPLS-TE and MPLS VPNs with OpenFlow. In: SIGCOMM'11, Toronto; August 2011.
[7] Green H, Kempf J, Thorelli S, Takacs A. MPLS openflow and the split router architecture: a research approach. Ericsson Research, MPLS 2010, Washington, DC, USA; 2010.
[8] Case Study: Ballarat Grammar Secures BYOD with HP Sentinel SDN. Hewlett-Packard case study. Retrieved from <h20195.www2.hp.com/v2/GetPDF.aspx/4AA4-7496ENW.pdf>.
[9] Hoelzle U. OpenFlow@Google. Google, Open Networking Summit, Santa Clara, CA, USA; April 2012.
[10] Verizon to demonstrate software defined networking principles with collaborative lab trials. Verizon Press Release; April 17, 2012. Retrieved from <newscenter2.verizon.com/press-releases/verizon/2012/verizon-to-demonstrate.html>.
[11] Schooler R, Sen P. Transforming networks with NFV & SDN. Intel/Verizon, Open Networking Summit, Santa Clara, CA, USA; April 2013. p. 24–9.
[12] McDysan D. Cloud-bursting use case. Internet Engineering Task Force; October 24, 2011 [internet draft].
[13] Farrel A, Vasseur JP, Ash J. A Path Computation Element (PCE)-based architecture, RFC 4655. Internet Engineering Task Force; August 2006.

CHAPTER

Players in the SDN Ecosystem 9

A technological shift as radical and ambitious as SDN does not happen without many players contributing to its evolution. Beyond the luminaries behind the early concepts of SDN are many organizations that continue to influence the direction in which SDN is headed and its rate of growth. These players' impact is sometimes positive and at other times contrary to the growth of SDN, but they are in any case influential in SDN's evolution. We classify the various organizations that have had the most influence on SDN into the following categories:

- Academic researchers
- Industry research labs
- Network equipment manufacturers
- Software vendors
- Merchant silicon vendors
- Original device manufacturers
- Enterprises
- Standards bodies and industry alliances

Figure 9.1 depicts these major groupings of SDN players and the synergies and conflicts that exist between them. We also show three super-categories in the figure. Academic and industry-based basic research, standards bodies and industry alliances, and the *white-box* ecosystem make up these super-categories. In the context of this book, a white-box switch is a hardware platform that is purpose-built to easily incorporate the OpenFlow device software that would come in the form of an OpenFlow implementation such as Switch Light or OVS. In this chapter we discuss each of these categories, how they have contributed to the evolution of SDN, and the various forces that draw them together or place them into conflict.

9.1 Academic Research Institutions

Universities worldwide have engaged in research related to SDN. Many academic institutions have contributed to the advancement of SDN; we mention here three institutions of particular interest. They are Stanford University, U.C. Berkeley, and Indiana University. Stanford and Berkeley have been at the forefront of SDN since its very inception and have employed an impressive number of the luminaries who created this new model of networking.

As we saw in Section 3.2.6, events at Stanford led directly to the creation of the *Clean Slate* [1] program, including forwarding and control plane separation and, notably, the OpenFlow protocol. The *Open Networking Research Center* (ONRC) began as a joint project of Stanford and U.C. Berkeley [6],

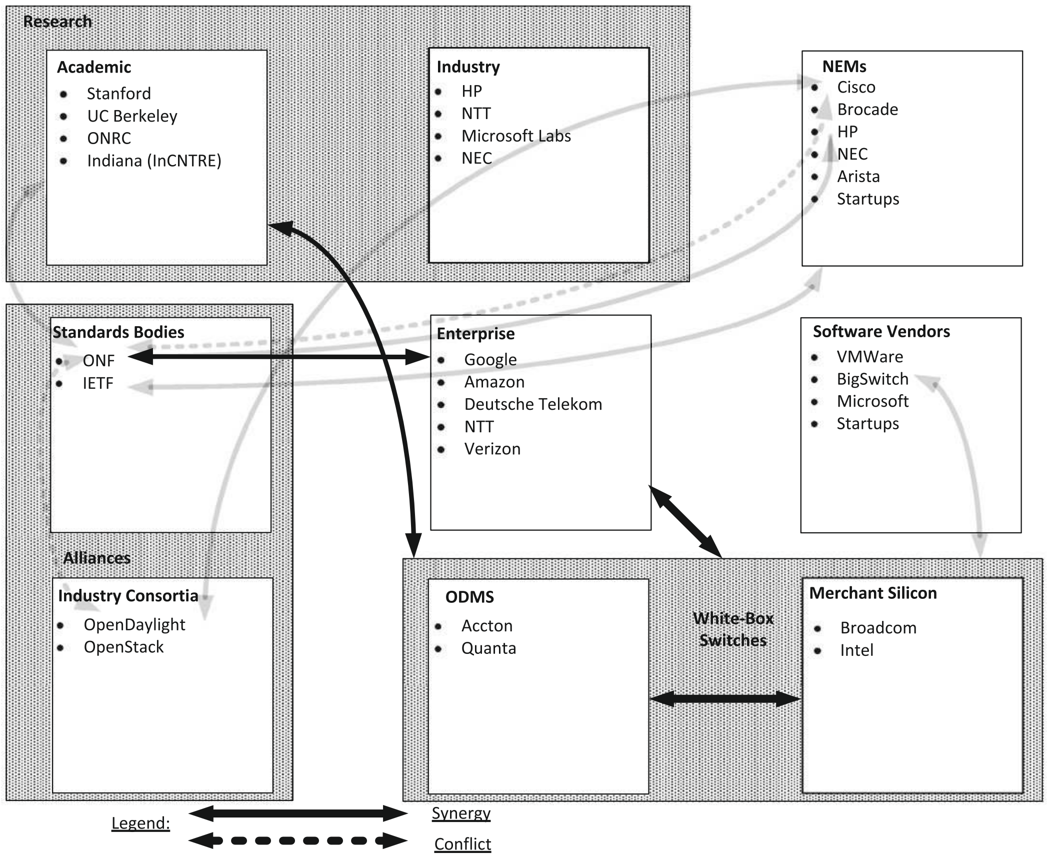


FIGURE 9.1

SDN players ecosystem.

receiving funding from a number of major vendors such as Cisco, Intel, Google, VMware, and others. The ONRC's stated intention is to *open up the Internet infrastructure for innovations*. The ONRC has produced a number of research projects [7]. An affiliated organization called ON.LAB seeks to develop, deploy, and support various SDN and OpenFlow-related tools and platforms. The ONRC is more research oriented, whereas ON.LAB is more focused on the practical application of SDN.

Indiana University has been instrumental in promoting SDN and OpenFlow with a number of research projects and implementations, including the *Indiana Center for Network Translational Research and Education* (InCNTRE) [8]. Its goal is to advance development, increase knowledge, and encourage adoption of OpenFlow and other standards-based SDN technologies. One of the most notable activities at InCNTRE is a plugfest hosted by the *Open Networking Foundation* (ONF), where vendors bring their networking devices and controllers for interoperability testing. InCNTRE is the first certified lab selected by ONF for its conformance-testing program [9].

9.1.1 Key Contributors to SDN from Academia

A significant number of SDN pioneers have worked or studied at Stanford and U.C. Berkeley, some of whom we list here:

- Nick McKeown, currently a professor of electrical engineering and computer science at Stanford, has established multiple technology companies that have been acquired by larger companies in the high-tech industry. He is credited with helping to ignite the SDN movement, along with Martin Casado and Scott Shenker.
- Martin Casado is currently a consulting assistant professor at Stanford University in addition to his work at Nicira, which he founded with Nick McKeown and Scott Shenker. He was initially one of the creators of OpenFlow through the Ethane project [3].
- Scott Shenker, a professor in the School of Electrical Engineering and Computer Science at U.C. Berkeley, was involved with Nick McKeown and Martin Casado in the creation of Nicira.
- Guru Parulkar is a professor at Stanford as well as having been executive director of the Clean Slate program at that university. He is also executive director of the ONRC. He is a strong advocate of Open SDN and a vocal opponent of *SDN washing* [4], the use of the term SDN to refer to related technologies other than Open SDN, such as SDN via APIs and SDN via hypervisor-based overlays.
- Guido Appenzeller, co-founder of Big Switch Networks, is another participant in Stanford's Clean Slate program. In his role at Big Switch he is one of the strongest advocates for the Open SDN approach to reinventing networks.
- David Erickson and Rob Sherwood are the authors of most of the modules for the Beacon SDN controller [5]. Since most of the current open source SDN controllers were forked from that original Beacon source code, they have had significant influence on the design of contemporary Open SDN controllers. Controller implementations that were derived from Beacon include Floodlight and the OpenFlow portion of OpenDaylight.

9.2 Industry Research Labs

Quite often major enterprises and vendors support their own research labs that are performing basic research. We distinguish such pure research from research directed toward specific product development. These corporations often participate in research organizations and conferences and contribute papers based on their findings. For example, some research labs that have presented papers on SDN at recent SIGCOMM conferences are as follows:

- Telekom Innovation Laboratories (research arm of Deutsche Telekom)
- NTT Innovation Institute
- Microsoft Research
- Hewlett-Packard (HP) Laboratories
- Fujitsu Laboratories Ltd.
- IBM Research
- NEC Labs

These companies contribute to the furthering of SDN through research into topics such as scalability, deployment alternatives, ASIC development, and many other topics central to SDN.

9.3 Network Equipment Manufacturers

A number of networking vendors were quick to join the SDN movement, some of them contributing OpenFlow-enabled devices to researchers for testing, even before the first official OpenFlow standard document was published. Others have provided funding for SDN-related standards organizations or industry alliance research institutions. Some have made small acquisitions to gain access to SDN technology. Some have begun to support Open SDN, whereas others promote an SDN-via-APIs or an SDN-via-overlays strategy.

Virtually every NEM has SDN as part of its story today. This is a virtual marketing necessity, considering all the hype surrounding SDN, yet much of it constitutes the SDN washing mentioned in Section 9.1.1. It is not our intent in this chapter to document all the vendors' SDN claims. We focus on specific NEMs that, due to either their size or their commitment to SDN, seem most likely to influence the future course of SDN for better or worse. In Table 9.1 we list some of these SDN networking vendors and the products they offer in the areas of SDN device, SDN controller, and SDN application. The NEMs called out in the table are not the only ones to have brought SDN products to market as of this writing, but we believe that they reflect the most significant contributions by NEMs thus far. In this section we provide some brief background on the NEMs cited in Table 9.1.

Cisco was not involved with the early work on OpenFlow; the company began to support OpenFlow in its devices in 2012. Cisco is now heavily involved with SDN. Though only a lukewarm supporter of

Table 9.1 2013 SDN Commercial Product List by Vendor

Vendor	SDN Devices	SDN Controllers	SDN Applications
Cisco	OpenFlow: coming SDN via APIs	ONE (XNC)/onePK	
Hewlett-Packard	OpenFlow: 3500, 5400 and 8200	OpenFlow	Security (Sentinel)
Brocade	OpenFlow: NetIron CER, CES SDN via APIs		
VMware	OpenFlow: Open vSwitch (OVS)	NVP (NSX)	NVP (NSX)
Big Switch	OpenFlow: Indigo OpenFlow: Switch Light	Big Network Controller	Big Virtual Switch Big Tap
IBM	OpenFlow: RackSwitch and Flex System	Programmable Network Controller	SDN VE DOVE
NEC	Programmable Flow 5240 and 5820	Programmable Flow Controller	Virtual Tenant Networks Firewall, others
Extreme	OpenFlow: Switch Light		
Juniper	OpenFlow: MX, EX series	Coming: API-based	
Alcatel-Lucent	SDN via APIs		
Arista	OpenFlow: 7050 series SDN via APIs		

the ONF, Cisco is the primary driving force behind the OpenDaylight Project, focused on an open source SDN controller, as we discuss in Section 9.9.2. In addition to its open source efforts in the OpenDaylight project, Cisco strongly promotes SDN via its proprietary APIs. The vendor claims to support APIs and programmability to more of the networking stack than is available with only OpenFlow. Cisco's two main thrusts are its *Extensible Network Controller* (XNC) and its programmability aid, the *Open Networking Environment Platform Kit* (onePK), which we discussed in Section 6.2.4. The controller supports proprietary APIs to Cisco devices as well as OpenFlow. Some of the stated possibilities for applications using onePK are for functions such as path determination, per-flow capabilities such as QoS, and automating configuration across many devices. As the dominant NEM, Cisco will undoubtedly play a large part in shaping the role that SDN will assume in industry. As Cisco creates APIs, other vendors will pragmatically follow suit and provide the same or very similar APIs in order to sell into Cisco markets, creating de facto standards. Two examples of this trend are:

- **Cisco Controller.** Cisco's XNC controller offers a proprietary northbound API, which is also part of OpenDaylight, and already some vendors (such as NEC, IBM, and others [22]) have written SDN applications that interface with that API.
- **Cisco Device APIs.** Cisco has made much progress in selling its prospective SDN solution, which includes OpenFlow but highlights the non-OpenFlow capabilities: Cisco's device APIs. As a consequence, other vendors that want to sell their products into environments using the Cisco controller may have to implement those same Cisco device APIs.

Note that Cisco's announced *spin-in* Insieme that we discuss in Chapter 12 is further evidence of the company's interest in aspects of SDN.

Brocade has been active for a number of years in providing OpenFlow-supporting networking devices to researchers. However, the vendor also provides SDN-via-APIs support in its devices and have promoted its RESTful API SDN support in use cases with various customers. This dual-pronged approach allows Brocade to offer customers short-term transition plans by offering SDN APIs on its devices as a migration path as well as supporting Open SDN as a longer-term strategy. Brocade is currently a major supporter of the OpenDaylight project. In Chapter 12 we discuss Brocade's acquisition of Vyatta, another SDN-related startup.

NEC and IBM have partnered together from the early days of the OpenFlow project. Their switch and OpenFlow controller implementations have similar capabilities. Both vendors have been stalwart contributors to the OpenFlow research community of OpenFlow-enabled switches. Both vendors have created applications for their controllers, including virtualization applications that implement overlay technology. Both companies are committed to OpenFlow and support SDN via overlays using OpenFlow as the southbound API from the controller.

IBM and NEC have joined the OpenDaylight project and have ported their virtualization applications to run on that controller. NEC's contribution is a version of its own network virtualization application, called *Virtual Tenant Networks*. IBM's contribution is a version of its network virtualization application, *SDN Virtual Environments*, which is called *Distributed Overlay Virtual Ethernet* (DOVE) in the OpenDaylight environment (see Section 6.3.3).

Like NEC and IBM, Hewlett-Packard contributed OpenFlow-supporting switches to the OpenFlow research community long before the first ratified version of OpenFlow was approved. HP continues to support OpenFlow in many switch product lines and claims to have shipped millions of

OpenFlow-supporting switch ports in the past few years. Among NEMs, HP stands out for its unequivocal support of OpenFlow and enthusiastic participation in the ONF.

HP offers a commercial OpenFlow-based SDN controller. At ONS 2013, HP presented a number of SDN applications using OpenFlow to implement security and traffic-prioritization features. HP currently offers the Sentinel SDN security application that we described in Section 8.3.3. Because of its large server business, HP is also heavily involved in OpenStack and, consequently, will likely provide some type of network virtualization support through that project.

In Table 9.1 we see several NEMs listed that we have not yet mentioned. Extreme is partnering with Big Switch Networks to create a lightweight switch, which we discuss in more detail in Section 9.5. Alcatel-Lucent provides SDN via APIs in its networking devices. Juniper is headed toward a strategy aligned with SDN via APIs and has also acquired the startup Contrail. The Contrail controller provides network virtualization technology via overlays, presumably using the RESTful APIs that Juniper is building into its networking products. Arista is very vocal in SDN forums and ships commercial SDN products. Arista does offer some OpenFlow support, but its emphasis is on SDN via APIs.

9.4 Software Vendors

The move toward network virtualization has opened the door for software vendors to play a large role in the networking component of the data center. Some of the software vendors that have become significant players in the SDN space include VMware, Microsoft, and Big Switch as well as a number of startups. Here we will try to put their various contributions into context.

VMware, long a dominant player in virtualization software for the data center, has contributed significantly to the interest and demand for SDN in the enterprise. VMware boldly altered the SDN landscape when it acquired Nicira. VMware's purchase of Nicira has turned VMware into a networking vendor. Nicira's roots, as we explained in Section 9.1.1, come directly from pioneers in the Open SDN research community.

VMware's current offerings include *Open vSwitch* (OVS) as well as the *Network Virtualization Platform* (NVP) acquired through Nicira. (NVP is now marketed as VMware NSX.) NVP uses OpenFlow (with some extensions) to program forwarding information into its subordinate OVS switches.

Current VMware marketing communications claim that SDN is complete with only SDN via overlays. OpenFlow is VMware's southbound API of choice, but the emphasis is on network virtualization via overlay networks, not on what can be achieved with OpenFlow in generalized networking environments. This is a very logical business position for VMware. The holy grail for VMware is not about academic arguments but about garnering as much of the data center networking market as it can. This is best achieved by promoting VMware's virtualization strengths and avoiding esoteric disputes about one southbound API's virtues versus other approaches.

Other vendors and enterprises have begun to see the need to inter-operate with the Nicira solution. Although VMware's solution does not address physical devices, vendors who want to create overlay solutions that work with VMware environments will likely need to implement these Nicira-specific APIs as well.

Though VMware and Cisco were co-definers of the VXLAN [17] standard, the two companies now appear to be diverging. Cisco is focused on its XNC controller and API strategy, and VMware is focused on NSX. With respect to tunneling technologies, Cisco promotes VXLAN, which is designed for software and hardware networking devices, whereas VMware promotes both VXLAN and STT. STT [19] has a potential performance advantage in the software switch environment customary for VMware. This advantage derives from STT's ability to use the server NICs TCP hardware acceleration to improve network speed and reduce CPU load.

As a board-level participant in the ONF, Microsoft is instrumental in driving the evolution of OpenFlow. Microsoft's current initiative and effort regarding SDN have been around the Azure public cloud project. Similar to VMware, Microsoft has its own server virtualization software, called *Hyper-V*, and is utilizing SDN via overlays to virtualize the network as well. Microsoft's solution uses NVGRE [18] as the tunneling protocol, providing multitenancy and the other benefits described in Section 7.3.2.

Big Switch Networks was founded in 2010 by Guido Appenzeller with Rob Sherwood as CTO of controller technology, both members of our SDN hall of fame. Big Switch is one of the primary proponents of OpenFlow and Open SDN. Big Switch has Open SDN technology at the device, controller and application levels. The company created an open source OpenFlow switch code base called Indigo. Indigo is the basis for Big Switch's commercial OpenFlow switch software, which the vendor markets as *Switch Light*. The Switch Light initiative is a collaboration of switch, ASIC, and SDN software vendors to create a simple and cost-effective OpenFlow-enabled switch. Big Switch provides a version of Switch Light intended to run as a virtual switch, called *Switch Light for Linux*, as well as one targeted for the white-box hardware market, called *Switch Light for Broadcom*. We discuss white-box switch concept in Section 9.5.

Big Switch provides both open source and commercial versions of an SDN controller. The commercial version is called *Big Network Controller*, which is based on its popular open source controller called *Floodlight*. Big Switch also offers SDN applications, notably *Big Virtual Switch*, which provides network virtualization through overlays using OpenFlow virtual and physical devices.

There are numerous software startups are minor players in the SDN space. Both Nicira and Big Switch were startups. Nicira, as we have mentioned, was the target of a major VMware acquisition. Big Switch has received a large amount of venture capital funding. Both of these companies have become major forces on the SDN playing field. There are a number of other SDN software startups that have received varying amounts of funding and in some cases have been acquired. Except for Nicira and Big Switch, we do not feel that any of these have yet to become major voices in the SDN dialogue. Since they have been influential in terms of being recipients of much venture capital attention and investment and may in the future become major players in SDN, in Chapter 12 we discuss startups such as Insieme, PLUMgrid, and Midokura and their business ramifications for SDN.

9.5 White-Box Switches

Earlier we defined a white-box switch as a hardware platform that is purpose-built to be easily loaded with an OpenFlow device implementation such as Switch Light or OVS. The goal is to create a simple, inexpensive device that can be controlled by an OpenFlow controller and the SDN applications that run on top of it. The control software that network vendors typically put into their devices is largely absent from a white-box switch.

The natural alliances in the white-box switch ecosystem are depicted in Figure 9.1. Enterprises that spend fortunes on networking equipment are naturally drawn to a technology that would allow them to populate the racks of their mega data centers with low-cost switching gear. A company like Big Switch that bets its future on the increasing demand for its SDN controller due to an explosion in the number of OpenFlow-enabled switches clearly wants to foster the white-box model [10]. Merchant silicon vendors and *original device manufacturers* (ODMs) form the nexus of the physical white-box switch manufacturing ecosystem.

The merchant silicon vendors that make the switching chips are increasingly aligned with capabilities in the OpenFlow standards. Traditionally, many of the hardware platforms sold by the major NEMs, particularly lower-end devices, are actually manufactured by ODMs. The ODMs, in turn, are focused on the most cost-effective manufacturing of hardware platforms whose hardware logic largely consists of switching chips from the merchant silicon vendors plus commodity CPUs and memory. Industry consolidation means that the hardware switching platforms look ever more similar. The major NEMs have traditionally distinguished their products by control and management software and marketed those features plus the support afforded by the large organization. Now that the control and management software can be provided by software vendors such as Big Switch, the possibility exists for the marriage of the general-purpose OpenFlow device software to be loaded onto a white-box device. As we explained in Section 6.4, this situation is reminiscent of the PC, tablet, and smartphone markets where ODMs manufacture the hardware platforms, and operating systems such as Windows and Linux are loaded onto those platforms in a generic fashion.

Obviously, though this ecosystem intimately involves the merchant silicon vendors, the ODMs, and the software vendors, the role of the traditional NEMs is diminished. It is important to recognize, though, that the white-box switch concept does not readily apply to all networking markets. When the customer is large and sophisticated, such as a major cloud services vendor like Google, the support provided by the traditional NEM is of less value. However, many customers will continue to need the technical support that comes from the traditional NEM, even if they choose to migrate to the OpenFlow model.

One example of a white-box ecosystem is the Switch Light Initiative formed by Big Switch. As the originator of Switch Light, Big Switch offers its OpenFlow device software to switch manufacturers, with the goal of developing a broader market for its OpenFlow controllers. Conceptually this switch software can be adapted to run with any merchant silicon switching chips, but the initial merchant silicon vendor involved is Broadcom, and there is now a version of the Switch Light software available called *Switch Light for Broadcom*, as we mentioned earlier. ODMs are a natural target for this software, but the initial device partner for Switch Light is Extreme Networks, which is a traditional NEM. Extreme Networks has an advantage over the ODMs in that it does have the support network that we stated was necessary for many networking customers but not within the ODMs' capacity to deliver.

The startup Pica8's entire business model is based on a white-box strategy wherein Pica8 pairs the open source OpenFlow software with white-box hardware and sells the bundled package. Their operating system and software run on white-box switching hardware available from multiple ODM partners, and the switches are controlled using the OpenFlow-capable OVS open source switching code. We discuss Pica8 further in Section 12.9.1.

Cumulus Networks is another startup that offers software ready-made for white-box hardware from ODMs such as Accton and Quanta. The Cumulus idea, though, is more related to the idea of *opening up the device* (introduced in Section 6.4) than to Open SDN. Although an OpenFlow-capable system can be loaded on top of the Cumulus operating system, its system is not coupled to OpenFlow. Indeed, Cumulus touts that it is possible to use its system with white-box switches without using a centralized controller at all.

9.6 Merchant Silicon Vendors

Merchant silicon vendors provide specialty chips that are assembled into finished hardware products by NEMs or by ODMs, which then sell the bare-metal boxes to NEMs, which market the hardware under their own brand, usually with their own software. Merchant silicon vendors are interested in any new approach that increases the volume of chips that they sell, so a higher-volume, lower-cost model for switching hardware is in their interests. To this end, merchant silicon vendors that make switching chips that are compatible with OpenFlow have been naturally interested in SDN and, in particular, in the white-box switch ecosystem. Two major merchant silicon vendors that have been particularly active in the white-box switch arena are Intel and Broadcom.

Intel is a strong proponent of the white-box switch approach, as well as of network device and server NICs that are designed to handle the flow-matching and processing needs of OpenFlow 1.1 and beyond. Intel's white-box strategy is focused on its *Device Plane Development Kit* (DPDK), which we introduced in Section 6.4.

For many years Broadcom has been a vendor of switching chips that ODMs have used to build wired switches. Broadcom's dominance in this area is evidenced by the fact that Big Switch's Switch Light currently comes in one version for hardware devices, and that version is called *Switch Light for Broadcom*.

A lesser-known player in merchant silicon for SDN is Mellanox. Long known for its InfiniBand products, Mellanox has quietly tried to take the lead in OpenFlow-enabled switching silicon. Thus far, there is not much evidence to document commercial success of the company's product, but there is certainly room in the marketplace for such technology. We explained in Section 5.7 that the most recent versions of OpenFlow have surpassed the ability of extant silicon to support the full feature set of those specifications. It is a widely held belief that the only real solution is to design switching chips specifically to support OpenFlow rather than trying to adapt older designs to do something for which they were not designed. Mellanox is attempting just that.

9.7 Original Device Manufacturers

ODMs are focused on the most cost-effective manufacturing of hardware platforms whose hardware logic is mostly embodied in switching chips from the merchant silicon vendors and commodity CPUs and memory. These companies largely tend to be Taiwanese, Korean, and Chinese low-cost manufacturers. They excel at high-volume, low-cost manufacturing. They have traditionally relied on their NEM customers to provide end-customer support and to develop complex sales channels. For this reason, they have been relegated to a relatively low-margin business of selling bare-metal hardware to NEMs that brand these boxes and sell them through their mature marketing and support organizations. The ODMs desire to move up the value chain and enter a higher-margin business model closer to the customer. This process should be aided by customers who are less reliant on NEMs and would rather purchase hardware directly from the ODMs with the expectation of lower CAPEX costs. Two examples of ODMs that are particularly interested in the white-box switch model for SDN are Accton and Quanta.

9.8 Enterprises

A significant amount of the momentum behind SDN has come from large customers who purchase networking gear from the likes of Cisco, Juniper, Brocade, and others. These enterprises are interested in stimulating the evolution of networking toward something more functional and cost-effective. As a consequence, these companies have made significant contributions to SDN over the last few years.

There are two subcategories of these enterprises that have displayed the greatest interest in SDN: cloud services and carriers. Cloud services companies such as Google started by providing public Internet access to specialized data that was a target of wide consumer interest. The carriers traditionally provided network infrastructure. Increasingly, though, both categories of enterprise offer *Everything as a Service* (XaaS), providing large data centers that offer a vast number of tenants dynamic access to shared compute, storage, and networking infrastructure. Such large data centers have provided the most fertile ground for making SDN inroads. Hence, these two classes of enterprise have taken prominent roles in the ONF. The synergy between the enterprise and the ONF, which we discuss further in Section 9.9.1, was reflected in Figure 9.1.

Google has intense interest in improving the capabilities and lowering the cost of networking and has thus created its own simplified set of networking devices that use OpenFlow and a controller to manage their inter-data-center WAN connections. We described such an example of Google using SDN in the WAN in Section 8.1.2.

NTT has long been involved in SDN and OpenFlow research. NTT has created its own SDN controller called *Ryu*, which manages network devices using OpenFlow 1.0, 1.1, 1.2, and 1.3; a well-known network configuration protocol called NETCONF [13]; and the *OpenFlow Configuration and Management Protocol* (OF-config) [14]. As described in Section 8.2.3, Verizon has been involved in large SDN proofs of concept related to cloudbusting Verizon has also been a leader in the related *network functions virtualization* (NFV) technology, since the company chairs the NFV committee for the *European Telecommunications Standards Institute* (ETSI), which is currently taking the lead in NFV research and standardization [15]. Deutsche Telekom recently selected Tail-f as partner for its ongoing architectural SDN-based project called TeraStream. Tail-f's *Network Control System* (NCS) is designed to support SDN functionality, including OpenFlow. NCS includes an OpenFlow controller called *Tailflow*, and applications written for other OpenFlow controllers may be integrated into NCS. NCS works not only with SDN and OpenFlow devices but also with legacy, non-SDN, and non-OpenFlow devices.

Other large enterprises such as banks run large data centers that also have significant interest in SDN even though they are not part of either of the two major categories we've discussed. For example, Goldman Sachs is on the board of the ONF and is very public about its support for OpenFlow.

9.9 Standards Bodies and Industry Alliances

For new technologies to become widely adopted, standards are required. In this section we discuss two different kinds of organizations that produce standards related to SDN. The most obvious genre is a true standards body, such as the IETF or IEEE. A less obvious sort is what we refer to here as an *open industry alliance*. Such an alliance is a group of companies with a shared interest surrounding a technology. A good historical example of an industry alliance and a standards body working on the same technology is the WiFi Alliance and the IEEE 802. The WiFi Alliance created important standards documents (e.g., WPA, WPA2) but was an industry alliance first, not a standards body. In the instance of SDN, it is the industry alliances that have taken the lead with standards, more so than the standards bodies. The three alliances are the ONF, OpenDaylight, and OpenStack. The general idea of OpenStack is to create an abstraction layer above compute, network, and storage resources. Though the network component of this abstraction layer has led to a relationship between OpenStack and SDN, SDN is not its primary focus, so we direct our attention first to the ONF and OpenDaylight, two industry alliances that are indeed laser-focused on SDN.

The ONF and OpenDaylight look at SDN from two very different perspectives. The driving force behind the ONF and the standards it creates are the board members, who represent enterprises such as Deutsche Telekom, Facebook, Goldman Sachs, Google, Microsoft, NTT Communications, Verizon, and Yahoo! There is not a single NEM on the board. OpenDaylight, on the other hand, was founded by NEMs and looks at SDN as a business opportunity and/or a challenge. Figure 9.2 illustrates the striking difference in board composition between the two bodies. The only company to hold a board seat on both alliances is Microsoft.

It is easy to fall victim to the temptation of thinking that an alliance controlled by NEMs will favor protecting its members' business interests over innovating in a way that most helps their user community. There are downsides to the enterprise-driven alliance as well, though. There is the real possibility

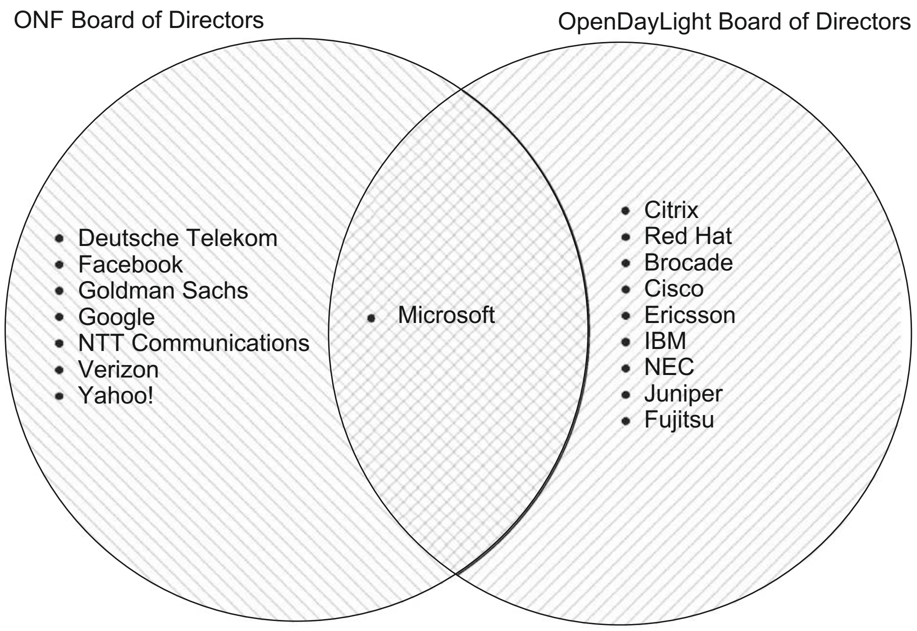


FIGURE 9.2

ONF vs. OpenDaylight board membership.

that the standards that are created will be out of sync with the hardware capabilities of networking devices. To a certain degree this has been true with OpenFlow 1.1, 1.2, and 1.3. This explains in part vendors' reluctance to implement these subsequent revisions of the OpenFlow standard. Conversely, being unencumbered by vendor limitations has resulted in OpenFlow defining what the industry needs rather than what the vendors can immediately provide. This is the genesis of the conflict between OpenDayLight and ONF reflected in Figure 9.1.

9.9.1 Open Networking Foundation

The ONF occupies a preeminent place within the set of standards bodies contributing to SDN. The ONF was created in 2011 with the stated purpose of *the promotion and adoption of Software Defined Networking (SDN) through open standards development* [12]. The ONF is the owner of the OpenFlow standards process. To date, OpenFlow is the most influential work produced by the ONF. The various working groups active within the ONF as of this writing were listed in Table 3.2.

9.9.2 OpenDaylight

OpenDaylight's mission is to *facilitate a community-led, industry-supported open source framework, including code and architecture, to accelerate and advance a common, robust Software Defined Networking platform* [23]. The project is part of the Linux Foundation of Collaborative Projects and boasts many major networking vendors as members (e.g., Brocade, Cisco, IBM, HP, Huawei, Intel, NEC, VMware, and others).

OpenDaylight welcomes source code contributions from its members and has a fairly rigorous process for software review and inclusion. As can be seen by the list of contributing engineers as well as the OpenDaylight controller interface, much of the controller software comes from Cisco. In addition to the many Cisco contributors, many other contributors have added software to the OpenDayLight open source code base. Plexxi has provided an API allowing the controller and applications to collaborate using an abstraction of the underlying network infrastructure that is independent of the particular switching equipment used. Software to prevent *distributed denial of service* (DDoS) attacks has been contributed by Radware. Ericsson and IBM collaborated with Cisco to provide OpenFlow plugins to the controller. Pantheon has provided a version of OpenFlow 1.3. The basic OpenFlow functionality in the controller is based on the Beacon open source OpenFlow controller.

OpenDaylight's first release, code-named "Hydrogen", was released in February 2014. Some applications have been ported onto the OpenDaylight controller, notably DOVE by IBM and Virtual Tenant Networks by NEC, as mentioned earlier in this chapter.

The ONF focuses on defining a specific protocol (OpenFlow) between network devices and a controller to move the control plane from those devices to the controller. OpenDaylight focuses not on a single control protocol or mandating that network devices conform to that protocol. Rather, OpenDaylight is focused on providing an open source *network operating system*, playing a role for SDN controllers much like the role Linux plays in servers. We depict this very different focus between the ONF and OpenDaylight consortia in Figure 9.3. Many different controller-to-device protocols are supported under this umbrella of open source functionality. Cisco remains a powerful driving force behind OpenDaylight, and there remains some skepticism regarding OpenDaylight over concerns that it will merely turn out to be a standardization of Cisco's controller. Thus, we are left with an open question

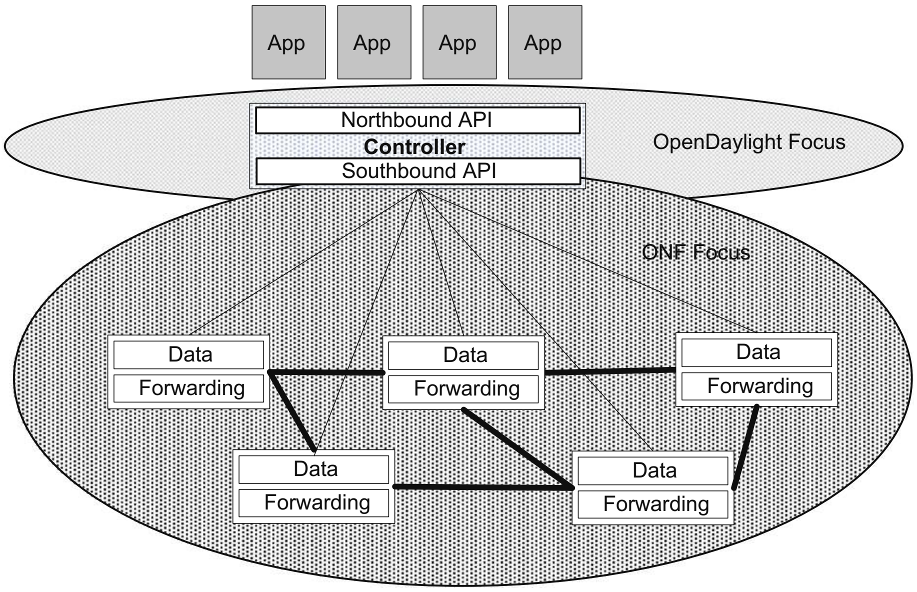


FIGURE 9.3

Focus of ONF vs. OpenDaylight.

regarding whether the *big tent* of tolerance in OpenDaylight is really about providing as many options as possible in an open source platform or about allowing incumbent NEMs to change only as much as they want while purporting to be part of the evolution toward SDN. We leave this to the reader to decide.

9.9.3 OpenStack

The OpenStack project was started by Rackspace and the *National Aeronautics and Space Administration* (NASA) in 2010 as an open source IaaS toolset. In a sense, OpenStack can be viewed as the Linux of cloud computing. The interest in this open source project quickly grew, culminating in the establishment in September 2012 of the nonprofit OpenStack Foundation. The mission of the OpenStack Foundation is to *promote, protect, and empower OpenStack software and its community*. As of this writing, the foundation has more than 6,700 member companies spread over 83 countries. The largest current deployment of OpenStack remains in one of its founding companies, Rackspace, which is a major provider of cloud computing.

The general idea of OpenStack is to create an abstraction layer above compute, network, and storage resources. This abstraction layer provides APIs that applications can use to interact with the hardware below, independent of the source of that hardware. The interfaces for compute, storage, and network present pools of resources for use by the applications above.

The OpenStack networking definition states that SDN is not required, but that *administrators can take advantage of SDN technology like OpenFlow to allow for high levels of multitenancy and massive* *scale* [16]. The network virtualization component of OpenStack is called Neutron and thus is the component most relevant to our discussion of SDN. Neutron was formerly called Quantum. Neutron provides an application-level networking abstraction. It is intended to provide users of cloud computing with the ability to create overlay networks that are easier for the cloud tenants to manage and understand by abstracting the details of the underlying physical network devices. It attempts to provide *networking as a service* by allowing the cloud tenants to interconnect *virtual network interfaces* (vNICs) [11]. The actual mapping of these virtual network interfaces to the physical network requires the use of Neutron plugins. This is where the relationship between OpenFlow and OpenStack exists. One Neutron plugin could be an interface to an OpenFlow controller that would control physical switches [2]. We discuss OpenStack architecture further in Section 11.10.

Although this link between OpenStack and SDN is thin, OpenStack's impact as an industry alliance means that it is an important SDN player despite the peripheral role played by SDN in the overall OpenStack architecture.

9.9.4 IETF and IEEE

Although there are no current standardization efforts in the IETF directed solely at SDN, the IETF is active in standards that are indirectly related to SDN, such as the existing standards for network virtualization including VXLAN, NVGRE, and STT, which we discussed in Section 7.3 as SDN technologies for overlays and tunneling. Others include the *path computation element* (PCE) [20] and *network virtualization overlays* (nvo3) [21] standards.

The IEEE was not involved in the early days of SDN, but this important standards body is now beginning work on aspects of SDN, so before long we expect to see SDN-related activity emanating from the IEEE.

9.10 Conclusion

In the course of presenting the major players in SDN, this chapter reads somewhat like a *Who's Who* in networking. At times it does appear that the world is jumping on the SDN bandwagon in order to not be left behind. We must temper this idea by saying that not every NEM or even every large enterprise customer is certain that SDN lies in their future. As an example, for all the enthusiasm for SDN emanating from Google and other cloud services enterprises, Amazon is notable by its absence from any type of membership whatsoever in the ONF and by its relative silence about building its own SDN solution. Amazon is a huge enterprise with large data centers and the need for the advances in networking technology that have the rest of the industry talking about SDN, and one would expect the company to have similar mega data center needs as Google and Yahoo! It is possible that Amazon is indeed working on an SDN solution but doing so in stealth mode. It is also possible that Amazon is working on a low-cost white-box switch solution that is not tied to SDN, such as the Cumulus solution. Only time will tell whether Amazon and other SDN backbenchers turn out to be latecomers to the SDN party or whether they were wise to be patient during the *peak of unreasonable expectations* that may have occurred with respect to SDN.

We have stated more than once in this work that openness is a key aspect of what we define as SDN. In formulating this chapter on the key players in SDN, however, it was not possible to talk about open source directly since it is not an entity, and thus it is not within the scope of this chapter. Indeed, many of the SDN players we discussed here are significant largely *because* of their open source contributions to SDN. In any event, openness and open source are key facets of SDN as we have defined it. Indeed, Chapter 11 focuses entirely on open source software related to SDN. First, though, in the next chapter we provide a tutorial on writing Open SDN applications.

References

[1] Clean slate: an interdisciplinary research program. Stanford University. Retrieved from <cleanslate.stanford.edu>.

[2] Miniman S. SDN, OpenFlow and OpenStack quantum. Wikibon, July 17, 2013. Retrieved from <wikibon.org/wiki/v/SDN,_OpenFlow_and_OpenStack_Quantum>.

[3] Ethane: a security management architecture. Stanford University. Retrieved from <yuba.stanford.edu/ethane>.

[4] Parulkar G. Keynote session: opening talk (Video), open networking summit, April 2013, Santa Clara, CA, USA. Retrieved from <www.opennetsummit.org/archives-april2013>.

[5] Beacon 1.0.2 API. Stanford University, Palo Alto, CA, USA. Retrieved from <openflow.stanford.edu/static/beacon/releases/1.0.2/apidocs>.

[6] ONRC Research. Retrieved from <onrc.stanford.edu>.

[7] ONRC Research Videos. Retrieved from <onrc.stanford.edu/videos.html>.

[8] InCNTRE, Indiana University, Retrieved from <incntre.iu.edu>.

[9] Open networking foundation sees high growth in commercialization of SDN and OpenFlow at PlugFest. Open Networking Foundation; October 18, 2012. Retrieved from <www.opennetworking.org/news-and-events/press-releases/248-high-growth-in-commercialization-of-sdn>.

[10] Banks E. Big switch leaves OpenDaylight, touts white-box future, network computing; June 6, 2013. Retrieved from <www.networkcomputing.com/data-networking-management/big-switch-leaves-opendaylight-touts-whi/240156153>.

[11] Neutron. OpenStack. Retrieved from <wiki.openstack.org/wiki/Neutron>.

[12] ONF overview. Open Networking Foundation. Retrieved from <www.opennetworking.org/about/onf-overview>.

[13] Enns R, Bjorklund M, Schoenwaelder J, Bierman A. Network configuration protocol (NETCONF), RFC 6241. Internet Engineering Task Force; June 2011.

[14] OpenFlow Management and Configuration Protocol, Version 1.1.1. Open Networking Foundation; March 23, 2013. Retrieved from <www.opennetworking.org/sdn-resources/onf-specifications>.

[15] Network Functions Virtualization. European Telecommunications Standards Institute. Retrieved from <www.etsi.org/technologies-clusters/technologies/nfv>.

[16] OpenStack networking. OpenStack Cloud Software. Retrieved from <www.openstack.org/software/openstack-networking>.

[17] Mahalingam M, Dutt D, Duda K, Agarwal P, Kreeger L, Sridhar T, et al. VXLAN: a framework for overlaying virtualized layer 2 networks over layer 3 networks, Internet Engineering Task Force; August 26, 2011 [internet draft].

[18] Sridharan M, Duda K, Ganga I, Greenberg A, Lin G, Pearson M, et al. NVGRE: network virtualization using generic routing encapsulation, Internet Engineering Task Force; September 2011 [internet draft].

[19] Davie B, Gross J. STT: a stateless transport tunneling protocol for network virtualization (STT), Internet Engineering Task Force; March 2012 [internet draft].

[20] Path Computation Element (PCE) Working Group. Internet Engineering Task Force. Retrieved from <datatracker.ietf.org/wg/pce>.
[21] Network Virtualization Overlays (NVO3) Working Group. Internet Engineering Task Force. Retrieved from <datatracker.ietf.org/wg/nvo3>.
[22] Influx of new technology contributions to OpenDaylight advances software-defined networking. OpenDaylight; July 25, 2013. Retrieved from <www.opendaylight.org/announcements/2013/07/influx-new-technology-contributions-opendaylight-advances-software-defined>.
[23] Welcome to OpenDaylight. OpenDaylight. Retrieved from <www.opendaylight.org>.

CHAPTER

SDN Applications

10

We now descend from the conceptual discussions about SDN use cases and take a deeper dive into how SDN applications are actually implemented. This chapter focuses on SDN application development in an Open SDN environment. Most SDN-via-overlay offerings today are bundled solutions that come ready-made with a full set of functionality, generally without the ability to add applications to those solutions. SDN-via-API solutions in general allow for some degree of user-level programming but do not provide the network-wide view and abstraction layer afforded by the Open SDN controller. OpenDaylight is a platform that forms the basis for hybrid application development, allowing applications to control non-OpenFlow-enabled switches as well as OpenFlow-enabled switches. Many detailed functional block diagrams are included in this chapter, as is a fair amount of sample source code that serves as helpful examples for the network programmer.

10.1 Before You Begin

Before embarking on a project to build an SDN application, one must consider a number of questions. Answering these questions will help the developer know how to construct the application, including what APIs are required, which controller might be best suited for the application, and what switch considerations should be taken into account. Some of these questions are:

- *What is the basic nature of the application?* Will it spend most of its time reacting to incoming packets that have been forwarded to it from the switches in the network? Or will it typically set flows proactively on the switches and only respond to changes in the network that require some type of reconfiguration?
- *What is the type and nature of network with which the application will be dealing?* For example, a campus network application geared toward access control will have a different design than a campus application that is concerned with general network policies such as the prioritization of different types of traffic.
- *What is the intended deployment of the controller with respect to the switches it controls?* A controller that is physically removed from the switches it controls (e.g., in the cloud) will clearly not be able to efficiently support an SDN application that is strongly reactive in nature because of the latency involved in forwarding packets to the controller and receiving a response.
- *Are the SDN needs purely related to the data center and virtualization?* If so, perhaps an SDN overlay solution is sufficient to meet current needs, and future needs related to the physical underlay network can be met later, as OpenFlow-related technologies mature.

- *Will the application run in a greenfield environment, or will it need to deal with legacy switches that do not support SDN?* Only in rare situations is one able to build an SDN application from the ground up. Typically, it is required to use existing equipment in the solution. It is important to understand whether the existing switches can be upgraded to have OpenFlow capability and whether or not they will have hybrid capability, supporting both OpenFlow and legacy operation. If the switches do not support OpenFlow, it is important to understand the level of SDN-via-APIs support that is available.

The answers to these and similar questions will help the developer understand the nature of the SDN application. Assuming that an Open SDN application will be developed, the developer will need to decide between two general styles of SDN applications: *reactive* or *proactive*.

10.2 Reactive versus Proactive Applications

Typically, in the case of reactive SDN applications, the communication between the switch and the controller will scale with the number of new flows injected into the network. The switches may often have relatively few flow entries in their flow tables, since the flow timers are set to match the expected duration of each flow in order to keep the flow table size small. This results in the switch frequently receiving packets that match no rules. In the reactive model, those packets are encapsulated in PACKET_IN messages and forwarded to the controller and thence to an application. The application inspects the packet and determines its disposition. The outcome is usually to program a new flow entry in the switch(es) so that the next time this type of packet arrives, it can be handled locally by the switch itself. The application will often program multiple switches at the same time so that each switch along the path of the flow will have a consistent set of flow entries for that flow. The original packet will often be returned to the switch so that the switch can handle it via the newly installed flow entry. The kind of applications that naturally lend themselves to the reactive model are those that need to see and respond to new flows being created. Examples of such applications include per-user firewalling and security applications that need to identify and process new flows.

On the other hand, proactive SDN applications feature less communication emanating from the switch to the controller. The proactive SDN application sets up the switches in the network with the flow entries that are appropriate to deal with incoming traffic before it arrives at the switch. Events that trigger changes to the flow table configurations of switches may come from mechanisms that are outside the scope of the switch-to-controller secure channel. For example, some external traffic-monitoring module will generate an event which is received by the SDN application, which will then adjust the flows on the switches appropriately. Applications that naturally lend themselves to the proactive model usually need to manage or control the topology of the network. Examples of such applications include new alternatives to spanning tree and multipath forwarding.

Reactive programming may be more vulnerable to service disruption if connectivity to the controller is lost. Loss of the controller will have less impact in the proactive model if the failure mode specifies that operation should continue. (If the failure mode configuration is set to *fail closed*, then loss of controller connectivity will cause the flow tables to be flushed independently of the style of the application.) With the proactive model, there is no additional latency for flow setup, because they are prepopulated. A drawback of the proactive model is that most flows will be wildcard-style, implying aggregated flows and thus less fine granularity of control.

We discuss these issues in more detail in the next subsections.

10.2.1 Terminology

To maintain consistency with the terminology used in the software libraries referenced in this section, the term *device* always refers to an end-user node, and the term *switch* always refers to a networking node, such as a wireless access point, switch, or router. In general in this book, the term *SDN device* includes SDN switches. In the vernacular of the particular Java APIs discussed in this section, *device* refers to the end-user node. Since we want to name the APIs faithfully to the actual source code implementation, we adhere to this convention for the term *device only* in Sections 10.2, 10.3, and 10.4.

10.2.2 Reactive SDN Applications

Reactive applications need to be asynchronously notified of incoming packets that have been forwarded to the controller from OpenFlow switches. In Section 6.2.3 we explained why RESTful APIs have become a popular method for manipulating SDN networks. The type of asynchronous notification required here is not performed via RESTful APIs, however. This is because a RESTful API is asymmetric, with a requester (the application) and a responder (the controller). Under the RESTful paradigm, it is not straightforward for the controller (responder) to asynchronously notify the application of an event such as the arrival of a packet from a switch. To accomplish this sort of asynchronous notification via a RESTful interface an additional, reverse-direction control relationship would need to be established, with the controller being the requester and the application being the responder. Though this is feasible, there is additional latency with this approach compared to the *listener* approach described below. With the Java APIs one can register a listener and then receive callbacks from the controller when packets arrive. These callbacks come with passed arguments including the packet and associated metadata, such as which switch forwarded the packet and the port on which the packet arrived. Consequently, reactive applications tend to be written in the native language of the controller, which, in the examples in this chapter, is Java. Note that C and Ruby controllers are also available, and the same designs and mechanisms described here apply to those languages as well.

Reactive applications have the ability to register listeners, which are able to receive notifications from the controller when certain events occur. Some important listeners available in the popular Floodlight and Beacon controller packages are:

- **SwitchListener.** Switch listeners receive notifications whenever a switch is added or removed or has a change in port status.
- **DeviceListener.** Device listeners are notified whenever a device (an end-user node) has been added, removed, or moved (attached to another switch) or has changed its IP address or VLAN membership.
- **MessageListener.** Message listeners get notifications whenever a packet has been received by the controller. The application then has a chance to examine it and take appropriate action.

These listeners allow the SDN application to react to events that occur in the network and to take action based on those events.

When a reactive SDN application is informed of an event, such as a packet that has been forwarded to the controller, a change of port state, or the entrance of a new switch or device into the network, the application has a chance to take some type of *action*. The most frequent event coming into the application would normally be a packet arriving at the controller from a switch, resulting in an action. Such actions include:

- **Packet-specific actions.** The controller can tell the switch to drop the packet, to flood the packet, to send the packet out a specific port, or to forward the packet through the NORMAL non-OpenFlow packet-processing pipeline, as described in Section 5.3.4.
- **Flow-specific actions.** The controller can program new flow entries on the switch, intended to allow the switch to handle certain future packets locally without requiring intervention by the controller.

Other actions are possible, some of which may take place outside the normal OpenFlow control path, but the packet-specific and flow-specific actions constitute the predominant behavior of a reactive SDN application.

Figure 10.1 shows the general design of a reactive application. Notice that the controller has a listener interface that allows the application to provide listeners for switch, device, and message (incoming packet) events. Typically, a reactive application will have a module to handle packets incoming to the controller that have been forwarded through the message listener. This packet processing can then act on the packet. Typical actions include returning the request to the switch, telling it what to do with the packet (e.g., forward out a specific port, forward NORMAL or drop the packet). Other actions taken by the application can involve setting flows on the switch in response to the received packet, which will inform the switch what to do the next time it sees a packet of this nature.

For reactive applications, the last flow entry will normally be programmed to match any packet and to direct the switch to forward that otherwise unmatched packet to the controller. This methodology is precisely what makes the application *reactive*. When a packet not matching any existing rule is encountered, it is forwarded to the controller so that the controller can react to it via some appropriate action. A packet may also be forwarded to the controller in the event that it matches a flow entry and the associated action stipulates that the packet be passed to the controller.

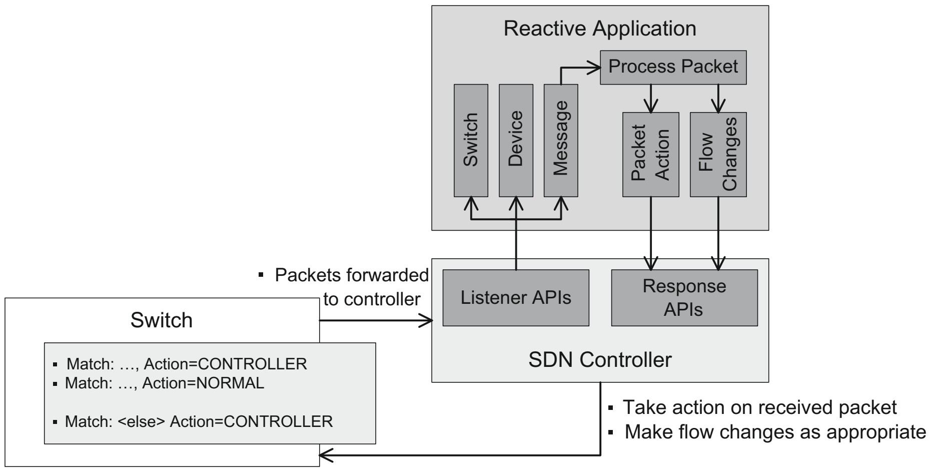

Reactive Application
Process Packet
Switch
Device
Message
Packet Action
Flow Changes
Packets forwarded to controller
Listener APIs
Response APIs
SDN Controller
Switch
Match: …, Action=CONTROLLER
Match: …, Action=NORMAL
Match: <else> Action=CONTROLLER
Take action on received packet
Make flow changes as appropriate

FIGURE 10.1

Reactive application design.

In the reactive model, the flow tables tend to continually evolve based on the packets being processed by the switch and by flows aging out. Performance considerations arise if the flows need to be reprogrammed too frequently, so care must be taken to appropriately set the idle timeouts for the flows. The point of an idle timer is to clear out flow entries after they have terminated or gone inactive, so it is important to consider the nature of the flow and what constitutes inactivity when setting the timeout. Unduly short idle timeouts will result in too many packets sent to the controller. Excessively long timeouts can result in flow table overflow. Knowing an appropriate value to set a timeout requires familiarity with the application driving the flow. As a rough rule of thumb, though, timeouts would be set on the order of tens of seconds rather than milliseconds or minutes.

10.2.3 Proactive SDN Applications

Proactive applications can be implemented using either the native API (e.g., Java) or using RESTful APIs. The reader should note that such RESTful APIs are not the ones resident in the switch that we described as part of SDN via APIs in Section 6.2.3. The RESTful APIs we describe here reside on the Open SDN controller and are called by the application. In contrast to reactive APIs, which are low-level and deal almost directly with the OpenFlow protocol below them, these controller RESTful APIs can operate at different levels, depending on the northbound API being used. They can be low-level, basically allowing the application developer to configure flow entries on the switches themselves. These RESTful APIs can also be high-level, providing a level of abstraction above the network switches so that the SDN application interacts with network components such as virtual networks rather than with the switches themselves.

Another difference between this type of application and a reactive one is that using RESTful APIs means there is no need for *listening* functionality. In the reactive application the controller can asynchronously invoke methods in the SDN application stimulated by events such as incoming packets and switches that have been discovered. This type of asynchronous notification is not possible with a unidirectional RESTful API, which is invoked on the controller by the application rather than the other way around. It is conceivable to implement an asynchronous notification mechanism from the controller toward a proactive application, but generally, proactive applications are stimulated by sources external to the OpenFlow control path. The traffic monitor described in Section 10.14 is an example of such an external source.

Figure 10.2 shows the general design of a purely proactive application. In such an application no listeners are receiving from the controller events about switches, devices, and messages. Such listeners are not a natural fit with a one-way RESTful API, wherein the application makes periodic calls to the API and the API has no means of initiating communication back to the application. As shown in Figure 10.2, proactive applications rely on stimulus from external *network events*. Such stimuli may originate from a number of sources, such as traffic or switch monitors using SNMP or SNORT, or external applications like a server virtualization service, which notifies the application of the movement of a virtual machine from one physical server to another. As we explained in Section 10.2, applications that deal with the network topology will be passed control protocol packets, such as DNS, DHCP, and BGP, by rules programmed to do just that.

The RESTful APIs are still able to *retrieve* data about the network, such as domains, subnets, switches, and hosts. RESTful APIs generally also provide an interface, often referred to as a *flow pusher*, which allows the application to set flows on switches. Thus the proactive application has the

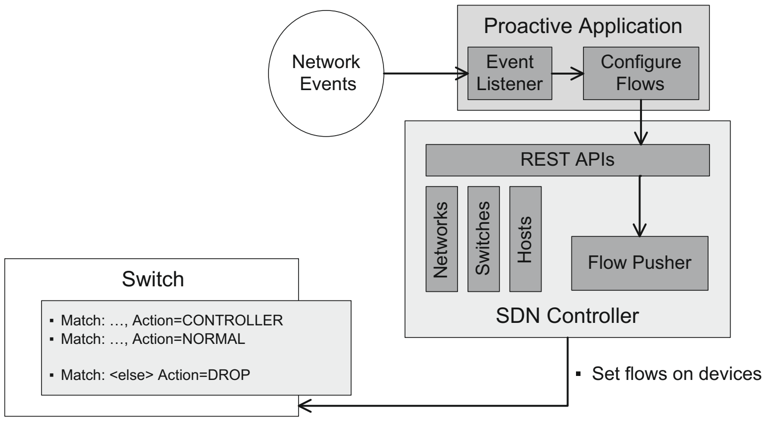


FIGURE 10.2

Proactive application design.

same ability to program flows as the reactive application but will tend to use wildcard matches more frequently.

As shown in Figure 10.2, the last flow entry will typically be to DROP unmatched packets. This is because proactive applications attempt to *anticipate* all traffic and program flow entries accordingly. As a consequence, packets that do not match the configured set of rules are discarded. We remind the reader that the match criteria for flow entries can be programmed such that most arriving packet types are expected and match some entry before this final DROP entry. If this were not the case, the purely proactive model could become an expensive exercise in dropping packets!

As mentioned earlier, there will be hybrid reactive-proactive applications that utilize a language such as Java for listeners and communicate via some external means to the proactive part of the application. The proactive component would then program new flow entries on the switch. As applications become more complex, this hybrid model is likely to become more common. The comment made previously about arriving packets in a proactive application matching some flow entry before the final DROP entry is also germane to the case of hybrid applications. The proactive part of such hybrid applications will have programmed higher-priority flow entries so that only a reasonable number of packets match the final DROP entry. In such a hybrid model, the final entry will send all unmatched packets to the controller, so it is important to limit this number to a rate that the controller can handle.

10.3 Analyzing Simple SDN Applications

Proactive, RESTful-based applications can run either on the same system as the controller or on a remote system, since HTTP allows that flexibility. In contrast, reactive applications are tightly bound to the controller as they access the controller via the native APIs (e.g., Java or C), and they run in the same container or execution space. In Section 10.12 we provide an example of proactive applications that use the RESTful *flow pusher* APIs. First, though, in Section 10.4 we look in detail at the source code involved in writing a reactive SDN application. We use this example as an opportunity to walk through SDN application source code in detail and explore how to implement the tight application-controller coupling that is characteristic of reactive applications.

The application examples we provide in this chapter use the Floodlight controller. We have chosen Floodlight because it is one of the most popular open source SDN controllers. It is possible to download the source code [1] and examine the details of the controller itself. Floodlight also comes with a number of sample applications, such as a learning switch and a load balancer, which provide excellent additional examples for the interested reader. The core modules of the Floodlight controller come from the seminal Beacon controller [2]. These components are denoted as *org.openflow* packages within the Floodlight source code. Since Beacon is the basis of the OpenFlow functionality present in OpenDaylight and other SDN controllers, the OpenFlow-related controller interaction described in the following sections would apply to those Beacon-derived controllers as well. We provide further description of the Beacon and Floodlight controllers in Sections 10.5 and 10.6.

10.4 A Simple Reactive Java Application

In the common vernacular, a blacklist is a list of people, products, or locations viewed with suspicion or disapproval. In computer networking, a blacklist is a list of hostnames or IP addresses that are known or suspected to be malicious or undesirable in some way. Companies may have blacklists that protect their employees from accessing dangerous sites. A school may have blacklists to protect children from attempting to visit undesirable web pages.

As implemented today, most blacklist products sit in the data path, examining all traffic at choke points in the network while looking for blacklisted hostnames or IP addresses. This clearly has disadvantages in terms of the latency it introduces to traffic as it all passes through that one blacklist appliance. There are also issues of scalability and performance if the number of packets or the size of the network is quite large. Cost would also be an issue, since provisioning specialized equipment at choke points adds significant network equipment expense.

It would be preferable to examine traffic at the edge of the network, where the users connect. But putting blacklist-supporting appliances at every edge switch is impractical from a maintenance and cost perspective. A simple and cost-effective alternative is to use OpenFlow-supporting edge switches or wireless access points (APs), with a Blacklist application running above the OpenFlow controller. In our simple application, we examine both hostnames and IP addresses. We describe the application below.

10.4.1 Blacklisting Hostnames

We introduced an SDN solution to the blacklist problem in Section 8.3.3. We provide further details of that solution here. The first part of the Blacklist application deals with hostnames. Checking for malicious or undesirable hostnames works as follows:

- A single default flow is set on the edge switches to forward all DNS traffic to the OpenFlow controller. (Note that this is an example of a proactive rule. Thus, though a blacklist is primarily a reactive application, it does exhibit the hybrid reactive-proactive characteristic discussed in Section 10.2.3.)
- The Blacklist application running on the controller listens for incoming DNS requests that have been forwarded to the controller.

- When a DNS request is received by the Blacklist application, it is parsed to extract the hostnames.
- The hostnames are compared against a database of known malicious or undesirable hosts.
- If any hostname in the DNS request is found to be bad, the switch is instructed to drop the packet.
- If all hostnames in the DNS request are deemed to be safe, the switch is instructed to forward the DNS request normally.

In this way, with one simple flow entry in the edge switches and one simple SDN application on the controller, end users are protected from intentionally or inadvertently attempting to access bad hosts and malicious websites.

10.4.2 Blacklisting IP Addresses

As described earlier in Section 8.3.3, websites sometimes have embedded IP addresses, or end users will attempt to access undesirable locations by providing IP-specific addresses directly rather than a hostname, thus circumventing a DNS lookup. Our simple Blacklist application works as follows:

- A single default flow is set on the edge switches, at a low priority, to forward all IP traffic that doesn't match other flows to the OpenFlow controller.
- The Blacklist application listens for IP packets coming to the controller from the edge switches.
- When an IP packet is received by the Blacklist application, the destination IP address is compared against a database of known malicious or undesirable IP addresses.
- If the destination IP address is found to be bad, the switch is instructed to drop the packet.
- If the destination IP address in the IP packet is deemed to be safe, the switch or AP is instructed to forward the IP packet normally, and a higher-priority flow entry is placed in the switch that will explicitly allow IP packets destined for that IP address.
- The IP destination address flow entry was programmed with an idle-timeout, which will cause it to be removed from the flow table after some amount of inactivity.

Note that a brute-force approach to the blacklist problem would be to explicitly program rules for all known bad IP addresses such that any packet matching those addresses is directly dropped by the switch. Though this solution works in principle, it is not realistic, because the number of blacklisted IP addresses is potentially very large and, as we have pointed out, the number of available flow entries, though varying from switch to switch, is limited.

There are three major components to this application:

- **Listeners.** The module that listens for events from the controller.
- **Packet handler.** The module that handles incoming packet events and decides what to do with the received packet.
- **Flow manager.** The module that sets flows on the switches in response to the incoming packet.

Figure 10.3 provides a more detailed view of the Blacklist application introduced in Section 8.3.3. As a reactive application, blacklist first programs the switch to forward unmatched packets to the controller. The figure depicts three listeners for network events: the switch listener, the device listener, and the message listener. We show the device listener because it is one of the three basic classes of listeners for reactive applications, but since it has no special role in Blacklist, we do not mention it further here. The *message listener* will forward incoming packets to the *process packet* module, which processes

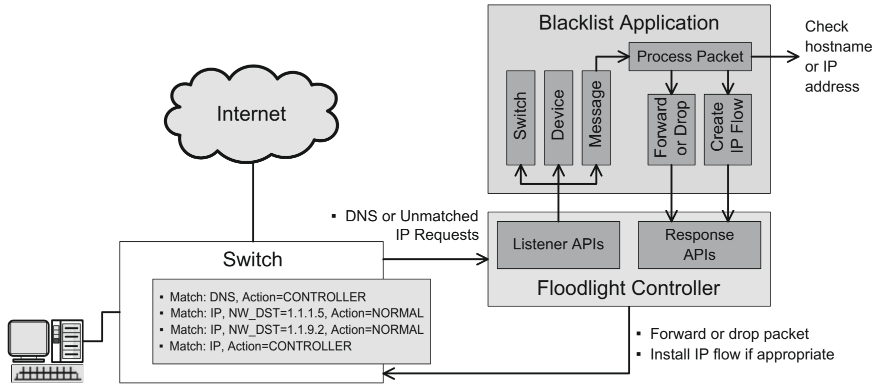

Blacklist Application
Process Packet
Check hostname or IP address
Switch
Device
Message
Forward or Drop
Create IP Flow
Internet
▪ DNS or Unmatched IP Requests
Listener APIs
Response APIs
Floodlight Controller
Switch
▪ Match: DNS, Action=CONTROLLER
▪ Match: IP, NW_DST=1.1.1.5, Action=NORMAL
▪ Match: IP, NW_DST=1.1.9.2, Action=NORMAL
▪ Match: IP, Action=CONTROLLER
▪ Forward or drop packet
▪ Install IP flow if appropriate

FIGURE 10.3

Blacklist application design.

the packet and takes the appropriate action. Such actions include forwarding or dropping the packet and programming flow entries in the switch. We discuss the message listener and the switch listener in greater detail in Section 10.4.3.

The flow entries depicted in the switch in Figure 10.3 show that DNS requests are always forwarded to the controller. If the destination IP address returned by DNS has already been checked, the application will program the appropriate high-priority flow entries. In our example, such entries are shown as the two flow entries matching 1.1.1.5 and 1.1.9.2. The reader should understand that the flows for destination 1.1.1.5 and 1.1.9.2 were programmed earlier by the controller when packets for these two destinations were first processed. If the IP address on the incoming packet has not been seen earlier or has aged out, there will be no flow entry for that address and the IP request will be forwarded to the controller for blacklist processing.

Source code for the listener, packet handler, and flow manager components is provided in Appendix B. Code snippets will also be provided throughout the following analysis. The reader should refer to the appendix to place these snippets in context. We include in the appendix the SDN-relevant source code for Blacklist, in part to illustrate that a nontrivial network security application can be written in just 15 pages of Java, evidence of the relative ease and power of developing applications in the Open SDN model.

We now delve more deeply into the components of Blacklist.

10.4.3 **Blacklist: Listeners**

Our Blacklist application uses two important listeners. The first is the *SwitchListener*. The *SwitchListener* receives notifications from the controller whenever a switch is discovered or when it has disconnected. The *SwitchListener* also receives notifications when the switch's port configuration has changed (e.g., when ports have been added or removed). The exact frequency of such notifications depends on the particular switch's environment. Once steady-state operation has been achieved, however, it is unlikely that these notifications will occur more often than once every few seconds. The two most common SDN switch-type entities in this example are wired switches and APs.

When the application first learns that a switch has been discovered, the application programs it with two default rules: one to forward DNS requests to the controller and the other a low-priority flow entry to forward IP packets to the controller. For nonblacklisted IP addresses, this low-priority IP flow entry will be matched the first time that IP destination is seen. Subsequently, a higher-priority flow is added that allows packets matching that description to be forwarded normally, without involving the controller. If that flow entry idle timer expires, the flow entry is removed and the next packet sent to that IP address will once again be forwarded to the controller, starting this process anew.

Examination of the actual source code in Appendix B reveals that we add one other default flow for ARP requests, which are forwarded normally. ARP requests pose no threat from the perspective of the Blacklist application. So, by default, all new switches will have three flows: DNS (send to controller), IP (send to controller), and ARP (forward NORMAL).

The other important listener is the *MessageListener*. The most important of the various methods in this listener is the *receive* method. The receive method is called whenever a packet arrives at the controller. The method then passes the packet to the Blacklist SDN application. When the *PACKET_IN* message is received, our application invokes the *PacketHandler* to process the packet. In Section 5.3.5 we described how the switch uses the *PACKET_IN* message to communicate to the controller.

The code for the *MessageListener* module is listed in Appendix B.1. We examine part of that module here. First, we see that our class is derived from the Floodlight interface called *IOFMessageListener*:

```
//--------------------------------------------------------------
public class MessageListener implements IOFMessageListener
//--------------------------------------------------------------
```

Next is the registration of our *MessageListener* method:

```
//--------------------------------------------------------------
public void startUp()
{
    // Register class as MessageListener for PACKET_IN messages.
    mProvider.addOFMessageListener( OFType.PACKET_IN, this );
}
//--------------------------------------------------------------
```

This code makes a call into Floodlight (*mProvider*), invoking the method *addOFMessageListener* and telling it we want to listen for events of type *PACKET_IN*.

The next piece of code to consider is our *receive* method, which we provide to Floodlight to call when a packet is received by the listener. We are passed a reference to the switch, the message itself, and the Floodlight context reference.

```
//--------------------------------------------------------------
@Override
public Command receive( final IOFSwitch         ofSwitch,
                        final OFMessage         msg,
                        final FloodlightContext context )
{
    switch( msg.getType() )
    {
    case PACKET_IN:  // Handle incoming packets here

       // Create packethandler object for receiving packet in
       PacketHandler ph = new PacketHandler( ofSwitch,
                                             msg, context);

       // Invoke processPacket() method of our packet handler
       // and return the value returned to us by processPacket
       return ph.processPacket();

    default: break;  // If not a PACKET_IN, just return

    }

    return Command.CONTINUE;
}
//-------------------------------------------------------------
```

In the preceding code, we see a Java *switch* statement based on the message type. We are only interested in messages of type *PACKET_IN*. Upon receipt of that message, we create a *PacketHandler* object to process this incoming packet. We then invoke a method within that new object to process the packet.

10.4.4 Blacklist: Packet Handlers

In our application there is one *PacketHandler* class that handles all types of incoming packets. Recall that our flow entries on the switches have been set up to forward DNS requests and unmatched IP packets to the controller. The *processPacket* method of the *PacketHandler* shown in Figure 10.3 is called to handle these incoming packets, where it performs the following steps:

1. **IP packet.** Examine the destination IP address and, if it is on the blacklist, drop the packet. *Note that we do not attempt to explicitly program a DROP flow for this bad address. If another attempt is made to access this blacklisted address, the packet will be diverted to the controller again. This is a conscious design decision on our part, though there may be arguments that support the approach of programming blacklisted flows explicitly.*
2. **DNS request.** If the IP packet is a DNS request, examine the hostnames, and if any are on the blacklist, drop the packet.
3. **Forward packet.** If the packet has not been dropped, forward it.
4. **Create IP flow.** If it is not a DNS request, create a flow for this destination IP address. Note that there is no need to set a flow for the destination IP address of the DNS server, because all DNS requests are caught by the higher-priority DNS flow and sent to the controller.

In our Blacklist application, the *Create IP flow* step above will actually invoke a method in the *FlowMgr* class that has a method for creating the Floodlight OpenFlow objects that are required to program flow entries on the switches.

The source code for the *PacketHandler* class can be found in Appendix B.2. We describe some of the more important portions of that code in the following paragraphs.

PacketHandler is a *Plain Old Java Object* (POJO), which means that it does not inherit from any Floodlight superclass. The object's constructor stores the objects passed to it, including (1) references to the switch that sent the packet, (2) the message itself (the packet data), and (3) the Floodlight context, which is necessary for telling Floodlight what to do with the packet once we have finished processing it.

The first piece of code is the beginning of the *processPacket* method, which was called by the *MessageListener*:

```
//-------------------------------------------------------------
public Command processPacket()
{
    // First, get the OFMatch object from the incoming packet
    final OFMatch ofMatch = new OFMatch();
    ofMatch.loadFromPacket( mPacketIn.getPacketData(),
                            mPacketIn.getInPort()     );

//-------------------------------------------------------------
```

This is our first look at the *OFMatch* object provided by Floodlight, which derives from the same OpenFlow Java source code that can be found in Beacon and, thus, in OpenDaylight. In this code, we create an *OFMatch* object, and then we load the object, from the packet that we have received. This allows us to use our own *ofMatch* variable to examine the header portion of the packet we just received.

In different parts of *processPacket* we look at various portions of the incoming packet header. The first example uses *ofMatch.getNetworkDestination()* to get the destination IP address, which is used to compare against our IP blacklist:

```
//-------------------------------------------------------------
IPv4.toIPv4AddressBytes( ofMatch.getNetworkDestination() ) );
//-------------------------------------------------------------
```

The next example determines whether this is a DNS request:

```
//-------------------------------------------------------------
if( ofMatch.getNetworkProtocol()       == IPv4.PROTOCOL_UDP &&
    ofMatch.getTransportDestination() == DNS_QUERY_DEST_PORT )
//-------------------------------------------------------------
```

Much of the other code in the module in Appendix B.2 deals with doing the work of the Blacklist application. This includes checking the destination IP address, parsing the DNS request payload, and checking those names as well. The most relevant SDN-related code is the creation of an action list to hand to the controller, which it will use to instruct the switch as to what to do with the packet that was received. In *forwardPacket()* we create the action list with the following code:

```
//-------------------------------------------------------------
final List<OFAction> actions = new ArrayList<OFAction>();
actions.add( new OFActionOutput( outputPort ) );
//-------------------------------------------------------------
```

This code creates an empty list of actions, then adds one action to the list. That action is a Floodlight object called *OFActionOutput*, which instructs the switch to send the packet out the given port.

The *FlowManager* object is responsible for the actual interaction with Floodlight to perform both the sending of the packet out of the port as well as the possible creation of the new flow entry. We describe this object next.

10.4.5 Blacklist: Flow Management

The *FlowMgr* class deals with the details of the OpenFlow classes that are part of the Floodlight controller. These classes simplify the process of passing OpenFlow details to the controller when our application sends responses to the OpenFlow switches. These classes make it possible to use simpler Java class structures for communicating information to and from the controller. However, there is considerable detail that must be accounted for in these APIs, especially compared to their RESTful counterparts.

We now examine some of the more important Java objects that are used to convey information about flows to and from the controller:

- ***IOFSwitch.*** This interface represents an OpenFlow switch and, as such, it contains information about the switch, such as ports, port names, IP addresses, etc. The *IOFSwitch* object is passed to the *MessageListener* when a packet is received, and it is used to invoke the *write* method when an OpenFlow packet is sent back to the switch in response to the request that was received.
- ***OFMatch.*** This object is part of the *PACKET_IN* message that is received from the controller that was originated by the switch. It is also used in the response that the application will send to the switch via the controller to establish a new flow entry. When used with the *PACKET_IN* message, the match fields represent the header fields of the incoming packet. When used with a reply to program a flow entry, they represent the fields to be matched and may include wildcards.
- ***OFPacketIn and OFPacketOut.*** These objects hold the actual packets received and sent, including payloads.
- ***OFAction.*** This object is used to send actions to the switch, such as NORMAL forwarding or FLOOD. The switch is passed a list of these actions so that multiple items such as *modify destination network address* and then NORMAL forwarding may be concatenated.

Code for the *FlowMgr* class can be found in Appendix B.3. Here we walk the reader through some important logic from *FlowMgr* using the code snippets below. One of the methods in *FlowMgr* is *setDefaultFlows*, which is called from the *SwitchListener* class whenever a new switch is discovered by Floodlight.

```
//-------------------------------------------------------------
public void setDefaultFlows(final IOFSwitch ofSwitch)
{
    // Set the intitial 'static' or 'proactive' flows
    setDnsQueryFlow( ofSwitch );
    setIpFlow( ofSwitch );
    setArpFlow( ofSwitch );
}
//-------------------------------------------------------------
```

As an example of these internal methods, consider the code to set the DNS default flow. In the following code, we see *OFMatch* and *OFActionOutput*, which were introduced in Section 10.4.4. The *OFMatch* match object is set to match packets that are Ethernet, UDP, and destined for the DNS UDP port.

```
//--------------------------------------------------------------
private void setDnsQueryFlow( final IOFSwitch ofSwitch )
{
    // Create match object to only match DNS requests
    OFMatch ofMatch = new OFMatch();
    ofMatch.setWildcards( allExclude( OFPFW_TP_DST,
                                      OFPFW_NW_PROTO,
                                      OFPFW_DL_TYPE ) )
               .setDataLayerType( Ethernet.TYPE_IPv4 )
               .setNetworkProtocol( IPv4.PROTOCOL_UDP )
               .setTransportDestination( DNS_QUERY_DEST_PORT );

    // Create output action to forward to controller.
    OFActionOutput ofAction  = new OFActionOutput(
                             OFPort.OFPP_CONTROLLER.getValue(),
                             (short) 65535                   );

    // Create our action list and add this action to it
    List<OFAction> ofActions = new ArrayList<OFAction>();
    ofActions.add(ofAction);

    sendFlowModMessage( ofSwitch,
                        OFFlowMod.OFPFC_ADD,
                        ofMatch,
                        ofActions,
                        PRIORITY_DNS_PACKETS,
                        NO_IDLE_TIMEOUT,
                        BUFFER_ID_NONE );
}
//--------------------------------------------------------------
```

The flow we are setting instructs the switch to forward the packet to the controller by setting the output port to *OFPort.OFPP_CONTROLLER.getValue()*. Then we call the internal method *sendFlowModMessage*. The code for that method is as follows:

```
//--------------------------------------------------------------
private void sendFlowModMessage( final IOFSwitch      ofSwitch,
                                 final short          command,
                                 final OFMatch        ofMatch,
                                 final List<OFAction> actions,
                                 final short          priority,
                                 final short          idleTimeout,
                                 final int            bufferId )
{
    // Get a flow modification message from factory.
    final OFFlowMod ofm = (OFFlowMod) mProvider
```

```
                                            .getOFMessageFactory()
                                            .getMessage(OFType.FLOW_MOD);

        // Set our new flow mod object with the values that have
        // been passed to us.
        ofm.setCommand( command ).setIdleTimeout( idleTimeout )
                                 .setPriority( priority )
                                 .setMatch( ofMatch.clone() )
                                 .setBufferId( bufferId )
                                 .setOutPort( OFPort.OFPP_NONE )
                                 .setActions( actions )
                                 .setXid( ofSwitch
                                    .getNextTransactionId() );

        // Calculate the length of the request, and set it.
        int actionsLength = 0;
        for( final OFAction action : actions )
           { actionsLength += action.getLengthU(); }
        ofm.setLengthU(OFFlowMod.MINIMUM_LENGTH + actionsLength);

        // Now send out the flow mod message we have created.
        try
        {
            ofSwitch.write( ofm, null );
            ofSwitch.flush();
        }
        catch (final IOException e)
        {
            // Handle errors with the request
        }
    }
    //--------------------------------------------------------------
```

Items to note in the code above are:

- The *OFFlowMod* object is created and used to hold the information regarding how to construct the flow, including items such as idle timeout, priority, match field, and actions.
- The length must still be calculated by this method. The length could be provided by the OpenFlow package, but as of this writing it is not.
- The switch is sent the flow modification message using the *write* method.

There is additional code available for the reader to examine in Appendix B.3. For example, consider the FlowMgr methods *sendPacketOut* and *createDataStreamFlow* for more examples of interacting with the Floodlight controller.

The next section looks at some of the open source and commercial controllers that are available today and examines the APIs that they make available to the SDN application developer.

10.5 Background on Controllers

The Beacon controller is truly a seminal controller in that much of the basic OpenFlow controller code in Floodlight and OpenDaylight was derived directly from Beacon. As we pointed out in Section 9.1.1, Beacon itself was based on early work performed at Stanford by David Erickson and Rob Sherwood. Both Beacon and Floodlight are based on OpenFlowJ for the core Java OpenFlow implementation. Floodlight is maintained by engineers from Big Switch. The distributions of Beacon and Floodlight are now distinct and different applications are packaged with each, but Beacon remains close enough to Floodlight that we will not elaborate on it further at the detailed implementation level presented here.

We should point out that a number of other OpenFlow controllers are available. For example, the NOX controller has been used in a considerable number of research projects. We provide basic information about a number of open source controller alternatives in Section 11.6. The subset of controllers that we present in this chapter represents those controllers that we feel an application developer today would be most likely to use.

It is also important to note that as of this writing there is no standard for the northbound API to the controller. The Java APIs that we have documented in the preceding sections may not be exposed to the application programmers in every one of the controllers that we present in this chapter, including those based on the original Beacon code. There may be a higher-level abstraction that is exposed that is not so tightly coupled to the OpenFlow protocol. This appears to be the direction in which OpenDaylight may be headed. The application API is an interface that may translate to a number of different southbound APIs, OpenFlow being just one of them.

10.6 Using the Floodlight Controller

Floodlight is arguably the most prevalent open source SDN controller available today. It has a number of packages denoted by *org.openflow*, which make up some of the core functionality. There are also many packages from Floodlight that implement components such as the web GUI, the device manager, link discovery, a learning switch, a load balancer, and even a virtual network package.

As we said earlier, the Floodlight controller is freely downloadable. The code and packages may be viewed, and it is fairly straightforward to begin experimenting with it. Once it is configured and running it will take control of the OpenFlow switches that have been configured to use it as their controller. By default it will operate its Learning Switch SDN application, setting flow entries on switches to forward packets across a layer two network, even one with loops and a mesh topology.

Floodlight provides both reactive Java APIs and proactive RESTful APIs. We look at these two classes of APIs next.

10.6.1 Java APIs

In Section 10.4 we provided detailed examples of the Floodlight Java APIs. There are APIs to register listeners for various events, such as switch and device changes, and to receive incoming packets. There are the OpenFlow primitive classes and interfaces such as *IOFSwitch*, *OFMatch*, *OFAction*, and others. These APIs are fundamentally the same in Floodlight, Beacon, and OpenDaylight. The Floodlight GUI is extensible and can be used to incorporate one's own application into the Floodlight GUI framework.

10.6.2 RESTful APIs

Proactive Floodlight applications can use the RESTful APIs to gather data about the network, to program flows in the network, and even to create virtual networks and manipulate a firewall. The following list of RESTful APIs for Floodlight provides an indication of the functionality available:

- **Switch** provides APIs to retrieve statistics for all or for a single switch.
- **Controller** provides APIs to retrieve a list of all switches being managed by this controller.
- **Counter** provides APIs to retrieve a list of traffic counters per switch.
- **Memory** provides APIs to retrieve the current memory usage by the controller.
- **Topology** provides APIs to retrieve a list of interswitch links, external switch links, and switch clusters.
- **Device** provides APIs to retrieve a list of all devices.
- **StaticFlowPusher** provides APIs to add or remove static flows from switches and to retrieve a list of static flows on a switch.
- **Network Service (aka Virtual Network)** provides APIs to create, modify, or remove a virtual network, to attach or remove a host from a virtual network, and to retrieve a list of virtual networks and their hosts.

The extensible RESTful API uses the Restlet framework [3] and includes a small web server that allows external applications and GUIs to communicate with the SDN application. Adding a RESTful interface to the SDN application is straightforward:

- Add the RESTful API and the associated class for handling calls to the API by instantiating a *RestRoutable* object, identifying the RESTful URL and the handling class.
- Create the RESTful resource handler class by extending the *ServerResource* class.
- Use Jackson [4] or a similar tool to translate back and forth between the data portions (often in JSON [5] format) of the RESTful requests and replies.

Since the resource handler is part of the Floodlight application, it has access to the rest of the application and its data. In addition, there are a number of Floodlight modules that can be used for various purposes. For example, Floodlight has information about the OpenFlow switches and devices (which are hosts or end nodes in the Floodlight jargon). It is possible to query the switch to find out about its ports and to query the devices about their attachment point (switch and port) to the network.

A fine example of how to write a Floodlight application is provided in [6], which should be of assistance in creating one's own application to control a network of OpenFlow switches.

10.7 Using the OpenDaylight Controller

OpenDaylight is a project intended to produce a common controller to be used as a basis for SDN applications for vendors and enterprises alike. At the time of this writing, the OpenDaylight controller is in prerelease form. The OpenFlow functionality in the OpenDaylight controller is derived from the Beacon code. The lion's share of the controller was contributed to the project by Cisco. This can be confirmed by inspection of the packages documented in [7], which reveals a small number of the *org.openflow* packages discussed earlier and a large number of *org.opendaylight.controller* packages, most of which have come from Cisco.

There is a Java API in the controller for performing both OpenFlow and non-OpenFlow management of the network and the switches. Like Floodlight, there is also a RESTful API available.

10.7.1 Java APIs

The OpenFlow functionality provided by the OpenFlow packages in the Floodlight, Beacon, and OpenDaylight controllers is essentially the same. Applications that run on Floodlight run with little modification on Beacon and OpenDaylight.

OpenDaylight provides an additional abstraction called the *Service Abstraction Layer* (SAL). SAL purportedly allows network programmers to write applications similarly to the way they would be written calling the OpenFlow Java APIs, with the advantage that SAL can translate the API requests into either OpenFlow or another non-OpenFlow API mechanism that may be available on the target switch. It should thus be possible to write applications that work with both OpenFlow and non-OpenFlow switches.

10.7.2 RESTful APIs

The RESTful APIs for OpenDaylight provide functionality that is similar to that provided by Floodlight and other controllers. Here is a summary of the categories of API:

- **Topology** provides APIs to retrieve the interswitch links in the network.
- **Host Tracker** provides APIs to retrieve the hosts (end nodes) in the network.
- **Flow Programmer** provides APIs for reading and writing flows on specific switches in the network.
- **Static Routing** provides APIs for reading and writing static routes (e.g., next-hop rules) on switches in the network.
- **Statistics** provides APIs for retrieving statistics for flows, ports, tables, and switches.
- **Subnets** provides APIs for retrieving information about subnets.
- **Switch Manager** provides APIs for retrieving information about switches.

10.8 Using the Cisco XNC Controller

We have mentioned that the bulk of the OpenDaylight controller implementation has come from Cisco. The Cisco *Extensible Network Controller* (XNC) was originally known as the Cisco ONE controller. The XNC controller is Cisco's commercial SDN controller, and there is considerable overlap between XNC and OpenDaylight. For example, the user interfaces for OpenDaylight and for Cisco XNC are very similar. However, there is some functionality that is part of the Cisco XNC controller that is not available with OpenDaylight. We list those next.

10.8.1 RESTful APIs

The APIs available in OpenDaylight are also available in XNC. In addition, two categories of RESTful APIs available with Cisco's XNC controller that are not part of OpenDaylight are:

- **Slices.** Slices addresses the need for administrative domains that provide different views, depending on the administrator that is using the controller. This allows the network to be partitioned into separate domains based on the administrator's role or based on the network entities that are under the control of specific administrators.

- **Topology Independent Forwarding (TIF).** The TIF functionality in the Cisco controller allows an administrator to create paths between two endpoints without taking into consideration the intermediate hops between one end and the other.

10.9 Using the Hewlett-Packard Controller

HP is an important vendor to SDN not only because of its networking products but also because it has a large share of the server market for data centers. HP has a major portion of the services market as well, expanding significantly with its purchase of Electronic Data Systems (EDS) in 2008. HP will likely have a significant SDN presence due to SDN's projected impact in data centers, where HP servers and services are prevalent.

HP's controller is expected to be released in late 2013. Early indications of the controller are that it will have both Java and RESTful APIs.

10.9.1 Java APIs

The HP Java API has diverged from what was prevalent in the earlier controllers (Beacon, Floodlight, etc.), in order to provide support for OpenFlow 1.3, including multiple tables, TLVs, and the like and to provide a richer software development environment.

10.9.2 RESTful APIs

The HP RESTful API includes functionality such as the following:

- **Nodes** retrieves information about end nodes in the networks.
- **Datapaths** retrieves information about switches in the network and for setting flows on these devices.
- **Clusters** retrieves information about groups of switches that form a broadcast domain.
- **Regions** retrieves information about groups of clusters.
- **Forward Path** returns the shortest path between a source and destination switch.
- **Teams** retrieves information about groups of controllers working in concert for high availability (HA).

10.10 Switch Considerations

One of the most important considerations in building an SDN application is the nature of the switches that are to be controlled by the application. Some of the relevant questions are:

- Do all the switches support OpenFlow? What version of OpenFlow do they support? How much of that version do they in fact support?
- Is there a mix of OpenFlow and non-OpenFlow switches? If so, will some non-OpenFlow mechanism be used to provide the best simulation of an SDN environment? For example, do the non-OpenFlow switches support some type of SDN API that allows a controller to configure flows in the switch?

- For OpenFlow-supporting switches, are they hardware or software switches, or both? If both, what differences in behavior and support will need to be considered?
- For hardware OpenFlow switches, what are their hardware flow-table sizes? If the switches are capable of handling flow table overflow in software, at what point are they forced to begin processing flows in software, and what is the performance impact of doing so? Is it even possible to process flows in software? What feature limitations are introduced because of the hardware?
- What is the mechanism by which the switch-controller communications channel is secured? Is this the responsibility of the network programmer?

Hardware switches vary in their flow capacity, feature limitations, and performance, and these should be considerations in creating an SDN application. The major issue regarding flow capacity is how many flow entries the hardware ASIC is capable of holding. Some switches support fewer than 1,000 flow entries, whereas other, such as the NEC PF5240 [8], support greater than 150,000. Note that for some switches the maximum number of flows depends on the nature of the flows supported. For example, the NEC PF5820 switch [9] can support 80,000 layer-two-only flows but only 750 full 12-tuple flows. (We described matching against the basic 12-tuple of input fields in Section 5.3.3.) A layer two flow entry only requires matching on the MAC address, so the amount of logic and memory occupied per flow entry is much less. Some applications, such as a traffic prioritization application or a TCP port-specific application, may use relatively few flow table entries. Others, such as an access control application or a per-user application, will be much more hungry for flow entries. The topological placement of the application logic may have a bearing on the maximum number of flow entries as well. For example, an access control application being applied in a switch or AP at the user edge of the network will require far fewer flow entries per device than the same application controlling an upstream or core switch.

Hardware implementations often have feature limitations because, when the OpenFlow specification is mapped to the capabilities of the hardware, generally some features cannot be supported. For example, some ASICs are not able to perform NORMAL forwarding on the same packet that has had its header modified in some way. Although it is true that the list of features supported by the device can be learned via OpenFlow *Features Request* and *Features Reply* messages (see Section 5.3.5), this does not tell you the specific nuances and limitations that may be present in any one switch's implementation of each feature. Unfortunately, this information must often be discovered via trial and error. Information about these issues may be available on the Internet sites of interested SDN developers. The application developer must be aware of these types of limitations before designing a solution.

Some hardware switches will implement some functions in hardware and others in software. It is wise to understand this distinction so that an application does not cause unexpected performance degradation by unknowingly manipulating flows such that the switch processes packets in software.

In general, software implementations of OpenFlow are able to implement the full set of functionality defined by the OpenFlow specification and can implement flow tables in standard computer memory. Thus, the feature limitations and table size issues we have described above are not a major concern for application developers. However, performance will still be a consideration, since implementations in hardware will generally be faster.

We now provide a number of high-level designs for some additional simple applications relevant to several different environments. These include the data center, the campus, and service provider environments. We restrict ourselves to high-level design issues in the following sections. It is our hope that the reader will be able to extend the detailed source code analyis performed for the Blacklist application in Section 10.4 and apply those learnings to the cases we cover here.

10.11 Creating Network Virtualization Tunnels

As explained in Section 7.3, data center multitenancy, MAC-table overflow, and VLAN exhaustion are being addressed with new technology making use of tunnels, the most common being the MAC-in-IP tunnels provided by protocols such as VXLAN, NVGRE, and STT. Figure 10.4 shows a possible software application design for creating and managing overlay tunnels in such an environment. We see in the figure the switch, device, and message listeners commonly associated with reactive applications. This reactive application runs on the controller and listens for IP packets destined for new IP destination addresses for which no flow entries exist. When the message listener receives such an unmatched packet, the application's tunnel manager is consulted to determine which endpoint switch is responsible for this destination host's IP address.

The information about the hosts and their tunnel endpoints is maintained by the application in the associated host and tunnel databases. The information itself can be gathered either outside the domain of OpenFlow or it could be generated by observing source IP addresses at every switch. Whichever way this is done, the information needs to be available to the application.

When a request is processed, the tunnel information is retrieved from the database and, if the tunnel does not already exist, it will be created. Note that there may be other hosts communicating between these same two switches acting as tunnel endpoints, so there may be a tunnel already in existence. The

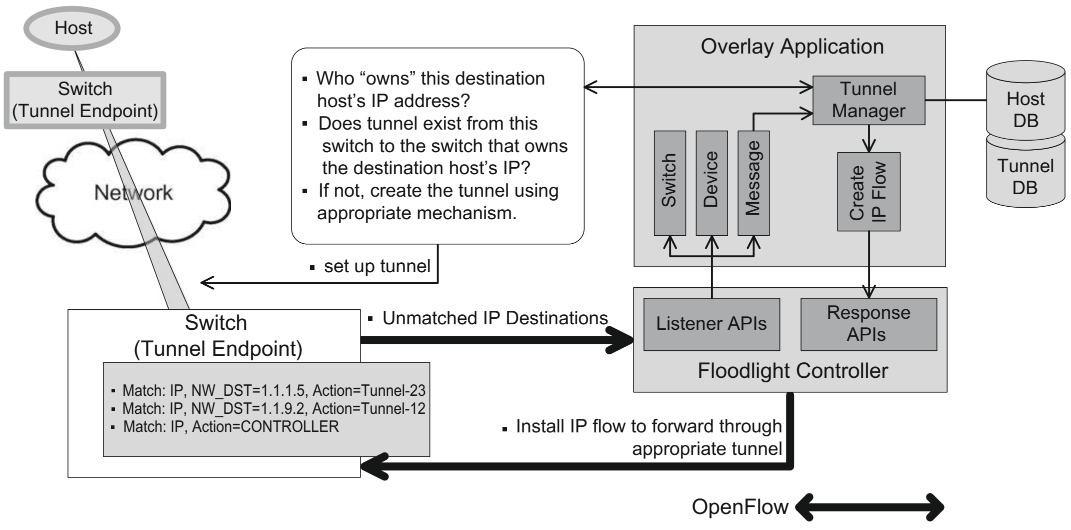

Host
Switch (Tunnel Endpoint)
Network
▪ Who "owns" this destination host's IP address?
▪ Does tunnel exist from this switch to the switch that owns the destination host's IP?
▪ If not, create the tunnel using appropriate mechanism.
▪ set up tunnel
Overlay Application
Tunnel Manager
Switch
Device
Message
Create IP Flow
Host DB
Tunnel DB
Switch (Tunnel Endpoint)
▪ Match: IP, NW_DST=1.1.1.5, Action=Tunnel-23
▪ Match: IP, NW_DST=1.1.9.2, Action=Tunnel-12
▪ Match: IP, Action=CONTROLLER
▪ Unmatched IP Destinations
Listener APIs
Response APIs
Floodlight Controller
▪ Install IP flow to forward through appropriate tunnel
OpenFlow

FIGURE 10.4

Tunnels application design.

tunnel creation can be done in a number of ways, all of which are outside the scope of the OpenFlow specification. The tunnel creation is performed on the OpenFlow device and a mapping is established between that tunnel and a virtual port, which is how tunnels are represented in flow entries. These methods are often proprietary and specific to a particular vendor. For the reader interested in configuring VXLAN or NVGRE tunnels on the OVS switch, a useful guide can be found in [10].

Once the tunnel is either created or identified, the tunnel application can set up flows on the local switch to direct traffic heading toward that destination IP address into the appropriate tunnel. Once the virtual port exists, it is straightforward to create flow entries such as those shown in the figure, where we see the tunneled packets directed to virtual ports TUNNEL-23 and TUNNEL-12. The complementary flow will need to be set on the switch hosting the remote tunnel endpoint as well.

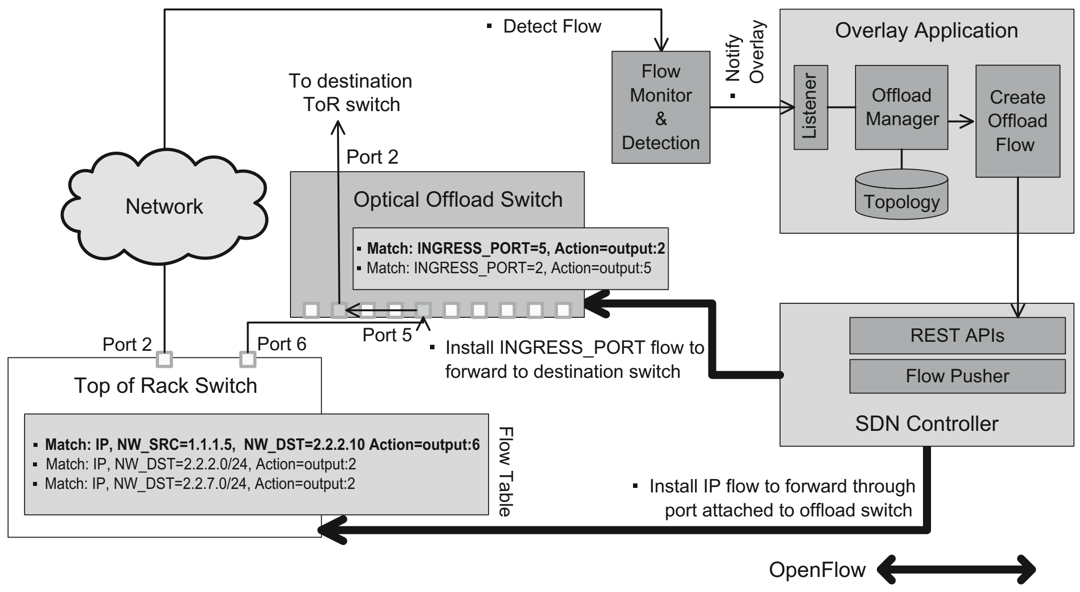
Note that the preceding high-level design is unicast-centric. It does not explain the various mechanisms for dealing with broadcast packets such as ARPs. These could be dealt with by a global ARP service provided by the controller. The idea behind such a controller-based service is that the switch learns IP-MAC relationships by listening to ARP replies and, when it has already cached the answer to an ARP request, it does not forward the ARP request but instead replies automatically, thus suppressing ARPs from using up broadcast bandwidth. This process is fairly easy to embody in the SDN controller because it learns the IP and MAC address of unmatched packets, which by default are forwarded to the controller. Thus, the controller is often able to trap ARP requests and provide the reply without propagating the broadcast.

There are also ways of handling these requests via VXLAN, NVGRE, and STT protocols. The methods of dealing with broadcasts and multicasts in the tunneling protocols are specific to each protocol and beyond the scope of this book. For details we refer the reader to the references we have previously provided for these three protocols.

10.12 Offloading Flows in the Data Center

One of the major problems in data centers is the elephant flows, which we introduced in Section 8.7.1. Such flows consume huge amounts of bandwidth, sometimes to the point of starving other flows of bandwidth. Figure 10.5 depicts a high-level design of an offload application. This proactive application uses RESTful APIs to communicate with the controller. The general overview of the operation of such an application is as follows:

- There is a flow monitor that monitors flows and detects the beginning of an elephant flow. (This is a nontrivial problem in itself. In Section 8.7.1 we provided simple examples as well as references for how elephant flows can be scheduled, predicted, or detected.)
- Once an elephant flow is detected, the flow monitor sends a notification message (possibly using the application's own REST API) to communicate details about the flow. This includes identifying the two endpoint ToR switches where the flow begins and ends as well as the IP addresses of the starting (1.1.1.5) and ending (2.2.2.10) hosts.
- The offload application has been previously configured with the knowledge of which ports on the ToR switches connect to the offload switch.
- Armed with this topology information, the offload application is able to set flows appropriately on the ToR switches and the offload switch.



Detect Flow
Notify Overlay
Overlay Application
To destination ToR switch
Flow Monitor & Detection
Listener
Offload Manager
Create Offload Flow
Topology
Port 2
Optical Offload Switch
Network
Match: INGRESS_PORT=5, Action=output:2
Match: INGRESS_PORT=2, Action=output:5
Port 5
Port 2
Port 6
Install INGRESS_PORT flow to forward to destination switch
REST APIs
Flow Pusher
SDN Controller
Top of Rack Switch
Match: IP, NW_SRC=1.1.1.5, NW_DST=2.2.2.10 Action=output:6
Match: IP, NW_DST=2.2.2.0/24, Action=output:2
Match: IP, NW_DST=2.2.7.0/24, Action=output:2
Flow Table
Install IP flow to forward through port attached to offload switch
OpenFlow

FIGURE 10.5

Offload application design.

- As shown in Figure 10.5, a new flow rule is added in this ToR switch. This is the flow shown in bold in the figure. It is important that this be a high-priority flow and match exactly on the source and destination host addresses. The next lower-priority flow entry matches on the *network* address 2.2.2.0/24, but, since that is a lower-priority entry, packets belonging to the elephant flow will match the higher-priority entry first, and those packets will be directed out port six to the optical offload switch. Note that traffic other than the elephant flow will match the lower-priority entries and will be transmitted via the normal port.
- On the optical offload switch, flow rules are programmed that connect the two ports that will be used to forward this elephant flow between the switches involved. For the sake of simplicity, we will assume that an elephant flow is unidirectional. If there is bulk traffic in both directions, the steps we describe here can be duplicated to instantiate a separate OTN flow in each direction.
- Once the elephant flow ceases, the offload application will be informed by the flow monitor and the special flows will be removed.

10.13 Access Control for the Campus

In Section 8.3.1 we introduced the idea of an SDN *network access control* (NAC) application for the campus. Figure 10.6 shows a high-level design of a campus NAC application. It should be apparent from the diagram that this application is a reactive application because it needs to be notified of unauthenticated

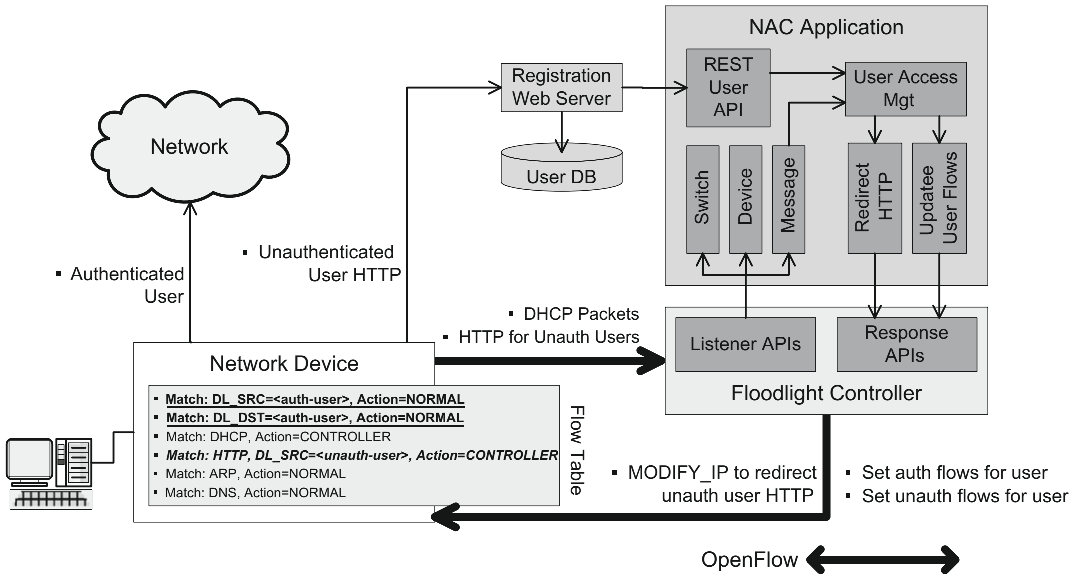

NAC Application
REST User API
User Access Mgt
Switch
Device
Message
Redirect HTTP
Updatee User Flows
Registration Web Server
User DB
Network
Unauthenticated User HTTP
Authenticated User
DHCP Packets
HTTP for Unauth Users
Listener APIs
Response APIs
Floodlight Controller
Network Device
Match: DL_SRC=<auth-user>, Action=NORMAL
Match: DL_DST=<auth-user>, Action=NORMAL
Match: DHCP, Action=CONTROLLER
Match: HTTP, DL_SRC=<unauth-user>, Action=CONTROLLER
Match: ARP, Action=NORMAL
Match: DNS, Action=NORMAL
Flow Table
MODIFY_IP to redirect unauth user HTTP
Set auth flows for user
Set unauth flows for user
OpenFlow

FIGURE 10.6

NAC application design.

users attaching to the network. The initial state of the switch is to have flows that allow ARP and DNS but that forward DHCP requests and responses to the controller. The DHCP responses will be used to build a database of MAC and IP addresses for the devices attaching to the network.

When the DHCP response is observed, if this is a new user, the NAC application adds the user to the database and the user's state is *unauthenticated*. The application then programs a flow entry in the switch based on the user's MAC address. This flow in the non-underlined italics text in Figure 10.6 is programmed to forward the user's HTTP requests to the controller. Thus, when a user opens a browser, in typical captive portal behavior, the application will cause the HTTP request to be forwarded to the captive portal web server where user authentication takes place. This is easily achieved by modifying the destination IP address (and possibly the destination MAC address as well) of the request so that when the switch forwards the packet, it will go to the captive portal *registration web server* shown in Figure 10.6 rather than to the intended destination.

Normal captive portal behavior is to force an HTTP redirect, which sends the user's browser to the desired captive portal registration, authentication, or guest login page. The controller will have programmed the switch with flows that will shunt HTTP traffic to the captive portal. No other traffic type needs to be forwarded, since the authentication takes place via the user's web browser. Thus the user engages in an authentication exchange with the captive portal. This could entail payment, registration, or some related mechanism. When complete, the captive portal informs the NAC application that the user is authenticated. Note that the registration server uses the application's RESTful APIs to notify the NAC application of changes in the user's status. At this point the application removes the nonunderlined italics flow entry shown in Figure 10.6 and replaces it with the two bold underlined flow entries, thus allowing the user unfettered access to the network. When the user later connects to the network, if he is still registered, he will be allowed into the network without needing to go through the registration/authentication step unless these flow entries have timed out.

The reader should note that in this example and in the accompanying Figure 10.6, we stop part-way through the complicated process of HTTP redirection. The full process entails sending an HTTP 302 message back to the user to redirect the user's browser to the registration server. These details are not germane to our example, so we exclude them here.

10.14 Traffic Engineering for Service Providers

In Section 8.2.1 we explained how traffic engineering is a key tool for service providers to optimize their return on investment in the large networks they operate for their customers. Figure 10.7 shows a simplified design for a proactive traffic engineering application that takes traffic monitoring information into account. As shown in the figure, the flow rules in the router(s) are very simple. Likewise, the design of the application is simple. The complexity resides inside the path computation manager component doing the path calculations and optimizing traffic over the available links.

This problem can also be addressed using reactive application design. Figure 10.8 reveals a high-level design of such an approach. The key difference between this design and the proactive one lies in the flow tables on the routers, which feature a final flow entry that forwards unmatched packets to the controller. In this solution, initially there are no flows on the switch except for the CONTROLLER flow. The

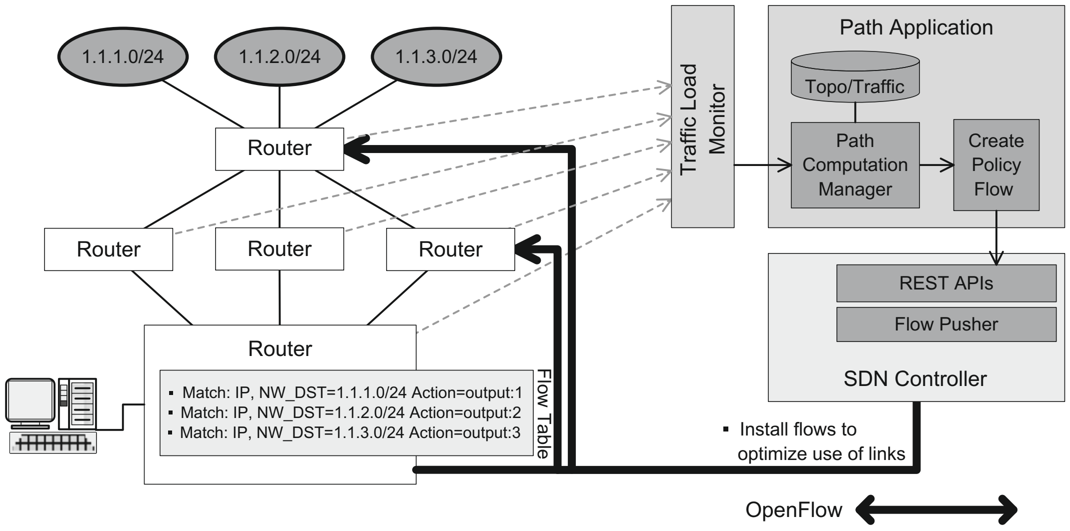


FIGURE 10.7

Traffic engineering proactive application design.

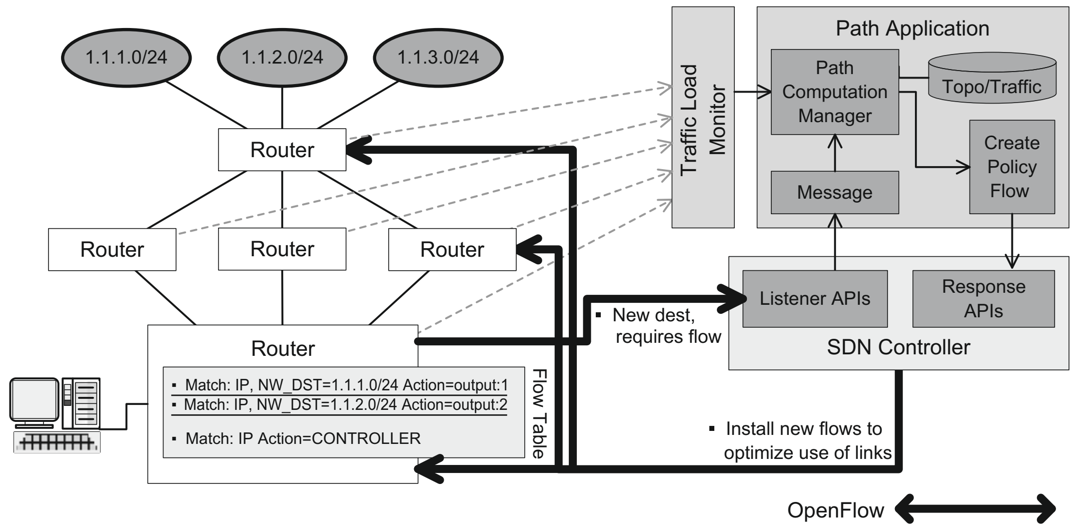


FIGURE 10.8

Traffic engineering reactive application design.

routers are not pre-configured with any specific flows other than to forward unmatched packets to the controller, as is typical for a reactive application. When these packets are received at the controller, the application's listener passes them to the path control manager. The path control manager then calculates the best path to the destination, given the current traffic loads. Thus the flows are established in an on-demand manner. When traffic to that destination eventually stops, the flow is removed. The flow could be removed implicitly by aging out, or an external traffic-monitoring function could explicitly inform the application that the flow has stopped and then the flow could be deleted by the application.

10.15 Conclusion

In this chapter we bridged the OpenFlow specifics of Chapter 5 with the conceptual use cases of Chapters 7 and 8, grounded with concrete design and programming examples. Our examples are all based on applications of Open SDN. Certainly, network programmers familiar with SDN via APIs or SDN via overlays would be able to use their respective technologies to address some of the scenarios presented in this chapter. However, it is unlikely, in our opinion, that those technologies would be so easily and simply adapted to the full spectrum of applications presented in this chapter. Having said that, the viability of a technology depends not only on its extensibility from a programmer's perspective but on other factors as well. Such issues include (1) the cost, sophistication, and availability of ready-made solutions, (2) the market forces that push and pull on various alternatives, and (3) how rapidly needed future innovations are likely to emerge from different alternatives. We spend the final three chapters of this book addressing these three aspects of SDN.

References

[1] Project floodlight download. Project floodlight. Retrieved from <www.projectfloodlight.org/download>.
[2] Erikson D. Beacon. OpenFlow @ Stanford; February 2013. Retrieved from <openflow.stanford.edu/display/Beacon/Home>.
[3] The leading web API framework for Java. Restlet framework. Retrieved from <restlet.org>.
[4] Jackson high-performance JSON processor. Retrieved from <jackson.codehaus.org>.
[5] Introducing JSON. Retrieved from <www.json.org>.
[6] Wang K. How to write a module. Project Floodlight; December 13, 2013. Retrieved from <docs.projectfloodlight.org/display/floodlightcontroller/How+to+Write+a+Module>.
[7] OpenDaylight controller: Java API reference. OpenDaylight. Retrieved from <wiki.opendaylight.org/view/OpenDaylight_Controller:Java_API_Reference>.
[8] ProgrammableFlow PF5240 switch. NEC Datasheet. Retrieved from <www.necam.com/SDN>.
[9] ProgrammableFlow PF5820 switch. NEC Datasheet. Retrieved from <www.necam.com/SDN>.
[10] Salisbury B. Configuring VXLAN and GRE tunnels on OpenvSwitch. NetworkStatic; July 2, 2012. Retrieved from <networkstatic.net/configuring-vxlan-and-gre-tunnels-on-openvswitch>.

This page is intentionally left blank

CHAPTER

SDN Open Source

11

In Chapter 5 we presented an overview of OpenFlow, ostensibly the single most important and fully open component of the SDN framework. The OpenFlow standard is developed by the *Open Networking Foundation* (ONF); however, the ONF does not provide a working implementation of either the switch or the controller. In this chapter we provide an inventory of a number of open source components that are available for use in research, including source code for OpenFlow itself. Some of these are available under licenses that make them suitable for commerical exploitation. Many of these initiatives will speed the standardization and rate of adoption of SDN. This chapter provides a survey of these initiatives and explains their relationship and roles within the broader SDN theme.

We have noted in earlier chapters that accepted use of the term SDN has grown well beyond its original scope. Indeed, the definition of SDN has blurred into most of the current work on network virtualization. Much of this work, though, particularly in Open SDN, has occurred in the academic world. As such, there is a significant library of open source available for those who want to experiment or build commercial products in the areas of SDN and network virtualization. Beyond academic contributions, many companies are contributing open source to SDN projects. Some examples that were mentioned in earlier chapters are Big Switch's Indigo, VMware's *Open vSwitch* (OVS), and the many companies cited in Section 9.9.2 contributing to OpenDaylight. This survey of SDN-related open source projects was greatly aided by the compilations already performed by SDN Central [3] and others [16,12].

In Figure 11.1 we map this wide selection of open source to the various network components. The major components, from the bottom up, are:

- OpenFlow device
- OpenFlow controller
- Applications
- Network virtualization
- Test and simulation

The last category, test and simulation, is shown to the side of the diagram because it does not fit neatly into a bottom-to-top hierarchy. For each of these categories we show in the callout ovals the notable available open source that contributes to that category. In the sections that follow we provide an inventory of this software. Some of these have surfaced as the basis for commercial deployments of network virtualization and SDN, and we explore that open source product in greater depth. As we move through the different network components in Sections 11.4–11.10, we encourage the reader to look back at Figure 11.1 to place those open source projects in the context of the SDN ecosystem.

The manner in which open source software may be commercially exploited is dependent on the nature of the open source license under which it is provided, and these licenses are by no means created

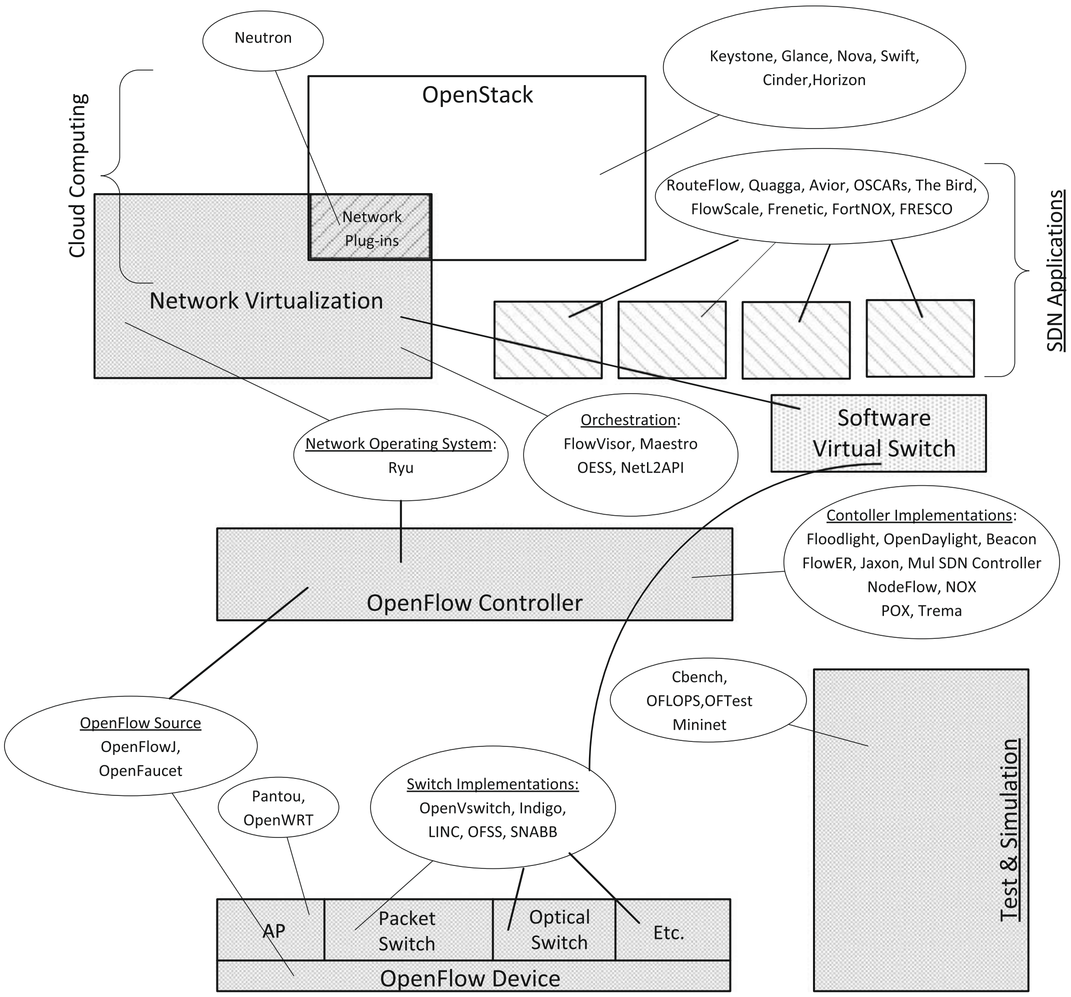

Neutron
Keystone, Glance, Nova, Swift, Cinder,Horizon
OpenStack
Cloud Computing
Network Plug-ins
Network Virtualization
RouteFlow, Quagga, Avior, OSCARs, The Bird, FlowScale, Frenetic, FortNOX, FRESCO
SDN Applications
Software Virtual Switch
Network Operating System: Ryu
Orchestration: FlowVisor, Maestro OESS, NetL2API
Contoller Implementations: Floodlight, OpenDaylight, Beacon FlowER, Jaxon, Mul SDN Controller NodeFlow, NOX POX, Trema
OpenFlow Controller
Cbench, OFLOPS,OFTest Mininet
Test & Simulation
OpenFlow Source OpenFlowJ, OpenFaucet
Pantou, OpenWRT
Switch Implementations: OpenVswitch, Indigo, LINC, OFSS, SNABB
AP
Packet Switch
Optical Switch
Etc.
OpenFlow Device

FIGURE 11.1

SDN open source landscape.

equal. So, before discussing the software modules themselves, let's discuss the prevalent types of open source licenses in use today.

11.1 Chapter-Specific Terminology

In this chapter we introduce a term commonly used by the open source development community: *forking*. If we view a source code base as the trunk of a tree, there are development branches in different locations around the world where innovation and experimentation occur. In many cases, successful and important innovations are contributed back to the evolving master copy of the code and appear in a future release higher up on the trunk of the source code tree, to continue with the analogy. When a new code base is forked from an existing release, it cuts the connection back to the trunk. This means that the continuing innovation on that fork will not be contributed back to the original code base.

11.2 Open Source Licensing Issues

Unless a person has been intimately involved in building a commercial product that incorporates open source software, it is unlikely that he or she understands the complexity of this field of open source licensing. A large number of different open source licenses is extant today. At the time of this writing, *gnu.org* [4] lists a total of 88 free software licenses.

There are many nuances between these free software licenses that may seem arcane to a nonlaw professional. Slight differences about permissions or requirements about using the name of the originator of the license or requirements about complying with U.S. export laws may be some of the only differences between them. Enterprises that offered free source code in the past did so with their organization's name in the license, where the name itself may be the only substantive difference between one company's license and another already existing license. We provide a brief description of the different licenses under which the source mentioned in this chapter has been released, paying particular attention to the business ramifications of choosing to incorporate the software under each of these licenses in a commercial product.

An accepted common denominator of open source is that it is source code provided without charge and that you may use and modify for a defined set of purposes. There is open source available that only allows non-commercial use. We believe that this greatly limits the impact that such software will have, so we do not discuss this further here. Even when commercial use is allowed, however, there are significant differences in the obligations implicit in the use of such software.

Some major categories of open source licensing in widespread commercial use today are the GNU *General Public License* (GPL), *BSD style*, and *Apache*. There are different organizations that opine on whether a given license is truly open source. These groups include the *Free Software Foundation* (FSF) and the *Open Source Initiative*. They do not always agree.

GPL is an extremely popular licensing form. The Linux operating system is distributed under the GPL license. It allows users to copy and modify the software for virtually any purpose. For many companies, though, GPL is too restrictive in one major facet. It incorporates the notion of *copyleft*, whereby if any derivative works are distributed, they must be distributed under the same license. From a practical standpoint, this means that if a commercial company extends a GPL open source code base to provide some new function that it wants to offer to the market, the modifications that are made must be made freely available to the open source community under GPL terms. If the feature in question here embodies core intellectual property that is central to that company's value proposition, this is rarely acceptable. If the function provided by the GPL open source can be used with little added intellectual property, GPL code may be suitable in commercial products. Ensuring that this distinction is kept clear can have massive consequences on the valuation that a startup receives from investors. For this reason, GPL open source is used with caution in industry. Nonetheless, GPL remains the most widely used free software license. There is a GPL V.2 and GPL V.3 that are somewhat more restrictive than the initial version.

The BSD-family licensing model is more permissive. This style of license was first used for the *Berkeley Software Distribution* (BSD), a Unix-like operating system. There have been a number of variants of this original license since its first use; hence we use here the term BSD family of licenses. Unlike GPL code, the user is under no obligation to contribute derivative works back to the open source community. This makes it much less risky to incorporate BSD-licensed open source into commercial products. In some versions of this style of license there is an *advertising clause*. This clause was originally required to acknowledge the use of U.C. Berkeley code in any advertisement of the product. Although this requirement has been eliminated in the latest versions of the BSD license, there are still many free software products licensed with a similar restriction. An example is the widely used OpenSSL encryption library, which requires acknowledgment of the author in product advertising.

The Apache license is another permissive license form for open source. The license was developed by the *Apache Software Foundation* (ASF) and is used for software produced by the ASF as well as some non-ASF entities. Software provided under the Apache license is unrestricted in use except for the requirement that the copyright notice and disclaimer be maintained in the source code itself. Because this restriction has minimal impact on potential commercialization of such software, many commercial entities build products freely incorporating Apache-licensed source code today.

The *Eclipse Public License* (EPL) [7] can be viewed as a compromise between the strong *copyleft* aspects of the GPL and the commercially friendly Apache license. Under certain circumstances, using EPL-licensed source code can result in an obligation to contribute one's own modifications back to the open source community. This is not a universal requirement, though, so developers can often incorporate this software into commercial products without risk of being forced to expose their own intellectual property to the entire open source community. The requirement comes into effect when the modifications are not a separate software module but contain a significant part of the EPL-licensed original and are thus a derivative work. Even when the requirement to release extensions to the original is not in effect, the user is required to grant a license to anyone receiving their modification to any patent the user may hold over those modifications.

The Stanford University *Free and Open Source Software* (FOSS) license [8] is noteworthy in the manner in which it attempts to address the common problem with commercial use of GPL-licensed source code due to its copyleft provision. Stanford's motivation in developing this concept was to avoid the situation where developers would hesitate to develop applications for the Beacon controller (see Section 11.6). Because of the way in which applications interface with the Beacon controller, if it were licensed under GPL without the FOSS exception, such developers would be obliged to contribute their work as GPL open source. This license is specific to the Beacon controller, though the concept can be applied more generally. In essence, Stanford wants to release the Beacon controller code under the terms of GPL. FOSS specifies that a user of the code must continue to treat the Beacon code itself under the terms of GPL, but the code they write as a Beacon application can be released under any of a number of approved FOSS licenses.

These approved licenses are the most common of the open source licenses discussed in this chapter. Notably, an open source application can be released under a license that is more commercially friendly, such as Apache, which does not require that the derivative works on the application code be released to the open source community, thus allowing the developer to guard his newly generated intellectual property.

11.3 Profiles of SDN Open Source Users

When judging the relevance of open source to the SDN needs of one's own organization, the answer will be highly dependent on the role to which the organization aspires on the SDN playing field. To be sure, the early SDN adopters were almost exclusively researchers, many of whom worked in an academic environment. For such researchers, keeping flexibility and the ability to continually tinker with the code is always paramount compared to concerns about support or protection of intellectual property. For this reason, open source OpenFlow controllers and switch code were and continue to be very important in research environments. Similarly, a startup hoping to build a new OpenFlow switch or controller will find it very tempting to leverage the work present in an existing open source implementation, provided that does not inhibit the development and commercial exploitation of the company's *own* intellectual property. If one is a cloud computing operator, tinkering with the OpenFlow controller is far less important than having a reliable support organization that one can call in the event of network failure.

Thus, we propose three broad categories of likely users of SDN-related open source software:

- Research
- Commercial developers
- Network operators

Assuming that a given piece of open source software functionally meets a user's requirements, each of these user classes will have a different overriding concern. A developer in a commercial enterprise will focus on ensuring that he/she does not incorporate any open source that will limit his/her company's ability to exploit its products commercially and ensuring that it will protect its intellectual property. The GPL is often problematic when the open source will be modified, and the modifications must thus be made publicly available under GPL. Generally, researchers have the greatest flexibility and simply desire that the source code be available free of charge and can be modified and distributed for any purpose. Often, though, researchers are also strong proponents of the growth of open source and, thus, may prefer a copyleft license that obliges the users to contribute their modifications back to the open source community. Finally, the network operator will be most concerned about maintaining the network in an operational state and therefore will focus on how the open source product will be supported and how robust it is.

Because of the fundamental noncommercial nature of open source, it will always be a challenge to identify whom to hold responsible when a bug surfaces. This will give rise to companies that will effectively commercialize the open source, where the value they add is not development of new features but to provide well-tested releases and configurations as well as a call desk for support. Some operators will simply fall back on the old habit of purchasing technology from the mainstream NEMs. Whether or not their NEM provider uses open source is almost irrelevant to the operators, since they really are purchasing equipment plus an insurance policy and it matters little where the NEM obtained this technology. On the other hand, whereas a cloud operator may not want to assume responsibility for support of its switching infrastructure, the open source applications, test, or monitoring software might serve as useful starting points for locally supported projects.

In terms of the software projects discussed in this chapter, some examples of how different user types might use some of the software include:

- A hardware switch company that wants to convert a legacy switch into an OpenFlow-enabled switch may use the OpenFlow switch code presented in Section 11.5 as a starting point.

- A company that wants to offer a commercial OpenFlow controller might start its project with one of the OpenFlow controllers we list in Section 11.6, add features to it, and harden the code.
- A cloud operator might use an open source SDN application such as those we describe in Section 11.7 to begin to experiment with OpenFlow technology prior to deciding to invest in this technology.

As we review several different categories of software projects in the following sections, for each we list which of these three classes of users are likely to adopt a given software offering. We now provide an inventory of some of the SDN-specific open source that is used by researchers, developers, and operators. Since the number of these open source projects is quite large, we group them into the basic categories of switches, controllers, orchestration, applications, and simulation and test, as depicted in Figure 11.1. We summarize the different open source contributions in each of these categories in a number of tables in the following sections. Only in an instance where the particular project has played an unusually important role in the evolution of SDN do we describe it in further detail.

11.4 OpenFlow Source Code

As Figure 11.2 shows, both the switch and controller implementations can leverage an implementation of the OpenFlow protocol. There are multiple open source implementations of OpenFlow. We use a format of two paired tables to provide a succinct description of these implementations. In Table 11.1 we provide a basic description of available implementations; in Table 11.2 we list detailed information regarding the provider of the code, the type of license, the source code in which it is written (where relevant), and the likely class(es) of users. We have assigned these target classes of users using a number of criteria, including: (1) the nature of the license, (2) the maturity of the code, and (3) actual users,

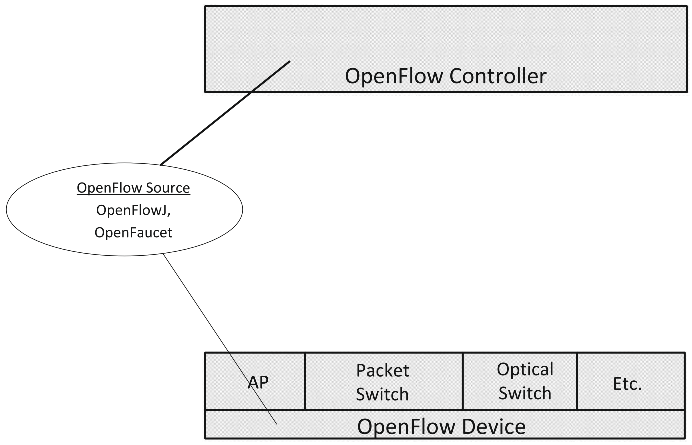


FIGURE 11.2

Applications of OpenFlow source.

Table 11.1 Open Source Implementations of OpenFlow: Description

Product Name	Description
OpenFlow Reference	Reference code base that tracks the specification.
OpenFaucet	Source code implementation of OpenFlow protocol 1.0, used in both controllers and switches.
OpenFlowJ	Source code implementation of OpenFlow protocol. Both Beacon and FlowVisor incorporate this code.

Table 11.2 Open Source Implementations of OpenFlow: Details

Product Name	Source	License	Language	Target User
OpenFlow Reference	Stanford	BSD-style	C	Research
OpenFaucet	Midokura	Apache	Python	Research
OpenFlowJ	Stanford	Apache	Java	Research

when known. OpenFlowJ is notable in that the Beacon controller, itself the fount of a number of other controller implementations, uses the OpenFlowJ code.

We continue to use this format of pairing two tables to provide the description of notable open source projects in each of the major categories we discuss in this chapter. In some cases the license or language may not be known. We indicate this by placing a hyphen in the corresponding table cell. We also mark the table cell with a hyphen for languages in the event that the project uses multiple languages.

11.5 Switch Implementations

In this section we discuss some of the more important available open source SDN switch implementations. The open source SDN switch implementations are summarized in Tables 11.3 and 11.4. Looking back at Figure 11.1 we can see that the open source switch implementations can potentially apply both to OpenFlow hardware device implementations as well as software-based virtual switches. In the following paragraphs we examine the two most widely discussed switch implementations, OVS and Indigo, as well as the only OpenFlow wireless access point code base listed, Pantou.

As of this writing, OVS is the most mature and feature-complete open source implementation of the switch side of an OpenFlow implementation. OVS has been used as the control logic inside genuine hardware switches as well as a virtual switch running within a hypervisor to implement network virtualization. It is in the latter context that the developer of OVS, Nicira, has been for some time shipping OVS with its NVP product. Since the acquisition of Nicira by VMware, the NVP product is now marketed under the name NSX. OVS uses OpenFlow for control of flows and OVSDB for configuration. OVS uses the sFlow protocol for packet sampling and monitoring. The code has been ported to multiple switching chipsets and has thus found a home in several hardware switches as well. Earlier, in Section 6.6.3, we presented an example of OVS being used for SDN via hypervisor-based overlays.

Table 11.3 Open Source OpenFlow Switches: Description

Project	Description
Open vSwitch	Production-quality multilayer virtual switch designed for programmatic network automation using standard management interfaces as well as OpenFlow. Also called OVS.
Indigo	OpenFlow switch implementation designed to run on physical switches and use their ASICs to run OpenFlow at line rates. Note that Switch Light is Big Switch's commercial switch software product and is based on the second generation of the Indigo platform.
LINC	OpenFlow 1.2 Softswitch.
OFSS	OpenFlow 1.1 Softswitch.
of13softswitch	User-space OpenFlow 1.3 Softswitch. Based on Ericsson TrafficLabs 1.0 softswitch code.
Pantou	Wireless router implementation based on OpenWRT with OpenFlow capabilities (see Section 6.4 for more information about OpenWRT.)

Table 11.4 Open Source OpenFlow Switches: Details

Product Name	Source	License	Language	Target User
Open vSwitch	Nicira	Apache 2.0	C	Researchers Developers
Indigo	Big Switch	EPL	—	Developers
LINC	Infoblox	Apache 2	Erlang	Developers
OFSS	Ericsson	BSD-like	C	Research
of13softswitch	CPqD	BSD-like	C	Research
Pantou	Stanford	—	—	Research

Big Switch Networks offers the *Indigo* switch code base under the Eclipse public license. This was a natural move for Big Switch in its desire to create an ecosystem of hardware switches that work well with its commercial OpenFlow controller product, Big Network Controller. Like OVS, Indigo is targeted for use both on physical switches as well as an OpenFlow hypervisor switch for network virtualization environments. In particular, this code can be used to convert a legacy layer two or three switch into an OpenFlow switch. By making this code base available to switch manufacturers, especially smaller manufacturers that may lack the means to develop their own OpenFlow switch-side control code, Big Switch hopes to expand the market for its controller.[1] Because Indigo is integrated with Ethernet switch ASICs, it is possible to switch flows under the OpenFlow paradigm at line rates. Another distinction between OVS and Indigo is that Indigo is implemented specifically to support OpenFlow, whereas OVS can support other control mechanisms such as OVSDB. In addition, OVS has broader support for affiliated network virtualization features than Indigo and includes richer contributions from the open source community. Nevertheless, after OVS, Indigo remains a more likely candidate for an open source starting point for a commercial OpenFlow physical switch than the alternatives discussed in this section. From a feature-set point of view, OVS seems to be a superset of Indigo's features.

[1]Big Switch is now marketing a commercial version of Indigo called Switch Light for this express purpose.

Pantou is actually designed to turn commodity APs into an OpenFlow-enabled AP. This is accomplished by integrating the NOX OpenFlow implementation with OpenWRT. We explained in Section 6.4 that OpenWRT is an open source project that applies the SDN principle of *opening up the device* to a wide selection of low-cost consumer AP hardware. This union of OpenFlow with basic wireless AP functionality available on low-cost hardware is a boon to researchers who want to experiment with the largely virgin field of SDN applied to 802.11 networks.

11.6 Controller Implementations

In this section we discuss some of the more important available open source SDN controller implementations. The open source SDN controller implementations are summarized in Tables 11.5 and 11.6.

The Beacon [8] controller was based on OpenFlowJ, an early open source implementation of OpenFlow in Java. The OpenFlowJ project is hosted at Stanford University. Beacon is a highly influential controller, both for the large amount of early OpenFlow research and development that was done on that controller as well as being the code base from which the Floodlight controller source code was forked.

Beacon is a cross-platform, modular OpenFlow controller [8]. At the time of this writing, Beacon has been successfully deployed in an experimental data center consisting of 20 physical switches and 100 virtual switches. It runs on many different platforms, including high-end Linux servers.

Table 11.5 Open Source Controllers: Description

Project	Description
NOX	The first OpenFlow controller, used as the basis for many subsequent implementations.
POX	Python version of NOX.
Beacon	Java-based OpenFlow controller that is multi-threaded, fast, and modular; uses OpenFlowJ.
Floodlight	Java-based OpenFlow controller derived from Beacon, supported by large developer community.
Ryu	Network controller with APIs for creating applications; integrates with OpenStack. Manages network devices not only through OpenFlow but alternatives such as NETCONF and OF-Config. More than just a controller, it is sometimes referred to as a network operating system.
OpenDaylight	This controller attempts to be the all-purpose SDN controller. It supports numerous southbound interfaces, including OpenFlow, and provides a generic network abstraction for applications in its northbound API.
FlowER	OpenFlow Controller (still in development).
Jaxon	Thin Java Interface to NOX controller.
Mul SDN Controller	OpenFlow Controller.
NodeFlow	OpenFlow Controller.
Trema	OpenFlow Controller, integrated test and debug. Extensible to distributed controllers.

Table 11.6 Open Source Controllers: Details

Product Name	Source	License	Language	Target User
NOX	ICSI	GPL	C++	Research, operators
POX	ICSI	GPL	Python	Research
Beacon	Stanford University	GPLv2 Stanford FOSS Exception v1.0	Java	Research
Floodlight	Big Switch Networks	Apache	Java	Research, developers, operators
Ryu	NTT Communications	Apache 2.0	Python	Research, developers, operators
OpenDaylight	OpenDaylight	EPL	—	Developers
FlowER	Travelping GmbH	MIT License	Erlang	Operators
Jaxon	University of Tsukuba	GPLv3	Java	Research
Mul SDN	Kulcloud	GPLv2	C	Operators
NodeFlow	Gary Berger	—	Javascript	Research
Trema	NEC	GPLv2	Ruby,C	Research

As mentioned earlier, though the core controller code is protected with GPL-like terms, the FOSS license exception allows developers to extend the controller with applications that may be licensed under more commercially favorable terms. Beacon's stability [6] distinguishes it from other controllers used primarily for research purposes. Modules of Beacon can be started, stopped, and even installed while the other components of Beacon continue operating. Evidence of the impact that this code base has had on the OpenFlow controller industry is found in the names of the two most influential open source controllers being used by industry today, *Floodlight* and *Open Daylight*, both echoing the illumination connotation of the name Beacon. In Sections 10.5–10.7, we discussed the use of Beacon, Floodlight, and OpenDaylight when developing SDN applications.

Floodlight was forked from Beacon prior to Beacon being offered under the current GPL/FOSS licensing model [9]. Because Floodlight itself is licensed under the Apache 2.0 license, it is not subject to the copyleft provisions of the Beacon license, and the source is thus more likely to be used by recipients of the code who need to protect the intellectual property of their derivative works on the code base. Floodlight is also integrated with OpenStack (see Section 11.10). A number of commercial OpenFlow controllers are being developed using Floodlight as the starting point. This includes Big Switch's own commercial controller, Big Network Controller. Big Switch's business strategy [5] surrounding the open source controller is complex. The company offers a family of applications that work with the Floodlight controller. Though its commercial controller will not be open source, the company promises to maintain compatibility at the interface level between the commercial and open source versions. This reflects a fairly classical approach to building a software business, which is to donate an open source version of functionality in order to bootstrap a community of users of the technology, hoping that it gains traction, and then offer a commercial version that can be *upsold* to well-heeled users that require commercial support and extended features. Big Switch joined the *Open Daylight* project, donating its Floodlight controller to the effort in hopes that this action would further propagate Floodlight. These hopes were ultimately not realized, as we explain below. Indeed, as of this writing, Big Switch has tweaked this original business strategy. We discuss the company's *big pivot* in Section 12.9.1.

The OpenDaylight project [10] was formed in early 2013 to provide an open source SDN framework. The platinum-level sponsors included Cisco, Brocade, Ericsson, Citrix, Microsoft, and Big Switch Networks. Unlike the ONF, OpenDaylight only considers OpenFlow to be one of many alternatives to provide software control of networking gear. Part of the OpenDaylight offering includes an open source controller, which is why we include OpenDaylight in this section. As we mentioned, Big Switch initially joined and donated its Floodlight controller to OpenDaylight, anticipating that Floodlight would become the key OpenFlow component of the project. This expectation was quickly shattered as OpenDaylight decided to make the Cisco *Open Networking Environment* (ONE) controller the centerpiece and to merely add Floodlight technology to that core. (Cisco now markets a commercial version of the ONE controller as the *Extensible Network Controller*, or XNC.) Subsequent to this, Big Switch reduced its involvement in the OpenDaylight project and ultimately withdrew completely [1]. The XNC controller [13] may use a variety of methods to control network devices. Whereas OpenFlow is one such method, it supports others, including RESTful APIs and Cisco's *Open Network Environment Platform Kit* (onePK). It is thus noteworthy that both SDN via APIs and Open SDN are present in the same XNC controller. The facts that Big Switch stepped back from its commitment to OpenDaylight, there is a total lack of coordination between the ONF and OpenDaylight, and the platinum sponsors are almost all incumbent network equipment manufacturers have all led to a growing skepticism [11] about the compatibility of OpenDaylight with the ONF. When one considers how alternative definitions of SDN have deviated from the initial Open SDN precepts and that OpenFlow remains tightly coupled to those founding precepts, it is not surprising that such schisms arise.

So, despite the recurring theme of illumination used in the names of the most prominent open source controller implementations, the future of the Beacon, Floodlight, and OpenDaylight controllers remains murky.

11.7 SDN Applications

Chapter 10 was dedicated to an analysis of how Open SDN applications are designed and how they interact with controllers and provided a number of sample designs of different SDN applications. If the reader wants to actually embark on the journey of writing his own SDN application, it is clearly advantageous to have a starting point that can be used as a model for new work. To that end, in this section we present some of the more important available open source SDN applications. The open source SDN applications are summarized in Tables 11.7 and 11.8. The term *applications* by its very nature is general, so it is not possible to list all the possible categories of SDN applications. Nevertheless, four major themes surface that have attracted the most early attention to SDN applications. These are security, routing, network management, and *network functions virtualization* (NFV).

In Table 11.7 there are three examples related to routing: the *BIRD*; *Quagga*, which is a general-purpose open source routing implementation that is suitable for use in SDN environments; and *Routeflow*,

Table 11.7 Open Source SDN Applications: Description

Product Name	Description
Routeflow	Integrates IP routing with OpenFlow controller; based on the earlier Quagflow project (see Section 11.11).
Quagga	Provides IP routing protocols (e.g., BGP, OSPF).
Avior	Management Application for Floodlight.
OSCARS	On-Demand Secure Circuits and Advance Reservation system used for optical channel assignment by SDN; predates SDN.
The BIRD	Provides IP routing protocols (e.g., BGP, OSPF).
FlowScale	Traffic load balancer as a service using OpenFlow.
Frenetic	Provides language to program OpenFlow controller abstracting low-level details related to monitoring, specifying, and updating packet-forwarding policies.
FortNOX	Security Framework, originally coupled with NOX controller, now integrated in SE-Floodlight; extends OpenFlow controller into a security mediation service. Can reconcile new rules against established policies.
FRESCO	Security application integrated with FortNOX; provides security-specific scripting language to rapidly prototype security detection and mitigation modules.

Table 11.8 Open Source SDN Applications: Details

Product Name	Source	License	Language	Target Users
Routeflow	CPqD (Brazil)	—	—	Research, developers
Quagga	Quagga Routing Project	GPL	C	Research, developers
Avior	Marist College	MIT License	Java	Research
OSCARS	Energy Services Network (U.S. Department of Energy)	New BSD	Java	Research
The BIRD	CERN	GPL	—	Research
FlowScale	InCNTRE	Apache 2.0	Java	Research
Frenetic	Princeton University	GPL	Python	Research
FortNOX	SRI International	—	—	Research
FRESCO	SRI International	—	—	Research

which is SDN-specific. In Section 11.11 we describe a specific example of open source being used to implement a complete SDN network where Routeflow and Quagga are used together. *Avior* is a network management application for the Floodlight controller. OpenFlow has great potential to provide the next generation of network security [15]. The two relevant examples in Tables 11.7 and 11.8 are *FortNOX* and *Fresco*. As we described in Section 8.6.1, NFV applications constitute the class of applications that virtualize a network services function that is performed by a standalone appliance in legacy networks.

Examples of such devices are traffic load balancers and intrusion detection systems. FlowScale is an NFV application that implements a traffic load balancer as an OpenFlow controller application.

Some of the open source applications covered in this section are OpenFlow applications, where domain-specific problems are addressed through applications communicating with the OpenFlow controller via its northbound interface. Not all SDN applications are OpenFlow applications, however.

11.8 Orchestration and Network Virtualization

We introduced the concept of orchestration in Section 3.2.3. There are many compute, storage, and networking resources available in the data center. When virtualization is applied to these resources, the number of assignable resources explodes to enormous scale. Assigning these resources to the tenants of the data center in an efficient and manageable fashion is a complex technology unto itself. This technology is called *orchestration*. In this section we list some of the more important open source implementations that provide orchestration functionality. Network orchestration is directly related to network virtualization. The term *orchestration* came into use as it became clear that virtualizing the networking portion of a complex data center, already running its compute and storage components virtually, involved the precise coordination of many independent parts. Unlike legacy networking, where the independent parts function in a truly distributed and independent fashion, network virtualization required that a centralized entity coordinate their activities at a very fine granularity—much like a conductor coordinates the exact timing of the thundering percussion interlude with the bold entry of a brass section in a symphony orchestra. The open source SDN orchestration and network virtualization solutions are summarized in Tables 11.9 and 11.10.

11.9 Simulation, Testing, and Tools

In this section we discuss some of the more important open source implementations related to network simulation, test, and tools pertinent to SDN. The open source SDN solutions available in this area are summarized in Tables 11.11 and 11.12. Some of these projects offer software that may have relevance in nonresearch environments, particularly *Cbench*, *OFLOPS*, and *OFTEST*. Although we have not seen

Table 11.9 Open Source Orchestration Solutions: Description

Product Name	Description
FlowVisor	Creates slices of network resources; delegates control of each slice, that is, allows multiple OpenFlow controllers to share a set of physical switches.
Maestro	Provides interfaces for network control applications to access and modify a network.
OESS	Provides user-controlled VLAN provisioning using OpenFlow switches.
NetL2API	Provides generic API to control layer two switches via vendors' CLIs, not OpenFlow; useable for non-OpenFlow network virtualization.
Neutron	OpenStack's operating system's networking component that supports multiple network plugins, including OpenFlow.

Table 11.10 Open Source Orchestration Solutions: Details

Product Name	Source	License	Language	Likely Users
FlowVisor	ON.LAB	—	Java	Research
Maestro	Rice University	Lesser GPL	Java	Research
OESS	Internet2 Indiana University	Apache 2.0	—	Research
NetL2API	Locaweb	Apache	Python	Research, developers, operators
Neutron	OpenStack Foundation	Apache	—	Operators

Table 11.11 Open Source Test and Simulation: Description

Product Name	Description
Cbench	OpenFlow Controller benchmarker. Emulates a variable number of switches, sends PACKET_IN messages to controller under test from the switches, and observes responses from controller.
OFLOPS	OpenFlow switch benchmarker. A standalone controller that sends and receives messages to/from an OpenFlow switch to characterize its performance and observes responses from the switch.
MiniNet	Simulates large networks of switches and hosts. Not SDN-specific, but widely used by SDN researchers who simulate OpenFlow switches and produce traffic for OpenFlow controllers.
OFTest	Tests switch compliance with OpenFlow protocol versions up to 1.2.

Table 11.12 Open Source Test and Simulation: Details

Product Name	Source	License	Language	Likely Users
Cbench	Stanford University	GPL V.2	C	Research, developers, operators
OFLOPS	Stanford University	Stanford License	C	Research, developers, operators
MiniNet	Stanford University	Stanford License	Python	Research
OFTest	Big Switch Networks	Stanford License	Python	Research, developers, operators

Mininet used in a commercial setting, this tool has seen wide use by SDN researchers to emulate the large networks of switches and hosts and the traffic that can cause a controller, such as an OpenFlow controller, to create, modify, and tear down large numbers of flows.

11.10 OpenStack

We introduced the OpenStack Foundation in Section 9.9.3. The OpenStack open source is a broad open source platform for cloud computing, released under the Apache license. We show the role of OpenStack as well as its components in Figure 11.3. OpenStack provides virtualization of the three main components of the data center: compute, storage, and networking. The compute function is called *Nova*. Nova works with the major available hypervisors to manage pools of virtual machines. Examples of the hypervisors that can be used with OpenStack include KVM, XenServer, and VMware, among others. The storage functions are *Swift* and *Cinder*. Swift provides redundant storage such that storage servers can be replicated or restored at will, with minimum dependency on the commodity storage drives that provide the actual physical storage. Cinder provides OpenStack compute instances with access to file and block storage devices. This access can be used with most of the prevalent storage platforms found in

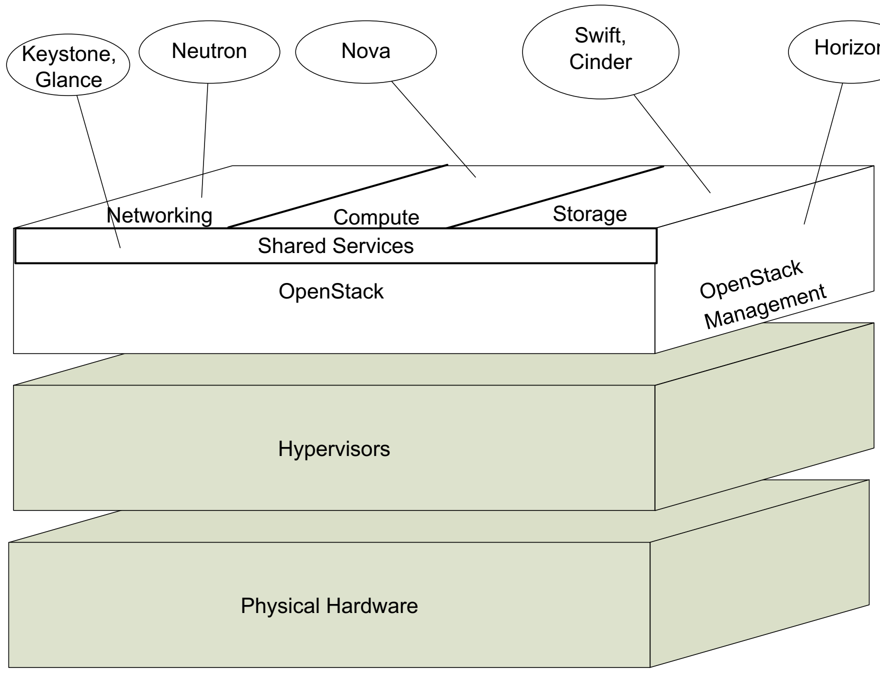


FIGURE 11.3

Openstack components and roles.

cloud computing today. *Horizon* provides a dashboard to access, provision, and manage the cloud-based resources in an OpenStack environment. There are two shared services, *Keystone* and *Glance*. Keystone provides a user authentication service and can integrate with existing identity services such as LDAP. Glance provides the ability to copy and save a server image such that it can be used to replicate compute or storage servers as services are expanded. It also provides a fundamental backup capability for these images. The network virtualization component of OpenStack is provided by *Neutron* and, thus, is the component most relevant to our discussion of SDN. Note that Neutron was formerly known as Quantum.

Architecturally, Neutron's role in OpenStack is embodied as a number of plugins that provide the interface between the network and the balance of the OpenStack cloud computing components. Though OpenStack is not limited to using Open SDN as its network interface, Open SDN is included as one of the networking options. Looking back at Figure 11.1 we see that the Neutron plugin can interface to the northbound API of an OpenFlow controller. Neutron can thus provide the network abstraction layer for an OpenFlow-enabled network. Just as OpenFlow can work with many different network-control applications via a northbound API, so can OpenStack's Neutron have many different kinds of network plugins. Thus, OpenStack and OpenFlow may be married to provide a holistic network solution for cloud computing, but neither is exclusively tied to the other. As shown in Figure 11.4, OpenStack can use Neutron plugins to control legacy network devices, an OpenFlow controller that controls OpenFlow-enabled physical switches, or virtual switches such as OVS [2]. The implementation

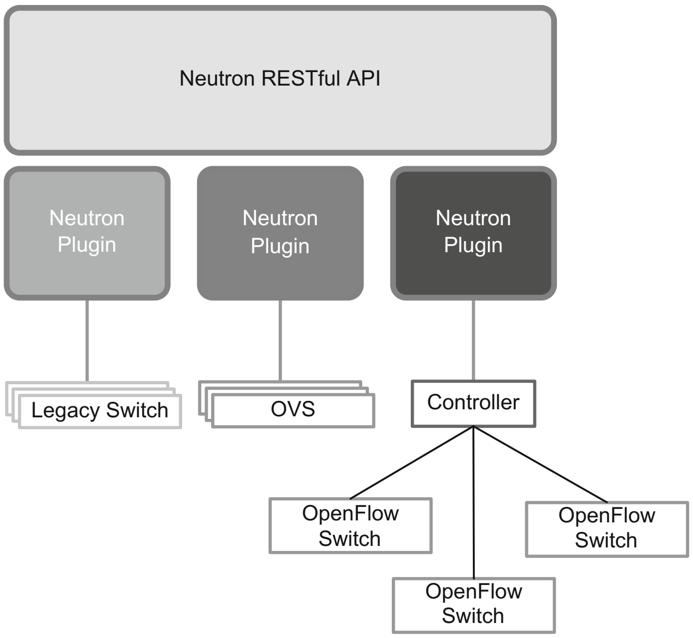


FIGURE 11.4

Openstack plugins.

of the OVS interface, for example, consists of the plugin itself, which supports the standard Neutron northbound APIs, and an agent that resides on the Nova compute nodes in the OpenStack architecture. An OVS instance runs locally on that compute node and is controlled via that agent.

OpenStack exposes an abstraction of a *virtual pool of networks*. This is closely related to the virtual networks abstraction we discussed in relation to the SDN via overlays solution. So, for example, with OpenStack one could create a network and use that for one specific tenant, which maps quite well to the SDN via overlays solutions concept. OpenStack has plugins for many of the existing Overlay solutions.

11.11 Example: Applying SDN Open Source

The *RouteFlow* project [14] provides an excellent use case for open source SDN code. The RouteFlow project has developed a complete SDN test network using only open source software. The goals of the RouteFlow project are to demonstrate:

- A potential migration path from traditional layer three networks to OpenFlow-based layer three networks
- A pure open source framework supporting different aspects of network virtualization
- IP Routing-as-a-Service
- Legacy routing equipment interoperating with simplified intra- and interdomain routing implementations

Figure 11.5 depicts the network topology the project uses to demonstrate their proposed migration path from networks of legacy routers to OpenFlow. This is particularly important because it demonstrates a practical method of integrating the complex world of routing protocols with the OpenFlow paradigm.

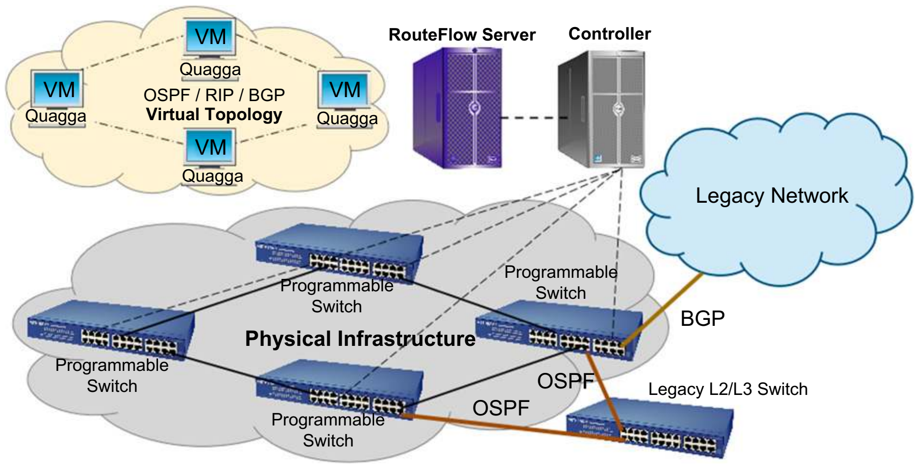


FIGURE 11.5

RouteFlow network topology (Courtesy of CPqD).

Today's Internet is utterly dependent on this family of routing protocols, so, without a clean mechanism to integrate the information they carry into an OpenFlow network, the use of OpenFlow will remain constrained to isolated data center deployments. In Figure 11.5 we see the test network connected to the Internet cloud with BGP routing information injected into the test network via that connection. The test network itself includes one legacy layer two/layer three switch, which communicates routing information to the OpenFlow part of the test network via OSPF. In the test network, we see a cloud of four OpenFlow-enabled switches under the control of an OpenFlow controller. This OpenFlow controller is connected to the RouteFlow server. The RouteFlow server, in turn, collects routing table information from the virtual switches in the adjacent cloud. Figure 11.6 shows these same components in a system architecture view. It is significant that all of the components presented in Figure 11.6 are derived from open source projects, though not all are SDN-specific.

The major components described in the figure are listed below, along with the source of the software used to build the component:

- Network (OpenFlow) controller: NOX, POX, Floodlight, Ryu
- RouteFlow server: Developed in RouteFlow Project
- Virtual topology RouteFlow client developed in RouteFlow Project
- Virtual topology routing engine: Quagga routing engine, XORP
- Virtual topology switches: OVS

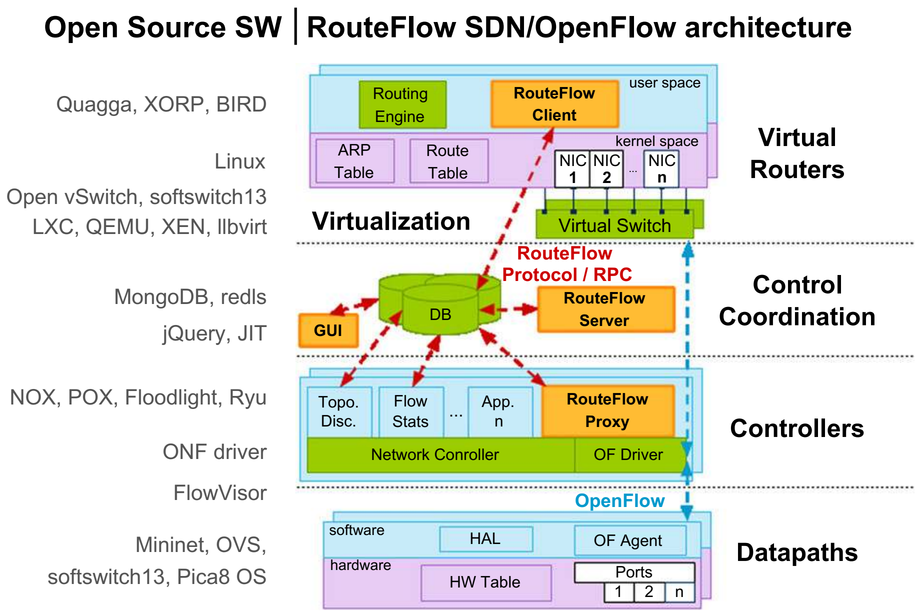


FIGURE 11.6

RouteFlow architecture (Courtesy of CPqD).

- OpenFlow-enabled hardware switches: For example, NetFPGA
- VMs in virtual topology: Linux virtual machines

The virtual routers shown in Figure 11.6 correspond to the VMs in Figure 11.5. Each virtual router comprises an OVS instance that forwards routing updates to the routing engine in the VM. This routing engine processes the OSPF or BGP updates in the legacy fashion and propagates the route and ARP tables in that VM accordingly. The key to the RouteFlow solution is the RouteFlow client that runs on the same VM instance that is in each virtual router. As shown in Figure 11.6, the RouteFlow client gathers the routing information in its local VM routing tables and communicates this to the RouteFlow server via the RouteFlow protocol. The RouteFlow server thus maintains global knowledge of the individual distributed routing tables in each of the virtual routers and can use this information to map flows to the real topology of OpenFlow-enabled switches. This information is communicated from the RouteFlow server to the RouteFlow proxy, which is an OpenFlow controller application running on the same machine as the OpenFlow controller. At this point, the routing information has been translated to OpenFlow tuples, which can be programmed directly into the OpenFlow switches using the OpenFlow protocol. The process results in the OpenFlow-enabled hardware switches forwarding packets just as they would if they followed local routing tables populated by local instances of OSPF and BGP running on each switch, as is the case with legacy layer three switches. RouteFlow is an example of the sort of proactive application that we described in Section 10.2. The flows are programmed proactively as the result of external route information. We contrast this to a hypothetical reactive version of RouteFlow, wherein an unmatched packet would be forwarded to the controller, triggering the determination of the correct path for that packet and the subsequent programming of a flow in the switch.

We have presented this test network showing OpenFlow-enabled hardware switches. RouteFlow additionally supports software switches such as Mininet and OVS for use with network virtualization applications. That is, whereas we described switches depicted in Figure 11.5 as OpenFlow-enabled hardware switches, the RouteFlow project has also demonstrated that these switches may be a soft switch that runs in the ToR alongside a hypervisor.

11.12 Conclusion

SDN's roots are squarely aligned with the open source movement. Indeed, what we call Open SDN in this book remains closely linked with the open source community. In addition to reviewing the better-known SDN-related open software projects, we have provided insight into the many variants of open source licenses in common use. Different open source licenses may be more or less suitable for a given organization. We have defined three broad classes of users that have very different requirements and goals when they use open source software, and we have explained why certain open source licenses might be inappropriate for certain classes of users. Depending on the class of user with which our readers associate themselves, we hope that it is now clear where these readers might find open source applicable for their SDN project. Whatever the commercial future of open source software is within the expanding SDN ecosystem, it is clear that the cauldron of creativity that spawned SDN itself could never have existed without open source software and the sharing of new ideas that accompanies it.

The open source movement may be a powerful one, but as the potential financial ramifications of SDN have become more apparent, other very strong forces have arisen as well. Some of these may be in concert with the open source origins of SDN; others may be in discord. In the next chapter we look at how those other forces are shaping the future of SDN.

References

[1] Duffy J. Big switch's big pivot. Network World; September 11, 2013. Retrieved from <www.networkworld.com/community/blog/big-switchs-big-pivot>.

[2] Miniman S. SDN, OpenFlow and OpenStack quantum. Wikibon; July 17, 2013. Retrieved from <wikibon.org/wiki/v/SDN,_OpenFlow_and_OpenStack_Quantum>.

[3] Comprehensive list of open source SDN projects. SDNCentral. Retrieved from <www.sdncentral.com/comprehensive-list-of-open-source-sdn-projects>.

[4] Various licenses and comments about them. GNU Operating System. Retrieved from <www.gnu.org/licenses/license-list.html>.

[5] Jacobs D. Floodlight primer: an OpenFlow controller. TechTarget. Retrieved from <searchsdn.techtarget.com/tip/Floodlight-primer-An-OpenFlow-controller>.

[6] Muntaner G. Evaluation of OpenFlow controllers; October 15, 2012. Retrieved from <www.valleytalk.org/wp-content/uploads/2013/02/Evaluation_Of_OF_Controllers.pdf>.

[7] Wilson R. The eclipse public license. OSS Watch; November 12, 2012. Retrieved from <www.oss-watch.ac.uk/resources/epl>.

[8] Erikson D. Beacon, OpenFlow @ Stanford; February 2013. Retrieved from <openflow.stanford.edu/display/Beacon/Home>.

[9] Floodlight documentation. Project floodlight. Retrieved from <docs.projectfloodlight.org/display/floodlightcontroller>.

[10] Lawson S. Network heavy hitters to pool SDN efforts in OpenDaylight project. Network World; April 8, 2013. Retrieved from <www.networkworld.com/news/2013/040813-network-heavy-hitters-to-pool-268479.html>.

[11] Duffy J. Skepticism follows Cisco-IBM led OpenDaylight SDN consortium. Network world; April 9, 2013. Retrieved from <www.networkworld.com/news/2013/040913-opendaylight-268534.html>.

[12] Casado M. List of OpenFlow software projects (that I know of). Retrieved from <yuba.stanford.edu/~casado/of-sw.html>.

[13] Stewart P. Defining Cisco's onePK. The Packet University; March 19, 2013. Retrieved from <www.packetu.com/2013/03/19/defining-ciscos-onepk>.

[14] Welcome to the RouteFlow project. Routeflow. Retrieved from <sites.google.com/site/routeflow>.

[15] Now available: our SDN security suite. OpenFlowSec.org. Retrieved from <www.openflowsec.org>.

[16] Sorensen S. Top open source SDN projects to keep your eyes on. O'Reilly Community; August 1, 2012. Retrieved from <broadcast.oreilly.com/2012/08/top-open-source-sdn-projects-t.html>.

CHAPTER 12

Business Ramifications

The fact that the very definition of SDN is vague confounds any attempt to assess the full business impact of SDN. If we narrow our focus to Open SDN, the business impact so far is minimal. The largest financial transactions to date, however, have revolved around technologies that only fall under broader definitions of SDN. This larger SDN umbrella encompasses any attempt to perform networking in a novel way that involves separation of control and data planes, unlike classical layer two and three switching. By casting this wider net, we include companies and technologies that are often totally proprietary and whose solution is tightly focused on a specific network problem, such as network virtualization in the data center. But as the saying goes, this has been where the money has gone, so this is where the greatest business impact has been.

There seems to be a growing consensus in the networking business community that SDN is here to stay and that its impact will be significant. This is supported by the sheer number of venture capital companies that have invested in SDN startups over the past six years, the magnitude of those investments, as well as the size of the acquisitions that have occurred since 2012 in this space. Unusually in the networking field, this particular paradigm shift does not hinge on any breakthroughs in hardware technology. In general, SDN is a software-based solution. This means that technical hurdles are more easily overcome than having to pass some gate-density or bits-per-second threshold. Customers generally view the advent of SDN with excitement and the hope of more capable networking equipment for lower costs. At first, entrenched NEMs may view it with trepidation because it is a disruptive force, then usually find a way to jump on the SDN bandwagon by incorporating some form of the technology into their product offering. We look at the dynamics of these various trends in this chapter.

12.1 Everything as a Service

Figure 12.1 shows that the ratio of *capital expenditures* (CAPEX) to *operational expenditures* (OPEX) on network equipment will *decrease* as we move through the first few years of SDN deployments. This is because the growth of SDN coincides with a fundamental shift in the way that data centers and other large IT consumers will pay for their network equipment and services. New licensing models, subscription-based models, and even usage-based models are all becoming increasingly common. The rapid expansion of new service delivery models such as *Software as a Service* (SaaS) and *Infrastructure as a Service* (IaaS) is testimony to this shift. The old model of enterprises purchasing network boxes from NEMs, amortizing them as a capital expense, and then repeating the cycle with forklift upgrades every few years was a great business model for the NEMs, but it is gradually disappearing. These new models, where virtually every aspect of the networking business is available as a service and treated as an operational expense, will likely continue to displace the traditional network spending patterns.

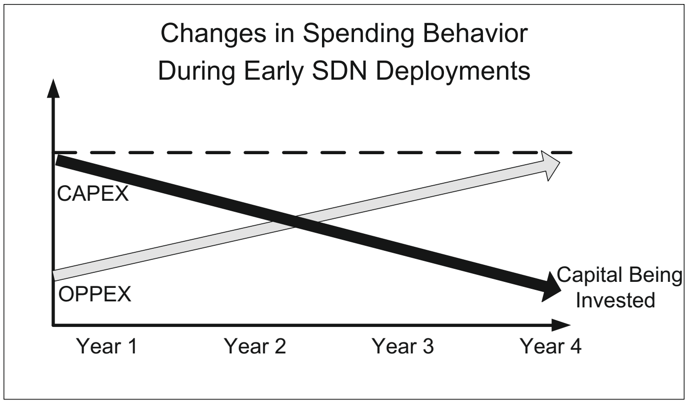


FIGURE 12.1

CAPEX on network equipment migrates to OPEX.

12.2 Market Sizing

The net impact of a move to SDN technology will be influenced more than anything by the *total available market* (TAM) for SDN. At this early stage, forecasts are notoriously unreliable, but some effort has been expended to try to quantify what the total opportunity may be. Based on the studies reflected in [22], though SDN revenues in 2012 were less than $200 million, they are likely to grow to more than $35 billion by 2018. This is forecast to derive 60% from layer two and layer three SDN equipment sales and 40% from services equipment (layers four through seven). This growth will be primarily driven by a growth in network virtualization. It does not represent new growth in networking expeditures due to SDN but rather the displacement of other network spending on SDN technology. Another study [18] predicts the SDN market will grow sixfold between 2014 and 2018, with SDN-enabled switches and appliances comprising the bulk of that revenue.

12.3 Classifying SDN Vendors

Network virtualization is happening, and the term SDN is being blurred with this concept, for better or worse. The kind of company that dominates network virtualization remains an open question, though. Certainly, there are at least three major classes of companies that are well positioned to make this transition. First, the NEMs themselves are well positioned, since by definition they already dominate the network equipment business. Second, since the server virtualization companies such as VMware and Citrix already have a firm grasp on compute and storage virtualization, it is possible for them to make a horizontal move into network virtualization. Finally, since success in this area will likely depend on being successful at a SaaS business model, it would be wrong to discount the possibility of a software giant such as Microsoft dominating here. At this time, though, it appears that the battle for network virtualization will be fought between the titans of the network equipment industry and those of server virtualization. We take a deeper look at the impact of SDN on incumbent NEMs in Section 12.4. Now let's look at the server virtualization companies most likely to be impacted by SDN.

12.3.1 Server Virtualization Incumbents and SDN

Since so much of the focus on SDN has been related to network virtualization in the data center, it is probable that incumbents that have traditionally been involved in compute and server virtualization will be drawn to extend their offerings to include network virtualization. The most salient example, VMware, has already surfaced many times earlier in this book. Already the world leader in server virtualization, VMware extended its reach into network virtualization with the acquisition of Nicira. Other companies that could follow suit include HP's server division, NEC, and Citrix, among others. Not unusually, HP is a wildcard here. HP is already both a major server manufacturer as well as a leading network equipment manufacturer. Of the larger incumbent NEMs, HP seems to be embracing Open SDN more enthusiastically than some of its competitors. Since HP's server division is already a powerful force in compute and storage servers, it would seem that HP was particularly well positioned to bring these two divisions together in a comprehensive networking-compute-storage virtualization story. HP will have to execute on this better than it has on most of its recent major initiatives, so it is impossible to predict what the outcome will be here.

12.3.2 Value-Added Resellers

One class of vendor that must not be neglected in this discussion is the *value-added reseller* (VAR). Such companies, including well-known examples such as Presidio and Compucom as well as numerous smaller companies, have long played a critical role in providing the customer face for the NEM. From a revenue perspective, this was possible because the network equipment sold had sufficient margins that the VAR could run a viable business based on its share of those margins. As illustrated in Figure 12.1, part of the premise of SDN is that the hardware boxes become a smaller percentage of the overall network expenditure. Also, the *box sale* will not involve as much customer touch as it has traditionally, and therefore the traditional value added by the VAR is not available to be added. It thus seems likely that in order to survive in an increasingly SDN world, these VARs will have to evolve into a service model of business. Indeed, their value was always in the service that they provided to their customers, but now this service will not be at the box level but at the network or service level. This problem will be exacerbated as the NEMs such as Cisco, due to the thinner margins, increasingly sell their boxes directly to customers. SingleDigits and SpotOnNetworks are classic examples of companies that will need to shift with this change in business model. With SDN, such VARs will have the opportunity to innovate in specific markets of interest to their customer base, not merely sell the features that are provided to them by the NEMs.

The companies most likely to be heavily impacted by a major transition to SDN are the incumbent NEMs.

12.4 Impact on Incumbent NEMs

Incumbent NEMs are extremely astute marketing organizations. When they intuit that a tectonic shift in network purchasing patterns is about to occur, they generally will find a way to redefine themselves or the problem set so that their current solutions are in lockstep with the paradigm shift. In some cases, this will be just an instance of slide-ware; in many other cases, there will be genuine shifts within the NEMs to align themselves in *some* way with the new paradigm.

This transition will not come without pain. There are already signs within some NEMs of belt-tightening, attributed to market shifts related to SDN. For example, in early 2013, Cisco laid off 500 employees, partly in an effort to realign with the advent of SDN [1]. The growth of SDN, along with network virtualization, will reduce the demand for legacy *purpose-built* routers and switches that have been the mainstay of the large NEMs' revenues for many years.

12.4.1 Protect Market Share

Large, entrenched NEMs, such as Cisco and Juniper, have to walk a fine line between protecting their market share and missing out on a new wave of innovation. For example, the high valuation given Nicira by VMware, shown in Table 12.2 later in this chapter, can clearly put Cisco, Juniper, and their ilk on the defensive. The NEMs are extremely competent in both technical and marketing terms, and they have the financial resources to make sure they have a competitive answer to the network virtualization offering from VMware.

When an incumbent has a large war chest of cash, one defensive answer to a new, competing technology is to simply purchase one of the startups offering it and incorporate that product into its solutions. However, if this is not done carefully, this can actually *erode* market share. Although the startup has the luxury of being able to tout disruptive technology as a boon to consumers, the incumbent must somehow portray that technology as complementary to its existing offerings. Failing to respond in this way, the company runs the risk of reducing sales of the traditional product line while the nascent technology is still gaining a foothold.

12.4.2 Innovate a Little

Although an established NEM can readily incorporate *adjacent* technologies into its product line and even gain market share as a result, doing so with a directly confrontational, paradigm-shifting technology such as SDN is a much more delicate proposition. To avoid getting left behind, the incumbent will need to ostensibly adopt the new technology, especially in name. An incumbent can provide a solution that is competitive to the paradigm shift by adhering to the following model:

- Purport to solve the same problem (e.g., network virtualization)
- Remain based on the incumbent's then-current technology
- Innovate by layering some aspects of the new technology on top of the existing technology
- Employ widespread use of the buzzwords of the new technology in all marketing materials (e.g., SDN washing).

The SDN-via-APIs approach, which we discussed in detail in Section 6.2, epitomizes the layering of the new approach on top of legacy technology. We recall that the underlying switching hardware of the leading NEMs has been optimized by custom hardware designs over many years. These hardware platforms may have inherent performance advantages over low-cost, *white-box* switches. If the incumbent provides a veneer of SDN over its already superior switching hardware, this may present formidable competition to a new SDN paradigm of a separate and sophisticated control plane controlling low-cost commodity switches. Indeed, it is possible that from a pure performance metric such as packets switched per second, the pure SDN model may suffer compared to high-end legacy switches. This veneer of SDN need not be just a stop-gap measure. As this situation evolves over time it can have the very positive effect of providing technologies that help their customers evolve to SDN. This can provide many of the benefits of SDN, albeit on proprietary technology. Most customers simply will not tolerate a forklift upgrade being imposed on them.

Following this model is a virtual necessity for large, established NEMs. This model provides them with margin preservation during a time of possibly great transition. If the new paradigm truly does become dominant, they will be able to ride the coattails of its success while slowing down the transition such that they are able to maintain robust margins without interruption. As we mentioned, they have the technical and marketing know-how to manage favorably this kind of transition. They also have something else, which may be even more important: *the bulk of the enterprise customers*. Such customers do not shift their mission-critical compute, storage, or networking vendors willy-nilly unless there are drastic technical shortcomings in their product offerings. By managing their customers' expectations carefully and innovating just enough, NEMs' advantage of owning the customer may be unbeatable.

12.5 Impact on Enterprise Consumers

Inevitably, some of this book may read like a sort of SDN evangelist's bible. The vast potential benefits of SDN have been inventoried and explained. It is easy to underestimate the risks that enterprises will run by severing their historic umbilical cords with the NEMs. These umbilical cords have been unwieldy and expensive, to be sure, but the security that comes with this safety net has undoubtedly provided countless IT managers with many a good night's sleep. We, the authors, are in fact SDN evangelists and believe that, at least in certain parts of networking infrastructure, SDN's power to innovate and to mitigate costs will indeed be a historically positive force. Nonetheless, it would be naïve to trivialize the birthing pains of a major paradigm shift in such a highly visible area. In this section we present an overview of both the positive business ramifications of this new technology paradigm (e.g., reduced costs, faster innovation) and the downside of diluted responsibility when the network goes down.

The most prevalent solution described in the SDN context is *network virtualization*, which we have discussed at length throughout this book. Although many of the solutions for network virtualization are not based on Open SDN, it is likely that the bulk of the enterprise budgets targeted for SDN technology will be spent on network virtualization. One challenge is that though these same customers understand compute and storage virtualization, for many network virtualization remains a snappy catchphrase. Customers need considerable sophistication to understand how network virtualization will work for them.

12.5.1 Reduced Equipment Costs

In the situation of an existing data center, the hoped-for reduction in equipment costs by migrating to SDN may prove elusive or at least delayed. A solution such as VMware's NSX touts the fact that its separate control plane can achieve network virtualization without switching out the legacy networking equipment. Though this is desirable in the sense that it produces less equipment churn in the data center racks, it does nothing to lower the money being spent on the switching equipment itself. Under this scenario, the customer continues to pay for highly capable and costly layer two and layer three switches but only uses them as dumb switches. It is hard to see a fast path to reduced equipment costs in this scenario. Over the longer term and as the legacy switching equipment becomes fully amortized, the opportunity will arise to replace that legacy underlay network with lower-cost OpenFlow-enabled switches.

In a greenfield environment, the Open SDN model prescribes intelligent software control of low-cost, OpenFlow-enabled switches. This scenario is conceivable, but that division of the control and data plane brings with it a concomitant dilution of responsibilities. In the past, when your data center network went down, the IT department could usually point their finger at their primary NEM, whether Cisco, Juniper, HP, or another. In a perfect, pure SDN world, the white-box switches are just hardware devices that perform flawlessly according to the programming received from their SDN controller. No one with operational experience with a big network would believe that things will actually play out according to that script. The customer will still want *one throat to choke*, as the colloquial saying goes. Getting one company to offer up that single throat to choke may be difficult when the network is instantiated on virtual and physical switches as well as a controller, all running on hardware purchased from three different vendors.

12.5.2 Avoiding Chaos

Who are you going to call … (when the network goes down)? When it boils down to a CIO or IT manager making the decision to migrate any major part of the network from the legacy gear to SDN, the hype about greater ease and granularity of control, higher equipment utilization rate, and the other pro-SDN arguments will run into a brick wall if that decision maker fears that network reliability may be jeopardized by making the transition. For that reason, it is a virtual certainty that any migration to SDN in typical enterprise customers will be a slow and cautious one. NEMs may incorporate fully integrated SDN-based solutions into their product portfolios, much like what has occurred with Linux in the world of servers. This will still provide the proverbial throat to choke while using SDN to meet specific customer needs.

We note that certain immense enterprises such as Google and Yahoo! are outliers here in that they may have as much networking expertise as the large NEMs themselves. This explains why these organizations, along with selected research universities, have been at the forefront of OpenFlow innovation. As we explained in Section 9.9, this is evidenced by the composition of the board of the ONF. There are no NEMs on that board, but Yahoo! and Google are represented. For the more typical enterprise, however, this transition to SDN will be viewed with much more caution. Experiments will be done carefully with non-mission-critical parts of the network.

In general, we should assume that large-scale transition to SDN will take much longer than some pundits have claimed. This will provide more time for incumbent NEMs to adapt in order to avoid a real shake-up in the networking industry.

12.6 Turmoil in the Networking Industry

12.6.1 Fewer and Fewer Large NEMs

One of the advantages of an ecosystem in which a small number of NEMs make huge profits off a market they dominate was that the huge profits did in fact drive continued innovation. Each NEM tried to use the next generation of features to garner market share from its competitors. Generally, this next generation of features addressed real problems that NEMs' customers faced, and innovation proceeded apace. A precept of Open SDN is that since users, via the open source community, will be able to *directly* drive this innovation, it will be both more pertinent and less expensive. However, this thesis has yet to be proven in practice. That is, will the creativity of the open source community be able to match the self-serving creativity of the NEMs themselves? The NEMs' ability to generate huge profits through innovation has resulted in much of the progress in data networking that our society has enjoyed over the past two decades, so the answer to this question is not an obvious one. The NEMs will strive to perform a balancing act between innovating sufficiently to retain their market control and their desire to maintain the status quo.

12.6.2 Migration to Cloud Computing

If we take as a given that the need for network virtualization in large data centers is the prime driver behind SDN, we must acknowledge that the growth of cloud computing is one of the major factors that has forced those data centers to grow ever larger and more complex. From the end user's perspective, cloud computing has grown more relevant due to transitions in a number of related elements:

- Usage-based payment models
- Digital media and service providers
- Digitial media access and sharing providers
- Online marketplaces
- Mobile phone network providers
- Web hosting

More and more enterprises avail themselves of the cloud computing capabilities of Google and Amazon for a variety of applications. Apple's *iCloud* takes local management of your smartphone, tablets, and laptops and handles it in the cloud. Sharing services such as *DropBox* and *GoogleDocs* offer media sharing via the cloud. Google offers a plethora of business applications via the cloud. Amazon Data Services offers web hosting on an unprecedented scale. As more users depend on the cloud for an ever-increasing percentage of their applications needs, both the quantity and reliability of data centers are under constant pressure to increase. This force creates the pressure on the data center providers and the vendors that supply equipment to them to innovate and keep up with this relentless growth. The nature of the networks in these data centers is virgin soil for a new networking paradigm in network virtualization if it can help manage this growth in a controlled fashion.

IaaS provides networking capacity via the OPEX payment model. The growing popularity of this model in itself contributes to the turmoil in the networking industry, where networking capacity has traditionally been obtained by purchasing equipment. Many of the SDN startups discussed in this chapter hope to displace incumbent technology by offering their solutions via some kind of subscription model.

12.6.3 Channel Dynamics

Among the consequences of the migration to cloud computing are radical changes to the way that customers purchase networking capabilities and to the companies that traditionally had the direct customer touch in those sales. Networking operating systems and applications will be bundled with a mix of private and public cloud offerings. A new set of vendors skilled at cloud-based orchestration solutions will customize, deploy, and manage compute, storage, and networking virtualization functions. Based on the enterprise's migration to cloud services, network solution providers will migrate from selling networking boxes to offering networking services. The VARs that now focus on reselling and supporting network hardware design and implementations will likely consolidate into larger and more comprehensive IT services companies. IT services companies will hire, train, and acquire skilled resources that understand how to design, implement, and manage *Web operations functions* (WebOps) on behalf of enterprise customers.

12.7 Venture Capital

It is no coincidence that a significant amount of *venture capital* (VC) investments have been placed in SDN startups in the past four years. It is the nature of venture capital firms to seek out groups of bright and ambitious entrepreneurs who have aspirations in a space that is likely to see rapid growth within a well-bounded time horizon. Depending on the field, the distance to that time horizon can vary greatly. In the field of medical startups, this horizon may be more than 10 years after the initial investments. In data networking, however, the time horizons tend to be shorter, between three and five years. SDN technology is the perfect contemporary fit for the VC firm with a history of investing in networking startups. The market, at least in the data center, is clamoring for solutions that help them break what is perceived as a stranglehold by the large incumbent NEMs. The amount of money available to be spent on upgrades of data center networks in the next few years is enormous. SDN's time horizon and the corresponding TAM is like a perfect storm for VCs. SDN may be the largest transformation in the networking world in the past three decades [8]. This is not merely a U.S.-based phenomenon, either. Some other countries, notably Japan and China, see SDN as a means to thwart the multidecade-long hegemony that U.S. NEMs have had on switching innovation. In these countries, both adoption by customers as well as offerings by local NEMs are likely to heavily influence the uptake of SDN in the coming years. In Table 12.1 we list VC firms that have been the earliest and most prominent investors in SDN startups.

These investors have furnished the nest from which many companies have been hatched. Some of these companies are still in startup mode; others have been acquired. As of this writing, none of the investments listed in Table 12.1 have resulted in *initial public offerings* (IPOs). In the next two sections

Table 12.1 Major VCs Investing in SDN Startups as of 2013

Venture Capital Company	Invested In
Khosla Ventures	Big Switch
Redpoint Ventures	Big Switch
Goldman Sachs	Big Switch
Intel Capital	Big Switch
Benhamou Global Ventures	ConteXtream
Gemini Israel Funds	ConteXtream
Norwest Venture Partners	ConteXtream, Pertino
Sofinnova Ventures	ConteXtream
Comcast Interactive Capital	ConteXtream
Verizon Investments	ConteXtream
Lightspeed Venture Partners	Embrane, Pertino, Plexxi
NEA	Embrane
North Bridge Venture Partners	Embrane, Plexxi
Innovation Network Corporation of Japan	Midokura
NTT Group's DOCOMO Innovations	Midokura
Innovative Ventures Fund Investment (NEC)	Midokura
Jafco Ventures	Pertino
Hummer Winblad	Plumgrid
U.S. Venture Partners	Plumgrid
Vantage Point Capital	Pica8
Battery Ventures	Cumulus
Matrix Partners	Plexxi

we take a look at what has happened with both the major acquistions that have resulted from these investments and those companies still striving to determine their future.

12.8 Major SDN Acquisitions

One of the most visible signs of major transformations in a given market is a sudden spate of large acquisitions of startups, all purporting to offer the same new technology. Certainly 2012 was such a year for SDN startups. The size of the acquisitions and the concentration of so many in such a relatively short calendar period was an extraordinary indicator of the momentum that SDN had gathered by 2012. Table 12.2 provides a listing of the largest SDN-related acquisitions that took place in 2012–2013. It is rare to see so many related acquisitions of such large dollar value in so short a period of time. If SDN did not already serve as the honey to attract investors, it surely did by 2012. Most of the companies listed in Table 12.2 were already either major players in the SDN landscape at the time of their acquisition or are now part of large companies that plan to become leaders in SDN as a result of their acquisition. In this

Table 12.2 Major SDN Acquisitions in 2012–2013

Company	Founded	Acquired By	Year	Price	Investors
Nicira	2007	VMware	2012	$1.26B	Andreessen Horowitz Lightspeed Ventures
Meraki	2006	Cisco	2012	$1.2B	Google Felicis Ventures Sequoia Capital DAG Ventures Northgate Capital
Contrail	2012	Juniper	2012	$176M	Khosla Ventures Juniper
Cariden	2001	Cisco	2012	$141M	Not Available
Vyatta	2006	Brocade	2012	Undisclosed	JPMorgan Arrowpath Venture Partners Citrix Systems HighBAR Partners
Insieme	2012	Cisco	2013	$863M	Cisco

section, we attempt to explain the perceived synergies and market dynamics that led to the acquisition of each.

12.8.1 VMware

Nicira's technology and the success the company has enjoyed have been a recurring theme in this book. VMware's acquisition of Nicira stands out as the largest dollar-value acquisition purporting to be primarily about SDN. VMware was already a dominant force in the business of compute and storage software in large data centers. Prior to the acquisition, VMware was largely leaving the networking part of the data center to the NEM incumbents, most notably Cisco [10], with which VMware had partnered on certain data center technologies such as VXLAN. Since SDN and the growing acceptance of the need for network virtualization were demonstrating that a shift was forthcoming in data center networking, it made perfect sense for VMware to use this moment to make a major foray into the networking part of the data center. Nicira provided the perfect opportunity for that. The explosive growth of cloud computing has created challenges and opportunities for VMware as a leader in the compute and storage virtualization space [2]. VMware needed a strategy to deal with cloud networking alternatives, such as OpenStack and Amazon Web Services, and Nicira provides that. This competitive landscape is fraught with fuzzy boundaries, though. Though VMware and Cisco have worked closely together in the past and Nicira and Cisco collaborated on OpenStack, it is hard not to imagine the battle for control of network virtualization in the data center pitting VMware and Cisco against one another in the future.

12.8.2 Juniper

Juniper was an early investor in Contrail [5]. This deal falls into the *spin-in* category of acquisition. Juniper recognized that it needed to offer solutions to address the unique East-West traffic patterns common in the data center but atypical for its traditional WAN networking customers. By acquiring Contrail, Juniper obtained network virtualization software as well as network-aware applications that help address these data-center-specific problems. Juniper additionally gained the Contrail management team, which consisted of senior managers from Google and Aruba as well as others with outstanding pedigrees that are targeted to help Juniper bring SDN solutions to enterprise customers. The Contrail solution has an orchestration component that supports existing protocols, notably BGP and XMPP. It also uses the OpenStack standard for network virtualization. Thus, support for OpenStack is a common thread among Cisco, VMware, and now Juniper. It is difficult to predict at this time whether these companies will use this standard as a platform for collaboration and interoperability or look elsewhere to distinguish their network virtualization offerings from one another.

12.8.3 Brocade

Brocade needed a virtual layer three switch to connect virtual network domains for cloud service providers [4]. Brocade believes it has found the solution to this missing link in its acquisition of Vyatta. Vyatta delivers a virtualized network infrastructure via its on-demand network operating system. The Vyatta operating system is able to connect separate groupings of storage and compute resources. Vyatta's solution works both with traditional network virtualization techniques such as VLANs as well as newer SDN-based methods [4].

12.8.4 Cisco

Not surprisingly, in total dollar terms, the biggest SDN-acquirer in 2012 was Cisco. As shown in Table 12.2, Cisco acquired both Cariden and Meraki in 2012. Both of these acquisitions are considered in the networking literature [5,7] to be SDN-related acquisitions, though the Meraki case may be stretching the definition of SDN somewhat. Cariden was founded in 2001, well before the term SDN came into use. Cariden was respected as the developer of IP/MPLS planning and traffic engineering software. It is easy to envision how Cariden's pre-SDN MPLS traffic engineering was able to grow into its work on SDN [3], since easier mapping of flows to MPLS LSPs and VLANs is a major step in the transition to SDN. Cisco hopes to use Cariden's software for managing networks to help achieve the industry-wide goal of being able to erect virtual networks with the same ease as instantiating a new virtual machine. As for Meraki, the value of the acquisition indicates the importance that Cisco places on this new member of the Cisco family. What distinguishes Meraki from Cisco's pre-acquisition technologies is that Meraki offers cloud-based control of the wireless APs and wired switches that it controls. This web-centric approach to device management fills a gap in Cisco's portfolio [7], providing it with a midmarket product in a space vulnerable to a competitive SDN-like offering. We believe that Meraki is really an outlier in terms of the classical definition of an SDN company, but the fact that its device control is separated from the data plane on the devices and implemented in the cloud supports the company's inclusion here as a major SDN acquisition.

Insieme was founded by Cisco in 2012 with plans to build *application-centric infrastructure* [25]. Insieme is a Cisco *spin-in* that received over $100 million in seed funding from Cisco. SDN is a game-changing technology that can be disconcerting to the largest incumbent in the field, Cisco. Though Cisco has brought to market a family of SDN-related products revolving around its commercial XNC controller, those actions had not convinced the market that Cisco had truly embraced SDN. The Insieme acquisition changed this situation. The high valuation given to the Insieme acquisition is affirmation of the importance Cisco is now giving to SDN. Understandably, the Insieme product offering attempts to pull potential SDN customers in a purely Cisco direction. Though the new products may be configured to interoperate with other vendors' SDN products, they may also be configured in a Cisco-proprietary mode that purportedly works better. Significantly, the Insieme product line does represent a break with Cisco's legacy products. The new router and switch line from Insieme will not smoothly interoperate with legacy Cisco routers [6].

12.9 SDN Startups

The frenzy of customer interest in SDN, combined with investors' willingness to provide capital for new business ventures, results in the unsurprisingly long yet inevitably incomplete list of SDN startups as of 2013, which we present in Table 12.3. SDN entrepreneurs are arriving so fast on the scene that we cannot claim that this is an exhaustive list. Rather than attempt to cite every SDN startup, we hope to describe a succinct set of different markets that might be created or disrupted by SDN startups. For each of these, we provide one or more examples of a new business venture approaching that market. The general categories we define are:

- OpenFlow stalwarts
- Network virtualization for the data center
- Network virtualization for the WAN
- Network functions virtualization
- Optical switching
- Mobility and SDN at the network edge

In deference to SDN purists, we begin with companies squarely focused on products compliant with the OpenFlow standard. Although this may represent to SDN purists the only true SDN, it has not had the financial impact that non-OpenFlow-focused network virtualization solutions have had. Indeed, network virtualization represents the bulk of investments and acquisition dollars expended thus far in SDN. For this reason, we treat network virtualization in the two separate subcategories listed above. We conclude with some markets peripheral to the current foci in SDN that may have considerable future potential.

12.9.1 OpenFlow Stalwarts

One of the tenets of SDN is that by moving the intelligence out of the switching hardware into a general-purpose computer, the switching hardware will become commoditized and thus low cost. Part of the tension created by the emergence of SDN is that such a situation is not favorable to the incumbent NEMs, accustomed to high margins on their switches and routers. OpenFlow zealots generally respond that lower-cost *original device manufacturers* (ODMs) will fill this space with inexpensive, OpenFlow-compatible white-box switches. This, of course, opens the door for startups that view providing the white-box switches as a "virgin soil" opportunity.

Table 12.3 Startup Landscape in 2013

Company	Founded	Investors	Product Focus	Differentiator
Big Switch	2010	Khosla Ventures Redpoint Ventures Intel Capital Goldman Sachs	OpenFlow-based software	
ConteXtream	2007	Benhamou Global Ventures Gemini Israel Funds Norwest Venture Prt. Sofinnova Ventures Verizon Investments Comcast Interactive Cap.	Network virtualization in the data center	Grid computing
Cumulus	2010	Battery Ventures	White-box switch software	
Embrane	2009	Lightspeed Venture Prt. NEA North Bridge Venture Prt.	Network virtualization in the data center	Network appliances via SDN
Midokura	2009	Innovation Network Corp. of Japan NTT Investment Prt.	Network virtualization for cloud computing	IaaS
Nuage	2012	Alcatel-Lucent	Network virtualization	
Pertino	2011	Norwest Venture Prt. Lightspeed Venture Prt. Jafco Ventures	Network virtualization for the WAN	
Pica8	2009	Vantage Point Capital	White-box switch software	OpenFlow 1.2 OVS
Plexxi	2010	North Bridge Venture Prt. Matrix Prt. Lightspeed Venture Prt.	Ethernet-optical switch orchestration controller	SDN control of optical switches
Plumgrid	2011	Hummer Winblad U.S. Venture Prt.	Network virtualization	No OpenFlow
Pluribus	2010	Menlo Ventures New Enterprise Assoc. Mohr Davidow Ventures China Broadband Cap.	Physical switch	Integrated controller
Tallac	2012	–	Cloud-based SDN WiFi providing multitenant services	Open API to enable NFV services for vertical market applications
Vello	2009	–	Network operating system (controller) optical switches, white-box Ethernet switches	OpenFlow support

As we described in Section 9.5, this is the basis of the strategy of Chinese-based startup Pica8 [9]. Pica8 provides a network operating system and reference architectures to help its customers build SDN networks from low-cost switches, the Pica8 operating system, and other vendors' OpenFlow-compatible products. Through its white-box ODM partners Accton and Quanta, Pica8 offers four Gigabit Ethernet switch platforms based on OVS that currently support OpenFlow 1.2. One wonders, though, if there is a way to turn this into a high-margin business.

Big Switch also now has a business strategy centered on working with white-box switch ODMs. Though Big Switch continues to put OpenFlow support at the cornerstone of its strategy, the company performed what it termed *a big pivot* in 2013 [27]. We described some of Big Switch's history, as one of the major players in the ecosystem, in Section 9.4. The company continues to offer the same set of commercial application, controller, and switch software products that it has all along. The pivot relates to selling them as bundles that white-box switches can download and self-configure. Big Switch's original product strategy included getting a number of switch vendors to become OpenFlow-compatible by facilitating this migration with the free switch code, thus creating a southbound ecosystem of switch vendors compatible with its controller. Big Switch also encouraged application partners to develop applications to the northbound side of its controller. Such applications could be in the areas of firewalls, access control, monitoring, network virtualization, or any of a long list of other possibilities. The amount of investments Big Switch received [10,11] indicates that it was very successful at convincing others that its business model is viable.

After a rocky experience in trying to get production networks configured with different vendors' switches to interoperate with its controller and applications, Big Switch concluded that it needed to make the rollout of the solution much easier than what it was experiencing with all but its most sophisticated customers. Big Switch concluded that by partnering instead with white-box ODMs rather than established switch vendors like Juniper, Arista, and Brocade, it could better control the total user experience of the solution. This new approach, or pivot, has Big Switch's customers purchasing white-box switches from Big Switch's ODM partners. These white-box switches will be delivered with a boot loader that can discover the controller and download Big Switch's Switch Light OpenFlow switch code [26]. The notion is to sell the customer an entire bundle of switch, controller, and application code that is largely autoconfigured and that provides a particular solution to the customer. There were two initial solution bundles offered at the time of the pivot: a network monitoring solution and a cloud fabric for network virtualization. Significantly, the cloud fabric solution is not based on an overlay strategy but instead on replacing all the physical switches with white-box switches running Big Switch's software. Big Switch is betting that by replacing the physical network with the low-cost white-box switches and by having Big Switch stand behind the entire physical and virtual network as the sole provider, the company will have created a compelling alternative to the many overlay alternatives being offered for network virtualization.

12.9.2 Non-OpenFlow White-Box Ventures

Cumulus Networks takes the white-box boot loader concept and generalizes it one step further than Big Switch. As of this writing, Cumulus offers the product closest to the concept of *opening up the device*, described in Section 6.4. As we pointed out in our discussion of white-box switches in Section 9.5, any switching code, OpenFlow or non-OpenFlow, controller-based or not, can be loaded into the Cumulus switches if written to be compatible with the Cumulus bootloader. Cumulus is a switch-centric company and does not offer the broad portfolio of controller and application software, as does Big Switch.

12.9.3 An OpenFlow ASIC?

Another OpenFlow hardware opportunity is to create a switching ASIC that is specifically designed to support OpenFlow 1.3 and beyond. Implementers have found it challenging to fully implement some parts of the advanced OpenFlow specifications in existing ASICs. One such challenge is supporting multiple, large flow tables. Since there is growing industry demand for the feature sets offered by OpenFlow 1.3, this creates an opportunity for semiconductor manufacturers to design an OpenFlow chip from scratch. Some of the incumbent switching chip manufacturers may be slow to displace their own advanced chips, but this is an inviting opportunity for a startup chip manufacturer. In Section 9.6 we described work being done at Mellanox that may be leading to such an advanced OpenFlow-capable ASIC. Intel has also announced switching silicon [28] that is explicitly designed to handle the multiple OpenFlow flow tables.

12.9.4 Data Center Network Virtualization

The team at ConteXtream has used its grid-computing heritage as the basis for its distributed network virtualization solution [12]. This is being touted as an SDN product, since the routing of sessions in the data center is controlled by a distributed solution with global knowledge of the network. This solution is very different from the classical OpenFlow approach of centralizing the network routing decisions to a single controller with a small number of backup controllers. In the ConteXtream approach, the control is pushed down to the rack level. If a control element fails, only a single rack fails, not the entire network. This per-rack control element is added to the TOR server. Admittedly, this architecture deviates significantly from the more classical SDN approaches of Big Switch or Nicira. In fact, a distributed algorithm with global knowledge of the network sounds similar to the definition of OSPF or IS-IS. The ConteXtream solution does differ significantly from classical routing in that (1) it can perform switching at the session level, (2) network devices such as firewalls and load balancers can be virtualized into their software, and (3) the network administrators have explicit control over the path of each session in the network.

As shown in Table 12.3, PLUMgrid has also received a sizable investment to develop network virtualization solutions. PLUMgrid offers an SDN-via-overlays solution that is designed for data centers and integrates with VMware ESX as well as with *Kernel-based Virtual Machine* (KVM). PLUMgrid does not use OpenFlow for directing traffic through its VXLAN tunnels but instead uses a proprietary mechanism. It also uses a proprietary virtual switch implementation rather than using OVS or Indigo. PLUMgrid indicates that OpenFlow might be supported in the future [15].

Midokura [16] is another well-funded startup attacking the network virtualization market. Midokura currently integrates with OpenStack and in the future plans to include support for other cloud-software platforms such as CloudStack. Like other network virtualization companies, Midokura's product, MidoNet, creates virtual switches, virtual load balancers, and virtual firewalls. MidoNet is an SDN-via-overlays product that runs on existing hardware in the data center. MidoNet claims to allow data center operators to construct public or private cloud environments through the creation of thousands of virtual networks from a single physical network. Since managing this plethora of virtual networks using traditional configuration tools is a huge burden, MidoNet provides a unified network management capability that permits simple network and service configurations of this complex environment. As shown in Table 12.3, Midokura's lead investors are large Japanese institutions, and some of the senior managers are battle-hardened data communications veterans. Midokura is further evidence of the enthusiasm in Japan for SDN, as is reflected by the incumbents NTT and NEC's interest in this technology [20,21].

12.9.5 WAN Network Virtualization

Pertino's offering is very different from the aforementioned data center-focused companies. Pertino believes that there is a large market for *SDN via cloud*. Pertino offers a cloud service that provides WAN or LAN connectivity to organizations that want to dynamically create and tear down secure private networks for their organizations [17]. The term SDN via cloud is another stretch from the original meaning of the term SDN, but, indeed, this concept truly is a Software Defined Network in the sense that software configuration changes in Pertino's cloud spin up and spin down its customers' private networks through simple configuration changes. The only technical requirement for the individual users of these virtual networks is that they have Internet connectivity. A handful of branch offices from a small enterprise can easily obtain a secure, private network via the Pertino service as long as each of those branch offices has Internet connectivity. Such a private network can be brought online or offline very easily under the Pertino paradigm. Pertino also attempts to make this service easy to use from a business perspective. This is another IaaS proposition. The company's business model is a usage-based service charge, where the service is free for up to three users and after that there is a fixed monthly fee per user.

12.9.6 Network Functions Virtualization

Embrane has a distributed software platform for NFV that runs on commodity server hardware. The company's focus is to virtualize load balancers, firewalls, VPNs, and WAN optimization through the use of *distributed virtual appliances* (DVAs) [14]. Considering that a typical physical server in a data center hosts multiple customers with a pair of firewalls for each customer, the number of physical devices that can be removed by deploying the Embrane product, called Heleos, is significant [13]. The company emphasizes that Heleos functions at network layers four through seven, unlike the OpenFlow focus on layers two and three. Although Heleos does work with OpenFlow, it is not required. Indeed, Heleos integrates virtualization technology from any hypervisor vendor. One of Embrane's goals is to allow Heleos to coexist with existing network appliances to permit a gradual migration to the virtualized appliance model. In keeping with the growth of IaaS, in addition to the standard annual subscription fee, Embrane supports a usage-based model that charges an hourly rate for a certain amount of available bandwidth.

Pluribus's offering is based on its server-switch hardware platform along with its NetVisor network operating system. The company's target market is network virtualization of public and private clouds. The Pluribus product, like that of others mentioned above, provides services such as load balancing and firewalls that formerly required independent devices in the data center [15]. The Pluribus SDN controller is integrated with its server switches such that the company's solution is completely distributed across the Pluribus hardware. The Pluribus solution also provides interoperability with OpenFlow controllers and switches by interfacing with the integrated Pluribus SDN controller. It also is compatible with the

VMware NSX controller. The combination of the NetVisor network operating system and the company's proprietary server switches attempts to provide a complete fabric for handling the complex cloud environment that requires coordination among applications, hypervisors, and the compute virtualization layer [17].

12.9.7 Optical Switching

Optical switches have been around for more than a decade now. They offer extremely high-bandwidth circuit-switching services but have been comparatively unwieldy to reconfigure for dynamic networking environments. The SDN concept of moving the layer two and layer three control plane to a separate controller translates readily to the layer one control plane appropriate for optical switches. A couple of SDN startups, Plexxi and Vello, are attempting to leverage this natural SDN-optical synergy into successful business ventures. Plexxi has a dual-mode switch that is both an Ethernet and an optical switch. The optical ports interconnect the Plexxi switches. The fact that each Plexxi switch is actually an optical switch allows extremely high-bandwidth, low-latency *direct* connections between many Plexxi switches in a data center. The Plexxi controller platform provides SDN-based network orchestration. This controller has been used to configure and control pure optical switches from other manufacturers such as Calient [19]. In this combined offering, the Plexxi switch's Ethernet ports are used for short, bursty flows, and the Plexxi optical multiplexing layer is used for intermediate-sized flows. High-volume, persistent flows (elephant flows) are shunted to the dedicated Calient optical switch.

Vello also offers Ethernet and optical switches. The Vello products are OpenFlow-enabled themselves and can interface to other vendors' OpenFlow-compatible switches [15]. Though Vello, like most of the other startups discussed above, addresses network virtualization, it focuses on use cases and applications related to storage devices.

12.9.8 Mobility and SDN at the Network Edge

Current SDN efforts have focused primarily on the data center and carrier WAN, but Tallac Networks is expanding that focus with an emphasis on the wireless access network. Tallac technology offers a new approach called *software defined mobility* (SDM), which enables wireless operators to connect mobile users to the network services they demand. One powerful aspect of OpenFlow is the ability to virtualize a network so it can be easily shared by a diverse set of network services. In the context of *WiFi-as-a-Service* (WaaS), this means creating a multitenant WiFi environment. The fine-grained control of the OpenFlow protocol allows network connections and secure tunnels to be provisioned dynamically. Through a set of technologies combining OpenFlow, embedded application containers for dynamic execution environments and standards-based Hotspot 2.0 (Passpoint), Tallac delivers an SDN-enabled, multitenant, service-oriented WiFi network as the next-generation hotspot for any wireless LAN operator. Combined with WaaS business frameworks, these new standards help wireless operators embrace the never-before-seen diversity of mobile users, and they help mobile users connect effortlessly to never-before-seen wireless networks. The current set of standards, however, assumes that wireless operators will forge and maintain complex agreements with multiple service providers in order to allow the providers' customers to access their services over WiFi. The unique combination of Tallac technology with Hotspot 2.0 will simplify this process and allow on-demand service provisioning.

12.10 Career Disruptions

Most of this chapter has focused on the potential disruption SDN is likely to cause to the incumbent NEMs as well as the opportunities it affords startup enterprises. We need to remember that the driving force inspiring this new technology was to simplify the tasks of the network administrators in the data center. Most of the marketing materials for the startups discussed in this chapter declare that the current network administration environment in the ever-more-complex data center is spiraling out of control. One consequence of this complexity has been a seemingly endless demand for highly skilled network administrators capable of the black art of data center network management. If *any* of the SDN hype presented in this chapter proves to be real, one certain outcome is that there will be less need for highly skilled, technology-focused network engineers to manually reconfigure network devices than there would be without the advent of SDN [13]. For example, consider that one of the highly sought-after professional certifications today is that of the *Cisco Certified Internetwork Expert* (CCIE). This sort of highly targeted, deep networking knowledge will become less and less necessary as SDN gains a larger foothold in the data center [14]. In [23], the author cites the VLAN provisioning technician as an example of a particularly vulnerable career as SDN proliferates.

However, there will be a growing need in the data center for the kind of professional capable of the sort of agile network service planning depicted in Figure 12.2. Much like the *Agile Development Methodology* that has turned the software development industry on its head in recent years, the network professionals in an SDN world will be required to rapidly move through the never-ending cycles of *plan*, *build*, *release*, and *run* shown in Figure 12.2 in order to keep pace with customers' demands for IaaS. *DevOps* is the term now commonly used for this more programming-aware IT workforce that will displace the traditional IT network engineer. This new professional will possess a stronger

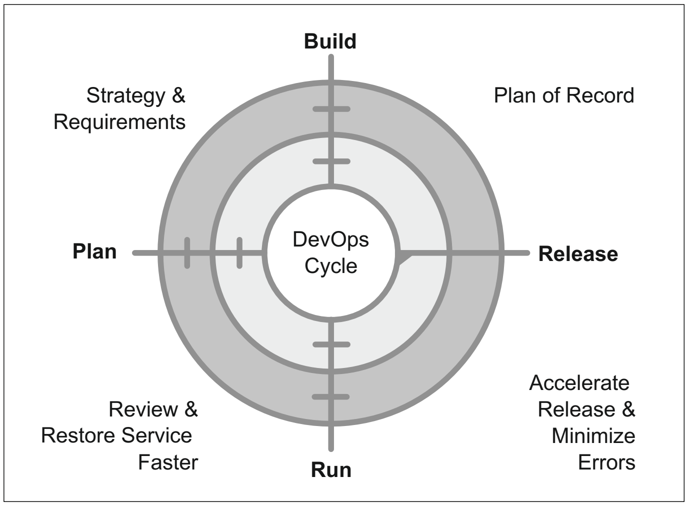


FIGURE 12.2

Agile environment for network administrators.

development operations background than the traditional CLI-oriented network administrator [24]. Since SDN increases how much innovation IT professionals will be able to apply to the network, professionals working in this area today need to embrace that change and ensure that their skills evolve accordingly.

12.11 Conclusion

There is a historical pattern with new Internet technologies whereby the hype surrounding the possibilities and financial opportunities reaches a frenzy that is out of proportion to its actual impact. We have seen this happen numerous times over the past two decades. FDDI and WIMAX are but two examples of this trend. Though it may be likely that the current hype surrounding SDN is probably near its peak [4], it does seem inevitable that some form of SDN will take root. In particular, de-embedding the control plane from the physical switch for data center products seems to be an unstoppable trend. The technical underpinnings of this idea are very sound, and the amount of investment that has poured into this area in the past few years is ample evidence of the confidence in this technology. The big question from a business ramification standpoint is whether or not there will be a sea change in the relationship between vendors and customers, or will the degree of vendor lock that the large NEMs enjoy today persist in a slightly modified form? Idealistic SDN proponents proselytize that the adoption of SDN will take the power from the NEMs and put it in the hands of customers. Whether or not this happens largely hinges on whether or not openness is truly a mandatory part of the SDN paradigm. In the business world today there is no consensus on that, and we acknowledge that separation of control plane from data plane need not be performed in an open manner. It is entirely possible that proprietary incarnations of this concept end up dominating the market. This is still an open question, and the answer to it will undoubtedly be the biggest determinant of the business ramifications of SDN.

In the next and final chapter we attempt to synthesize the current state of SDN research with current market forces to forecast where SDN is headed in the future.

References

[1] Duffy J. Cisco lays off 500: company realigning operations, investments on growth areas. Network World; March 27, 2013. Retrieved from <www.networkworld.com/news/2013/032713-cisco-layoffs-268165.html>.

[2] Williams A. VMware buys Nicira for $1.26 billion and gives more clues about cloud strategy. Techcrunch; July 23, 2012. Retrieved from <techcrunch.com/2012/07/23/vmware-buys-nicira-for-1-26-billion-and-gives-more-clues-about-cloud-strategy>.

[3] Hesseldahl A. Cisco keeps up acquisition pace with $141 million cariden buy. All Things D; November 29, 2012. Retrieved from <allthingsd.com/20121129/cisco-keeps-up-acquisition-pace-with-141-million-cariden-buy>.

[4] Hachman M. Brocade buys Vyatta for SDN Tech. Slashdot; November 6, 2012. Retrieved from <slashdot.org/topic/datacenter/brocade-buys-vyatta-for-sdn-tech>.

[5] Higginbotham S. Juniper to buy SDN startup contrail in deal worth $176M. Gigaom; December 12, 2012. Retrieved from <gigaom.com/2012/12/12/juniper-to-buy-sdn-startup-contrail-in-deal-worth-176m>.

[6] Bort J. Cisco launches its secret startup insieme, then buys it for $863 million. Business Insider; November 6, 2013. Retrieved from <www.businessinsider.com/cisco-buys-insieme-for-863-million-2013-11>.

[7] Malik O. Here is why Cisco bought Meraki for $1.2 billion in cash. Gigaom; November 18, 2012. Retrieved from <gigaom.com/2012/11/18/cisco-buys-meraki-for-1-2-billion-in-cash-here-is-why>.

[8] Heavy reading rates the 10 most promising SDN startups. InvestorPoint; October 29, 2012. Retrieved from <www.investorpoint.com/news/MERGERAC/55405881>.

[9] Rath J. Pica8 launches open data center framework data center knowledge; April 15, 2013. Retrieved from <www.datacenterknowledge.com/archives/2013/04/15/pica8-launches-open-data-center-framework>.

[10] Williams A. Big switch raises $6.5M from Intel Capital, gains attention for next generation networking, challenges Cisco. Techcrunch; February 8, 2013. Retrieved from <techcrunch.com/2013/02/08/big-switch-raises-6-5m-from-intel-capital-gains-attention-for-next-generation-networking-challenges-cisco>.

[11] Wauters R. OpenFlow startup big switch raises $13.75M from index, Khosla ventures. TechCrunch; April 22, 2011. Retrieved from <techcrunch.com/2011/04/22/openflow-startup-big-switch-raises-13-75m-from-index-khosla-ventures>.

[12] Higginbotham S. ConteXtream joins the software defined networking rush. Gigaom; December 13, 2011. Retrieved from <gigaom.com/2011/12/13/contrextream-joins-the-software-defined-networking-rush>.

[13] Higginbotham S. Embrane's virtual network appliances for an SDN world. Gigaom; December 11, 2011. Retrieved from <gigaom.com/2011/12/11/embranes-virtual-network-appliances-for-an-sdn-world>.

[14] Berndtson C. Startup embrane looks beyond the SDN trend. CRN; July 20, 2012. Retrieved from <www.crn.com/news/networking/240004082/startup-embrane-looks-beyond-the-sdn-trend.htm>.

[15] Guis I. SDN start-ups you will hear about in 2013: snapshots of Plexxi, Plumgrid, Pluribus and Vello. SDN Central; January 24, 2013. Retrieved from <www.sdncentral.com/companies/sdn-start-ups-you-will-hear-about-in-2013-pt-2/2013/01>.

[16] Etherington D. Midokura scores $17.3M series a to ramp up its network virtualization offering on a global scale. Techcrunch; April 1, 2013. Retrieved from <techcrunch.com/2013/04/01/midokura-scores-17-3m-series-a-to-ramp-up-its-network-virtualization-offering-on-a-global-scale>.

[17] Clancy H. Pertino serves up small-business networks in the cloud. ZDNet; March 29, 2013. Retrieved from <www.zdnet.com/pertino-serves-up-small-business-networks-in-the-cloud-7000013277>.

[18] Burt J. SDN market to grow sixfold over 5 years: Dell'Oro. eWeek; November 4, 2013. Retrieved from <mobile.eweek.com/blogs/first-read/sdn-market-to-grow-six-fold-over-5-years-delloro.html>.

[19] Lightwave Staff. Calient, Plexxi bring SDN to large-scale data center networks. Lightwave; March 19, 2013. Retrieved from <www.lightwaveonline.com/articles/2013/03/calient–plexxi-bring-sdn-to-large-scale-data-center-networks.html>.

[20] Pica8 partners with NTT data for end-to-end SDN solutions. BusinessWire; April 2, 2013. Retrieved from <www.businesswire.com/news/home/20130402005546/en/Pica8-Partners-NTT-Data-End-to-End-SDN-Solutions>.

[21] NEC displays SDN leadership at PlugFest. InCNTRE, Indiana University; October 12, 2012. Retrieved from <incntre.iu.edu/featured/NEC-displays-SDN-leadership-at-PlugFest>.

[22] Palmer M. SDNCentral exclusive: SDN market size expected to reach $35B by 2018. SDN Central; April 24, 2013. Retrieved from <www.sdncentral.com/market/sdn-market-sizing/2013/04>.

[23] Pepelnjak I. SDN's casualties. ipSpace; June 12, 2013. Retrieved from <blog.ioshints.info/2013/06/response-sdns-casualties.html>.

[24] Knudson J. The ripple effect of SDN. Enterprise networking planet; October 22, 2013. Retrieved from <www.enterprisenetworkingplanet.com/netsysm/the-ripple-effect-of-sdn.html>.

[25] Burt J. Cisco may Unveil Insieme SDN Technology November 6. eWeek; September 24, 2013. Retrieved from <www.eweek.com/blogs/first-read/cisco-may-unveil-insieme-sdn-technology-nov.-6.html>.

[26] Kerner S. Big switch switches SDN direction, staying independent. Enterprise Networking Planet; September 11, 2013. Retrieved from <www.enterprisenetworkingplanet.com/datacenter/big-switch-switches-sdn-direction-staying-independent.html>.

[27] Duffy J. Big switch's big pivot. Network World; September 11, 2013. Retrieved from <www.networkworld.com/community/blog/big-switchs-big-pivot>.

[28] Ozdag R. Intel ethernet switch FM6000 series: software defined networking. Intel white paper. Retrieved from <www.intel.com/content/dam/www/public/us/en/documents/white-papers/ethernet-switch-fm6000-sdn-paper.pdf>.

This page is intentionally left blank

CHAPTER

13 SDN Futures

An accurate forecast of the future of any nascent technology calls for a better crystal ball than what is usually available. To attempt such a look into the future requires that we first lift ourselves up out of the details of SDN on which we have focused in the latter half of this book and view the current state of SDN affairs with a broader perspective. We need to consider what lies in store for SDN as a whole as well as individually for each of the SDN alternatives described in Chapter 6. We are optimistic about the future of SDN. Our optimism for the future of SDN is partly founded on the belief that there are many potential areas of application for the technology that have yet to be seriously explored. For that reason, we dedicate a significant part of this chapter to exploring just a few of the novel use cases and research areas for Open SDN that are beginning to attract attention. Before that, however, we should acknowledge that not all the 2013 press releases about SDN were filled with unbridled optimism. In the next section, we look at the turbulence and some of the pessimism surrounding SDN.

13.1 Current State of Affairs

The turbulence that has surrounded the SDN movement during these early years has closely tracked the well-known *Gartner Hype Cycle* [1]. We depict a generic version of this hype cycle in Figure 13.1. For SDN, the *technology trigger* was the birth of OpenFlow and the coining of the term *Software Defined Networks* in 2009.[1] The MAC table and VLAN ID exhaustion we have discussed in earlier chapters were manifesting themselves in the data center in the years leading up to 2012. This provided hard evidence that networking was at the breaking point in the data center. The fact that SDN provides a solution to these problems added fuel to the SDN fire. The *peak of inflated expectations* was reflected in the flurry of acquisitions in 2012 and the sizable VC investments preceding and coinciding with that. The *trough of disillusionment*, in our opinion, occurred in early 2013.

Big Switch's fate leading up to 2013 and then in the first half of that year mirrored these trends. The company's withdrawal first from board status of the OpenDaylight consortium and subsequent total exit reflected its feeling that forces other than a commitment to Open SDN were influencing that organization's choice of controller technology. That was followed by Big Switch's *big pivot*, which was an admission that its early strategy of open platforms needed to be retooled to be plug-and-play solutions for a broader customer base (see Section 12.9.1). Big Switch's moves during 2013 were reflective of a more general fragmentation of the SDN movement.

[1]When we refer to the first use of the term *Software Defined Networks*, we mean in the specific context in which it is used in this book and in conjunction with the work surrounding OpenFlow at Stanford University. The term itself was used previously in patents in the 1980s.

Software Defined Networks. http://dx.doi.org/10.1016/B978-0-12-416675-2.00013-9

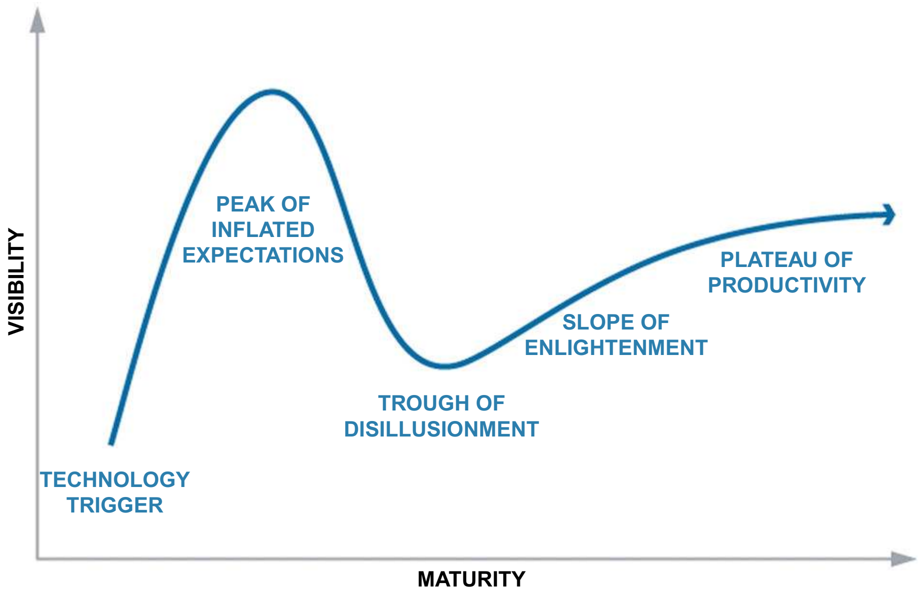


FIGURE 13.1

Gartner hype cycle. ((Courtesy Gartner, Inc.) [1]).

Many industry watchers feel that the degree of SDN washing that now occurs means that SDN can mean anything to anyone. Indeed, one of the founding fathers of SDN listed in Section 9.1.1, Martin Casado, was quoted in April 2013 as saying, “I actually don’t know what SDN means anymore, to be honest” [2]. If this is the feeling of one of the core group that coined the very term SDN, then one might reasonably fear that SDN has lost much of its original focus.

The Gartner Hype Cycle does not end with the trough of disillusionment, however, nor does Open SDN’s story begin to fall apart in 2013. Just as the Hype Cycle predicts that disillusionment is followed by the *slope of enlightenment* and then by the *plateau of productivity*, so do we now begin to see realistic assessments for where SDN will be applied. We believe that this slope of enlightenment in SDN’s future reflects a middle ground between the extremes of belief that it will totally unseat the incumbent NEMs and traditional switching technology and the belief that it is largely an academic exercise with applicability restricted to the 1% of customers sophisticated enough to grapple on their own with the technological change. As an example, Big Switch’s pivot has moved the company toward a strategy based on OpenFlow-enabled white-box switches. Thus, whereas Big Switch’s strategy is more tactical than it had been, the company is still betting on the long-term future of OpenFlow. Furthermore, SDN has already established a beachhead in the data center that demonstrates that it is there to stay. SDN experts from HP, Dell, and NEC debating the future of SDN in [3] caution against impatience with the rate of penetration of SDN. They point out that server virtualization in the data center has existed for over a decade and only now is reaching 50% market penetration.

We believe that for SDN to transition from the slope of enlightenment to the plateau of productivity, we must first see evidence of an entirely new generation of switching ASICs that have been purposely built with SDN and OpenFlow in mind. Switch ASICs may take two years or more to mature and reach production. Perhaps the amount of flow table space required will mandate new silicon geometries, and the time to availability may be even longer.

The integration of SDN into the work being performed by the IEEE and other important forums is further evidence of its growing acceptance. Nonetheless, the exact forms of SDN that are here to stay are still debated. In this chapter we discuss some of the areas of research and experimental use cases that provide evidence of the totally unique ways that SDN can address networking problems. The slope of enlightenment embodies a recognition that SDN solves certain networking problems better than any other known technology and that it will indeed be disruptive there. This is a strong vindication of SDN but falls short of being the death knell for traditional networking, as some have argued. It is likely that the reduction in hype will reduce the appeal of applying the SDN moniker to every new networking variant, and the different versions of SDN that gain market traction, though not all OpenFlow-based by any means, will tend increasingly to those characteristics that we used to define Open SDN in Section 4.1.

Among those basic characteristics, one wildcard is that of *openness*. Openness means different things to different members of the networking ecosystem. For an academic researcher, it means open source available to all. On the other extreme, openness to a NEM may mean exposing certain APIs to your product in order to allow user-written applications to fine-tune external control knobs that the NEM exposes. (These two extremes reflect our earlier definitions of Open SDN versus SDN via APIs.) A recent special report [17] on the history and demise of the *Open Systems Interconnection* (OSI) standards concluded with the following remark: “Perhaps the most important lesson is that ‘openness’ is full of contradictions. OSI brought to light the deep incompatibility between idealistic visions of openness and the political and economic realities of the international networking industry. And OSI eventually collapsed because it could not reconcile the divergent desires of all the interested parties.” This lesson, learned the hard way for OSI, should not be lost on Open SDN evangelists. Too strict an interpretation of openness in the definition of SDN could result in a similar fragmentation of interests between the open source community and industry. If major parts of industry cannot align behind the Open SDN movement, the movement will morph into something that is synergystic with industry’s goals or it will be relegated to the halls of academia. This is the *realpolitik* of the Internet.

13.2 Potential Novel Applications of Open SDN

There is considerable active research related to SDN as well as many novel use cases being proposed that illustrate potential new areas where SDN can be applied or improved. In this section we present a sampling of some of the areas that promise to increase the scope of applications of Open SDN and thus broaden the foundation of its future.

A full survey of SDN-related research is beyond the scope of this work. A number of recent conferences have focused on SDN research, including [14,15]. The proceedings of these conferences provide a good review of current research trends in this area. The survey of the past, present, and future of SDN found in [16] may also provide the reader with broader coverage of current research directions in this field.

Here we have selected a number of areas of potential application of Open SDN with the intent of giving the reader a flavor of the areas of application that radically depart from those that we have described during the bulk of this work. It is our hope that this discussion will underscore that this new network programming paradigm opens the doors to many possibilities, only a few of which have already been explored.

13.2.1 Managing Nontraditional Physical Layer Links

There is growing interest in using OpenFlow to control devices that have flows but are not traditional packet switches. The two most promising areas involve flows over optical and wireless links. In Section 8.7.1 we presented a use case in which SDN was used for offloading elephant flows onto optical devices. This represents one example of the general area of using OpenFlow to manage flows over physical layers that are outside the domain of classical LANs and WANs. In addition to the example in Section 8.7.1 and in Section 12.9.7, we mentioned two startups, Plexxi and Vello, that have brought products to market that marry optical switching with OpenFlow.

In [5], the researchers formalize the study of how to map big data applications to an OpenFlow network. An elephant flow is an example of a big data application. The authors propose mechanisms to build on the existing Hadoop[2] job-scheduling method to optimize how and when to schedule jobs over the optical links. They point out that unlike the more classical SDN use cases of cloud network provisioning and WAN traffic engineering, such as those discussed in Chapters 7 and 8, such big-data jobs require more rapid and frequent updates of the flow tables on the switches. They conclude that current OpenFlow switches are capable of handling their predicted number and frequency of flow table changes. They believe that there may be challenges in keeping the various flow tables across the network switches synchronized with this rate of flow table changes. This is important because one of the problems in traditional networks that we described in Section 1.5.3 was the relatively slow convergence time in the face of routing changes. At the fine granularity of flow table changes being made to route these big-data flows, slow convergence times may be an issue for optical offload.

Because of the ease of installing wireless backhaul links to bring mobile traffic back to the core from *radio access network* (RAN) cells, wireless backhaul is becoming increasingly popular. One of the challenges in doing this is that the effective bandwidth of a wireless link is not constant, as in its wired counterpart. In [4, Section 8.4], OpenFlow is proposed as a mechanism to segregrate traffic from different providers and different types into separate flows that are then transmitted over a single shared wireless backhaul medium. In the example, the wireless backhaul technology is 802.16. Although the segregation of different traffic types for differential QoS treatment is well established, in the example the wireless backhaul link is potentially shared by a number of different colocated RAN technologies (e.g., LTE, 3G, WiFi) that may be offered by several different operators. The wireless backhaul provider may have different SLAs with each of those operators and for each operator different levels of service, depending on the terminating RAN technology, traffic type, or business agreement. Since the wireless backhaul bandwidth capability itself may vary due to environmental conditions, the process of satisfying all of the various SLA commitments becomes far more complex and can benefit from OpenFlow's ability to segregate the traffic into different flows and route or police those flows dynamically with the changing wireless conditions.

[2]Hadoop is open source software that enables the distributed processing of large data sets across clusters of commodity computers.

13.2.2 Applying Programming Techniques to Networks

One of the themes of this work has been that the greater network programmability inherent in OpenFlow provides benefits in the richness of policies and in the fine-grained control it offers the network programmer. To be fair, though, when a programming environment gives the programmer a lot of power, the probability of misprogramming the network is likely greater than the legacy situation where the network engineer was constrained by control knobs of limited functionality. Thus, part of the ultimate success of OpenFlow is the development of tools that will aid in detecting such programming flaws.

More formally, in [18] the author suggests that Open SDN provides a proper abstraction of the network that allows us to address networking challenges more efficiently. The author draws an anology to the difference between programming a CPU in today's high-level languages and application development environments versus programming in machine language. A corollary is that the network abstraction allows other advanced programming methodologies to be applied, including debuggers, analysis tools, network simulation, verification tools, and others. These tools have enabled tremendous progress in the development of software because of formal analysis and the application of theory. No equivalents have been practical in the networking space due to lack of the abstraction as described in [18]. We consider here some of the tools that are enabled by this new way of looking at networking.

Network Debugger (ndb)

In Section 4.1.3 we drew an analogy between the control plane of networks and computer languages and CPUs. We explained that we no longer work at the level of assembly language because high-level languages provide us with efficient abstractions to program the CPUs effectively. We argued that the network needs a similar evolution toward a high-level abstraction of network programming, and we suggested that OpenFlow is a good example of such an abstraction. Following this same analogy, if OpenFlow provides us with a way of programming the network as a single entity, can we now consider other advanced programming techniques and tools for application in the network? For example, if we view the network as a vast distributed computer, can we envision a debugger for that computer? This is precisely what the work performed in [6] attempts. The authors propose a *network debugger* (ndb) that will implement basic debugger functionality at the packet level. They aspire to implement a full set of debugger actions, including *breakpoint*, *watch*, *backtrace*, *single-step*, and *continue*. In [6] the authors describe an example whereby they have implemented both breakpoints and packet backtraces. For instance, a breakpoint could be set in a network switch such that when a packet matches no flow entry in that switch, the breakpoint is hit. At that point the network programmer could request a packet backtrace showing which flow entries were matched by which switch that led to the packet arriving to this switch where no flow entry matched. The fact that such technology can even be contemplated underscores the more deterministic network behavior possible under the centralized programming model of Open SDN.

No Bugs in Controller Execution (NICE)

It is also naïve to believe that a controller programming a large set of switches that are in a constant state of forwarding packets is an *entirely* deterministic system. The architecture of OpenFlow does not tightly synchronize the programming of multiple switches when the controller must simultaneously program flow entries in several switches. Due to the entropy involved in this process, it might not be possible to ensure that all flow entries are in place before the packets belonging to that flow begin to arrive. Thus, race conditions and even routing loops are possible for short periods of time. Although it is likely impossible to completely eliminate this hysteresis during rapid network reprogramming, a tool that could model the effects of a particular controller program (i.e., an application) could help minimize such occurrences and ensure that their effects are short lived. In [7] the authors propose *No Bugs in Controller Execution* (NICE), a tool for modeling the behavior of OpenFlow applications to ensure their correctness, including detecting forwarding loops and black holes. Advanced modeling techniques are used to create input scenarios that test all possible execution paths within the application while taking account of variability of the state of the network switches and links. In [7] the authors modeled and debugged three sample applications: a MAC-learning switch, an in-network server load balancer, and energy-efficient traffic engineering.

Veriflow

Although there is great appeal to checking for OpenFlow application correctness offline and before deployment, it may be necessary or desirable to perform real-time formal checking for correct network behavior. In [8] the authors describe Veriflow, a system that resides between the controller and the switches and verifies the correctness of each flow entry update before it is applied. One advantage of the approach described in [8] is that it does not require knowledge of all the network programs themselves, since, it verifies correctness based on observations of flow rules as they are sent from the controller to the switches. One of the biggest challenges faced in this study is to keep the latency of the correctness checks low enough to avoid becoming a bottleneck in the path between the controller and the switches. Such speed is not possible if one were to take a brute-force approach to reflecting the global network state, which is very complex and changes rapidly. The authors claim to have developed algorithms that allow such checks to be performed at hundred-microsecond granularity, which is sufficiently fast to avoid adversely affecting network performance. Their novel work is related to how to represent the portion of network state that could possibly be affected by a given rule change and to develop algorithms that reduce the search space such that the effect of a rule change on network state can be ascertained in real time.

13.2.3 Security Applications

Hiding IP Addresses

Many network attacks are based on identifying active IP addresses in a targeted domain. Protecting hosts by making it impossible for an attacker to identify a host's IP address would be an effective countermeasure. In [9] the authors propose that each protected host be assigned a virtual IP address that is the one exposed to the outside world by DNS lookups. In this method, the OpenFlow controller randomly and at high frequency assigns the virtual IP address to the protected hosts, maintaining the temporary mapping of virtual to physical IP addresses. Earlier systems based on DHCP and NAT exist to change the physical IP address of hosts, but these mechanisms change the IP address too infrequently to be a strong defense. In [9] only authorized hosts are allowed to penetrate through to the physical IP address. The translation from the virtual IP address to the physical IP address happens at an OpenFlow switch immediately adjacent to the protected hosts. The unpredictability and speed of the virtual IP address mutation is key to thwarting the attacker's knowledge of the network and the planning of the attacks. Although this approach could conceivably be implemented using specialized appliances in a conventional network, OpenFlow provides a flexible infrastructure that makes this approach tractable.

Segregating IPSec Traffic in Mobile Networks

Wireless providers using LTE secure the user and control data between the base station (eNB) and their network core using IPSec tunnels. These tunnels terminate in the core at a secured gateway (S-GW). As currently specified, there is a single IPSec tunnel encrypting all services. IPSec tunneling carries significant overhead, and significant efficiency gains are possible if traffic that does not require the security afforded by IPSec can be sent in the clear. In two separate proposals [4, Section 8.12] and [4, Section 8.5] the authors suggest using OpenFlow to map between different traffic types and the appropriate level of IPSec security. In [4, Section 8.5] the focus is on making the mapping of individual flows to different IPSec tunnels. In [4, Section 8.12] the emphasis is on distinguishing those traffic types requiring security and to only tunnel those. The authors suggest that YouTube videos, social media, and software updates are examples of traffic that does not require IPSec encryption and can thus be sent in the clear. Though the authors do not describe how such traffic types would be detected, one possible means of doing so that avoids the deep packet inspection problem is to base the decision on server addresses that belong to YouTube, Facebook, Microsoft software updates, and so on. An additional possible benefit of providing flow-specific treatment to these other traffic types is that it is not always necessary to send separate copies of this kind of traffic from the network core. By inserting a *deduplication* processor closer to the mobile users, this redundant traffic can be cached at that location, and a twofold bandwidth savings is realized by eliminating the round trip to the S-GW. By providing the operators control over which flows are tunneled and which are not, this system allows operators to monetize their investment by offering different plans with different levels of security.

13.2.4 Roaming in Mobile Networks

Mobile Traffic Offload

In Chapter 8 we presented a number of SDN applications involving traffic steering, where flows are directed toward security systems or load balanced among a set of similar-function servers. In the multi-radio environment common for today’s mobile operators, a new SDN application is possible in the area of mobile traffic offload. Mobile offload means moving a client *mobile node* (MN) from one RAN to another. This might make sense for a number of reasons, but the one most often offered is to shunt the traffic to a RAN where the spectrum is more available or less expensive than the one currently used by the MN. Such offloading has been contemplated for some time by mobile operators, but existing approaches have not provided the flexible, fine-grained control offered by Open SDN. In [4, Section 8.3] a use case is described where OpenFlow switches are used in key gateways in the 3GPP architecture. Based on observing flow-related criteria and the location of the MN, an OpenFlow application can redirect the MN’s access connection from 3G to a WiFi hotspot, for example. If the MN needs to roam from WiFi to a cellular radio technology, the same mechanism may be used. This approach allows operators to flexibly and dynamically apply offloading policies rather than static policies that are not able to adapt to changing network conditions. For example, if a user is currently connected to a heavily loaded WiFi hotspot and located within a lightly loaded LTE cell, it may make sense to do reverse offload and move the user from WiFi back to LTE. Such a decision hinges on observing rapidly changing conditions and could not be put into effect based on static policies. Note that though the basic traffic steering inside the 3GPP packet network is native to OpenFlow, the control signaling related to the RF aspects of the roam would have to be coordinated with the appropriate 3GPP signaling functions, since those are outside the current scope of OpenFlow.

Media-Independent Handovers

IEEE 802.21 is an established protocol for media-independent handovers between 802-family *points of access* (PoAs). Examples of PoAs are 802.11 access points and 802.16 base stations. The 802.21 *point of service* (PoS) is responsible for the messaging to the PoAs to accomplish either make-before-break or break-before-make handovers (roams). A significant part of accomplishing such handovers is to redirect the traffic flow from an MN such that it enters and exits the network from the new PoA. This, combined with the fact that OpenFlow is natively media-independent, leads to a natural overlap between the role defined for the 802.21 PoS and an OpenFlow controller. This synergy is discussed in [4, Section 8.6] and a proof of concept is proposed to design a system whereby an OpenFlow controller uses the 802.21 protocol messages to control roaming between 802.11 access points. The authors suggest that extensions to OpenFlow may be necessary, however.

Infrastructure-Controlled Roaming in 802.11 Networks

In [10] the authors present Odin, an SDN-based framework to facilitate AAA, policy, mobility, and interference management in 802.11 networks. The fact that in 802.11 the client makes the decision regarding with which AP to associate at any given time has presented a major challenge to giving the network explicit control over with which AP a client associates. Controlling the choice and timing of the client-AP association in 802.11 is fundamental to roaming, RF interference, and load-balancing solutions. The Odin master function, which is an SDN application, controls Odin agents in each of the APs. These agents implement light virtual access points (LVAPs) that appear as physical APs to the client. Each client is assigned a unique LVAP, giving each client a unique BSSID. Since LVAPs may be instantiated on any of the APs under the control of the controller, the client may physically roam to another physical AP by the controller moving its LVAP instantiation to that new physical AP. OpenFlow is used to move the user data flows in accordance with such roaming.

13.2.5 Traffic Engineering in Mobile Networks

Dynamic Assignment of Flows to Fluctuating Backhaul Links

In Section 8.2.2 we presented an example of Open SDN being used for traffic engineering in MPLS networks. In mobile networks, there is a novel opportunity to apply traffic engineering to wireless backhaul infrastructure links. Unlike the backhaul example in Section 13.2.1, the wireless links we refer to here connect switching elements that may carry the traffic of multiple base stations (eNBs). The appeal of wireless backhaul links is growing as the number of base stations grows, the locations of the switching elements becomes more dynamic, and new backhaul capacity is brought online to a more freely chosen set of locations. A downside of wireless backhaul is that the bandwidth of the wireless backhaul is both more limited and, more important, less stable than in its wired counterparts. Current resource management practices are static and do not redirect load dynamically based on short-term fluctuations in wireless capacity.

In [4, Section 8.2] the authors propose that an OpenFlow controller be enabled to be aware of the current available bandwidth on the set of wireless links it is managing. It may be managing a hybrid set of wired and wireless backhaul links. If OpenFlow is responsible for assigning user flows to that set of backhaul links, that assignment can be made as a function of the SLAs specific to each flow. High SLA (higher guarantees) traffic can be steered over wired links, and low SLA traffic can be steered over a wireless link. If one wireless link is experiencing temporary spectrum perturbation, OpenFlow can shunt the traffic over a currently stable wireless link. Note that this proposal requires a mechanism by which the SDN application on the OpenFlow controller be made aware of changes in the wireless bandwidth on each of the wireless links it is managing.

Sharing Wireless Backhaul Across Multiple Operators

The IEEE has chartered a group to study how to enable an Open Mobile Network Interface for omni-Range Access Networks (OmniRAN). The omni-Range aspect implies a unified interface to the multiple radio access network types in the 802 family. These include 802.11 and 802.16, among others. The business model behind this idea is that multiple operators offering a range of RAN technologies in a well-defined geographic area could benefit from a common backhaul infrastructure shared by all the operators for all the different radio technologies. The cost savings of this approach compared to separate backhaul networks for each (operator, RAN) pair are significant. In [4, Section 8.16] the authors argue that since both OpenFlow and OmniRAN are media-independent, there is a natural synergy in applying OpenFlow as the protocol for controlling and configuring the various IEEE 802 nodes in this architecture as well as using OpenFlow for its native benefits of fine-grained control of network flows and implementation network policy. This effort would require extension to OpenFlow for controlling and configuring the IEEE 802 nodes.

An OpenFlow Switch on Every Smartphone!

Today’s smartphones generally have multiple radios. For example, it is common to see LTE, WiFi, and 3G radios on the same mobile phone. In the existing model, the MN chooses which radio to use based on a static algorithm. This algorithm normally routes voice traffic over the best available cellular connection and data over a WiFi connection if it is available. If no WiFi connection is available, data is sent via the cellular connection. This is a fairly inefficient and brute-force use of the available radio spectrum. For example, if a functioning WiFi connection is poor, it may be much more efficient to route data over a functioning LTE connection. If multiple radios are available, it may be wise to use more than one radio simultaneously for different data flows. Expanding the horizon for OpenFlow applications to a bold new level, in [4, Section 8.8] there is a proposal to install an instance of *Open vSwitch* (OVS) on every mobile device. This virtual switch would direct flows over the radio most appropriate for the type and volume of traffic as well as the current spectrum availability for the different radio types. The controller for this OVS instance resides at a gateway in the 3GPP architecture and uses its global knowledge of network state to steer flows in the mobile phone over the most appropriate radio according to the bandwidth and current loading of that particular radio spectrum in that time/location.

13.2.6 Energy Savings

When asked about their vision for the future role of SDN, members of an expert panel [3] from HP, Dell, and NEC indicated that energy savings are an important area where SDN can play an increasing role. Data centers incur enormous OPEX costs in keeping their massive data warehouses cooled and fully and redundantly powered. Companies that produce compute and storage servers now tout their energy savings relative to their equivalents of only a few years ago. There have been significant improvements in the energy efficiency of servers, but the corresponding gains in networking equipment have been smaller.

Between the cooling and the energy consumed by the compute and storage servers, it is estimated [11] that about 70% of the power is allocated to those resources. Beyond that, however, it is estimated that approximately 10% to 20% of the total power is spent to power networking gear. Considering the vast sums spent on electric bills in data centers, making a dent in this 10–20% would represent significant savings.

ElasticTree

One approach to applying OpenFlow to energy savings in the data center, called *ElasticTree*, is described in [11]. The premise of that work is that data centers' networking infrastructure of links and switches is designed to handle an anticipated peak load and is consuming more power than necessary during normal operation. If a means could be found to only power the minimum necessary subset of switches at any moment, there is an opportunity for significant energy savings. The assertion in [11] is that during periods of less than peak load, OpenFlow can be used to shunt traffic around switches that are candidates for being powered off. Such a system assumes an out-of-band mechanism whereby switches may be powered on and off via the OpenFlow application. Such systems exist and are readily integrated with an OpenFlow application. The authors suggest that by varying the number of powered-on switches, their system can provide the ability to fine-tune between energy efficiency, performance, and fault tolerance. The novel work in [11] is related to determining what subset of network switches and links need to be powered on to meet an established set of performance and fault-tolerance criteria.

Dynamic Adjustment of Wireless Transmit Power Levels

A related use case is proposed for wireless backhaul for mobile networks in [4, Section 8.8]. In this case, the mobile operator has the ability to vary the amount of power consumed by the wireless links themselves by varying the transmission power levels. For example, relatively lower traffic loads over a microwave link may require less bandwidth, which may be achieved with a lower transmission power level. If no traffic is flowing over a wireless link, the transmission power may be turned off entirely. As with ElasticTree, the proposal is to use OpenFlow to selectively direct traffic flows over the wireless links such that transmission power levels may be set to globally optimal settings.

More Energy Efficient Switching Hardware

A more direct approach to energy savings in an SDN environment is to directly design more energy-efficient switches. Well-known methods of reducing power consumption used on servers are already being applied to switch design. At some point, use of electronic circuitry consumes energy, though. One of the most power-hungry components of a modern switch capable of flow-based policy enforcement is the TCAM. In [12] the authors propose prediction circuitry that allows flow identification of a high percentage of packets in order to avoid the power-hungry TCAM lookup. The prediction logic uses a memory cache that exploits *temporal locality* of packets. This temporal locality refers to the fact that numerous packets related to the same application flow often arrive at a switch ingress port close to one another. Each TCAM lookup that can be circumvented lowers the switch's cumulative power consumption. A different approach to such TCAM-based energy savings is proposed in [13], where the authors suggest organizing the TCAM into blocks such that the search space is segmented in accordance with those blocks. When a particular lookup is initiated by the switch, it is possible to identify that subset of internal blocks that may be relevant to the current search and only power on that subset of the TCAM during that search.

13.2.7 SDN-Enabled Switching Chips

In Section 12.9.3 we described two ongoing commercial efforts to build chips that are designed from the ground up to support advanced OpenFlow capability. We asserted in Section 13.1 that the *plateau of productivity* would likely remain elusive until such chips become commercially available. In addition to the commercial efforts mentioned above, this is also an active area of research. For example, in [20] the authors show a model of a 256-core programmable network processing unit. This chip is highly programmable and would support the nature and size of flow tables that will be required to exploit the features of OpenFlow 1.3. The main argument of [20] is that such a programmable network processor can be produced and still have the high-throughput characteristics more commonly associated with purpose-built, nonprogrammable Ethernet chips. It is reasonable to believe that such a 256-core chip will be commercially available in the not-too-distant future. In [19] the authors describe an existing progammable networking processor with 256 cores. Though this is a general-purpose processing unit, it provides evidence of the scale that multicore SDN-specific ASICs are likely to achieve in the near future.

In Section 13.2.6 we described a modified ASIC design that would be more energy efficient than existing switching ASICs in performing the flow table lookups required in SDN flow processing. Another advancement is proposed in [21], where the combination of an SDN-enabled ASIC and local general-purpose CPU and memory could be programmed to handle current and yet unanticipated uses of per-flow counters. This is an important issue, since maintaining and using counters on a per-flow basis is a key part of SDN programming, yet the inability to anticipate all future applications of these counters makes it difficult to build them all into ASIC designs. In [21] the authors describe a hybrid approach whereby the hardware counters on the ASIC are programmable and may be assigned to different purposes and to different flows. The counter information combines the matching rule number with the byte count of the matched packet. This information is periodically uploaded to the local CPU, where the counters are mapped to general-purpose OpenFlow counters. Such a system can enable some of the high-level network programming we describe in Section 13.2.2 whereby complex software triggers based on a counter value being reached can be programmed directly into the general-purpose CPU.

13.3 Conclusion

Ironically, one advantage of SDN washing is that it leads to an easy conclusion that SDN is on a path to a permanent and important part of networking technology for decades to come. This is true simply because the term SDN can be stretched to apply to *any* networking technology that involves a modicum of software control. Virtually all networking technologies involve software to some degree; hence, they are all SDN. This flippant analysis is actually not far removed from what appears in some NEM marketing materials. Thus, interpreted that way, SDN's solid hold on the future of networking is ascertained.

Taking a step in a more serious direction, SDN via overlays has had such success in the data center that we conclude that this technology, in one form or another, is here for the long run. SDN via opening up the device is so nascent, with so little track record, that any prediction about its future is pure speculation. We feel that SDN via APIs is really a stopgap approach designed to prolong the life of existing equipment and, thus, will not be long-lived. To the extent that the APIs evolve and permit a remote and centralized controller to directly program the data plane of the switch, then SDN via APIs begins to approach Open SDN. It is significant that such API evolution is not merely a matter of developing new software interfaces. Direct programming of the data plane depends on an evolution of the data plane itself. The API evolution, then, is irrevocably caught up in the much longer delays customary with ASIC design cycles.

The founding precepts of the SDN movement are embodied in Open SDN. As we have admitted previously, we, the authors, are indeed Open SDN evangelists, and we want to conclude this work with a serious look at its future. We begin our answer by retreating to the analogy of Open SDN compared to the relationship of Linux with the entrenched world of the PC industry, dominated by the incumbent Microsoft and, to a much smaller degree, Apple. Linux has actually enjoyed tremendous success, but it has not come close to displacing the incumbents from their predominant market. Linux's most striking success has come instead in unexpected areas. The incumbents continue to dominate the PC operating system market, notwithstanding the few zealots who prefer the freedom and openness of Linux. (*How many people do you know who run Linux on their laptops?*) Linux has enjoyed great success, however, in the server and embedded OS markets. The amazing success of the Android operating system is, after all, based on Linux. Thus, Linux's greatest success has come in areas related to but orthogonal to its original area of application.

In our opinion, the most likely scenario for Open SDN is to follow that Linux trajectory. For a variety of reasons, it will not soon displace the incumbent NEMs in traditional markets, as was posited in 2009 through 2012. It will maintain and expand its beachhead in spaces where its technical superiority simply trumps all alternatives. Whether that beachhead remains restricted to those very large and very sophisticated customers for whom technical superiority of the Open SDN paradigm trumps the safety of the warm embrace of the established NEMs or it expands to a larger market because some NEMs manage to adopt Open SDN without sacrificing profits, it is too early to tell. There will be applications and use cases such as those cited in Section 13.2 where Open SDN solves problems for which no good solution exists today. In those areas, Open SDN will surely gain traction, just as Linux has tranformed the world of operating systems.

A Chinese proverb states, "May you live in interesting times." For better or worse, due to data centers, cloud, mobility, and the desire to simplify and reduce costs, networking today is in the midst of such "interesting times." How it all plays out in the next few years is up for debate—but whatever the outcome, *Software Defined Networking* is certain to be in the middle of it.

References

[1] Research methodology, Gartner. Retrieved from <www.gartner.com/it/products/research/methodologies/research_hype.jsp>.

[2] Kerner S. OpenFlow inventor martin Casado on SDN, VMware, and software-defined networking hype [VIDEO]. Enterprise networking planet; April 29, 2013. Retrieved from <www.enterprisenetworkingplanet.com/netsp/openflow-inventor-martin-casado-sdn-vmware-software-defined-networking-video.html>.

[3] Pronschinske M. Debating the future of SDN [VIDEO]. NetEvents; September 8, 2013. Retrieved from <server.dzone.com/articles/debating-future-sdn>.

[4] Wireless and mobile working group Charter application: use cases. Open Networking Foundation; October 2013.

[5] Wang G, Ng T, Shaikh A. Programming your network at run-time for big data applications. HotSDN, Helsinki, Finland; August 2012. Retrieved from <conferences.sigcomm.org/sigcomm/2012/paper/hotsdn/p103.pdf>.

[6] Handigol N, Heller B, Jeyakumar V, Mazieres D, McKeown N. Where is the debugger for my software-defined network? HotSDN, Helsinki, Finland; August 2012. Retrieved from <conferences.sigcomm.org/sigcomm/2012/paper/hotsdn/p55.pdf>.

[7] Canini M, Venzano D, Peresini P, Kostic D, Rexford J. A NICE way to test OpenFlow applications. In: Ninth USENIX symposium on networked system design and implementation, San Jose, CA, USA; April 2012. Retrieved from <www.usenix.org/system/files/conference/nsdi12/nsdi12-final105.pdf>.

[8] Khurshid A, Zhou W, Caesar M, Brighten Godfrey P. VeriFlow: verifying network-wide invariants in real time. HotSDN, Helsinki, Finland; August 2012. Retrieved from <conferences.sigcomm.org/sigcomm/2012/paper/hotsdn/p49.pdf>.

[9] Jafarian JH, Al-Shaer E, Duan Q. OpenFlow Random host mutation: transparent moving target defense using software-defined networking. HotSDN, Helsinki, Finland; August 2012. Retrieved from <conferences.sigcomm.org/sigcomm/2012/paper/hotsdn/p127.pdf>.

[10] Suresh L, Schulz-Zander J, Merz R, Feldmann A, Vazao T. Towards programmable enterprise WLANs with Odin. HotSDN, Helsinki, Finland; August 2012. Retrieved from <conferences.sigcomm.org/sigcomm/2012/paper/hotsdn/p115.pdf>.

[11] Heller B, Seetharaman S, Mahadevan P, Yiakoumis Y, Sharma P, Banerjee S, et al. ElasticTree: saving energy in data center networks. In: Seventh USENIX symposium on networked system design and implementation, San Jose, CA, USA; April, 2010. Retrieved from <www.usenix.org/legacy/event/nsdi10/tech/full_papers/heller.pdf>.

[12] Congdon PT, Mohapatra P, Farrens M, Akella V. Simultaneously reducing latency and power consumption in OpenFlow switches. July 2013. IEEE/ACM Trans Network PP(99).

[13] Ma Y, Banerjee S. A smart pre-classifier to reduce power consumption of TCAMs for multidimensional packet classification. SIGCOMM 2012, Helsinki, Finland; August 2012. Retrieved from <conferences.sigcomm.org/sigcomm/2012/paper/sigcomm/p335.pdf>.

[14] Hot topics in software-defined networking (HotSDN). SIGCOMM 2012, Helsinki, Finland; August 2012. Retrieved from <conferences.sigcomm.org/sigcomm/2012/hotsdn.php>.

[15] ACM sigcomm workshop on hot topics in software-defined networking (HotSDN). SIGCOMM 2013, Hong Kong; August 2013. Retrieved from <conferences.sigcomm.org/sigcomm/2013/hotsdn.php>.

[16] Mendonca M, Astuto B, Nunes A, Nguyen X, Obraczka K, Turletti T. A survey of software-defined networking: past, present and future of programmable networks. To appear in IEEE Communications Surveys & Tutorials (2014). Available at http://hal.inria.fr/hal-00825087.

[17] The internet that wasn't. IEEE Spectrum. 2013;50(8):38–43.

[18] Shenker S. The future of networking, and the past of protocols. In: Open networking summit. Palo Alto, CA, USA: Stanford University; October 2011.
[19] Dupont de Dinechin B. KALRAY: high-performance embedded computing on the MPPA single-chip manycore processor. CERN seminar; October 2013. Retrieved from <indico.cern.ch/getFile.py/access?resId=0&materialId=slides&confId=272037>.
[20] Pongrácz G, Molnár L, Kis ZL, Turányi Z. Cheap silicon: a myth or reality? Picking the right data plane hardware for software-defined networking. HotSDN, SIGCOMM 2013, Hong Kong; August 2013. Retrieved from <conferences.sigcomm.org/sigcomm/2013/papers/hotsdn/p103.pdf>.
[21] Mogul J, Congdon P. Hey. You darned counters! get off my ASIC! HotSDN, SIGCOMM 2012, Helsinki, Finland; August 2012. Retrieved from <conferences.sigcomm.org/sigcomm/2012/paper/hotsdn/p25.pdf>.

APPENDIX

Acronyms and Abbreviations

A

The acronyms and abbreviations are sorted alphabetically based on the acronym or abbreviation, not their expansions. In the event that the acronym describes a particular company's product or is only meaningful in the context of another entity, we cite that company or entity in parentheses.

AAA	Authentication, Authorization, and Accounting
AC	Access Controller
ACL	Access Control List
ACS	Auto-Configuration Server
AP	(Wireless) Access Point
API	Application Programmer Interface
AR	Access Router
ARP	Address Resolution Protocol
ARPANET	Advanced Research Projects Agency Network
ASF	Apache Software Foundation
ASIC	Application-Specific Integrated Circuit
BSD	Berkeley Software Distribution
B-VID	Backbone VLAN ID
BYOD	Bring Your Own Device
CAM	Content-Addressable Memory
CAPEX	Capital Expense
BOM	Bill of Materials
CE	Control Element
CE	Customer Edge
CLI	Command-Line Interface
COPS	Common Open Policy Service
COS	Class of Service
COTS	Common Off-the-Shelf
CPE	Customer Premises Equipment
CPU	Central Processing Unit
CSMA/CD	Carrier-Sense Multiple-Access/Collision Detect
C-VID	Customer VLAN ID
DCAN	Devolved Control of ATM Networks
DDoS	Distributed Denial of Service
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name System

DoS	Denial of Service
DOVE	Distributed Overlay Virtual Ethernet (IBM)
DPDK	Data Plane Development Kit (Intel)
DPI	Deep Packet Inspection
DSCP	Differentiated Services Code Point
DSL	Digital Subscriber Loop (aka Digital Subscriber Line)
DVR	Digital Video Recorder
ECMP	Equal-Cost Multipath
EPL	Eclipse Public License
ETSI	European Telecommunications Standards Institute
EVB	Edge Virtual Bridging
FE	Forwarding Element
FDDI	Fiber Distributed Data Interface
ForCES	Forwarding and Control Element Separation
FOSS	Free and Open Source Software
FSF	Free Software Foundation
Gbps	Gigabits Per Second
GPL	General Public License
GSMP	General Switch Management Protocol
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure
IaaS	Infrastructure as a Service
IDS	Intrusion Detection System
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
InCNTRE	Indiana Center for Network Translational Research and Education
IP	Internet Protocol
IPS	Intrusion Prevention System
IPv4	Internet Protocol Version 4
IPv6	Internet Protocol Version 6
ISG	Industry Specification Group (ETSI)
I-SID	Service Instance VLAN ID
IS-IS	Intermediate System to Intermediate System
ISP	Internet Service Provider
IT	Information Technology
ITU	International Telecommunications Union
KVM	Kernel-Based Virtual Machine
LAG	Link Aggregation
LAN	Local Area Network
LIFO	Last-In/First-Out
LSP	Label Switched Path
MAC	Media Access Control
Mbps	Megabits Per Second

MIB	Management Information Base
MPLS	Multiprotocol Label Switching
MPLS-TE	MPLS Traffic Engineering
MSTP	Multiple Spanning Tree Protocol
MTU	Maximum Transmission Unit
NAC	Network Access Control
NAT	Network Address Translation
NE	Network Element
NEM	Network Equipment Manufacturer
NETCONF	Network Configuration Protocol
NIC	Network Interface Card
NFV	Network Functions Virtualization
nvo3	Network Virtualization Overlays
NVP	Network Virtualization Platform (Nicira)
NVGRE	Network Virtualization using Generic Routing Encapsulation
ODM	Original Device Manufacturer
OEM	Original Equipment Manufacturer
ONE	Open Networking Environment
onePK	Open Networking Environment Platform Kit
ONF	Open Networking Foundation
ONRC	Open Networking Research Center
OOOD	Out-of-Order Delivery
OPEX	Operational Expense
OTN	Optical Transport Network
OSI	Open Systems Interconnection
OVS	Open vSwitch
OVSDB	Open vSwitch Database Management Protocol
PAN	Personal Area Network
PCE	Path Computation Element
PE	Provider Edge
PE-VLAN	Provider Edge VLAN
PBR	Policy-Based Routing
PHY	Physical Layer and Physical Layer Technology
PaaS	Platform as a Service
PKI	Public Key Infrastructure
PoA	Point of Access
PoC	Proof of Concept
POJO	Plain Old Java Object
PoS	Point of Service
QoS	Quality of Service
RAN	Radio Access Network
RFC	Request for Comments
REST	Representational State Transfer

RISC	Reduced Instruction Set Computing
ROI	Return on Investment
RSTP	Rapid Spanning Tree Protocol
RSVP	Resource Reservation Protocol
SaaS	Software as a Service
SAL	Service Abstraction Layer
SDH	Synchronous Digital Hierarchy
SDM	Software Defined Mobility
SDN	Software Defined Networking
SDN VE	Software Defined Network for Virtual Environments
SIP	Session Initiation Protocol
SLA	Service-Level Agreement
SNMP	Simple Network Management Protocol
SONET	Synchronous Optical Networking
SPAN	Switch Port Analyzer
SPB	Shortest-Path Bridging
STP	Spanning Tree Protocol
STT	Stateless Transport Tunneling
S-VID	Service Provider VLAN ID
TAM	Total Available Market
TCAM	Ternary Content-Addressable Memory
TLV	Type-Length-Value
TL1	Transaction Language 1
ToR	Top of Rack
ToS	Type of Service
TRILL	Transparent Interconnection of Lots of Links
TSO	TCP Segmentation Offload
TTL	Time to Live
VAR	Value-Added Reseller
VEPA	Virtual Edge Port Aggregator
VLAN	Virtual Local Area Network
VoIP	Voice over IP
VTEP	Virtual Tunnel Endpoint or VXLAN Tunnel Endpoint
VXLAN	Virtual Extensible Local Area Network
WaaS	WiFi as a Service
Wintel	Refers to the collaboration of Microsoft (through its Windows OS) with Intel (through its processors) in creating a de facto standard for a personal computer platform
WLAN	Wireless Local Area Network
XaaS	Everything as a Service
XML	Extensible Markup Language
XMPP	Extensible Messaging and Presence Protocol
XNC	Extensible Network Controller

APPENDIX

B Blacklist Application

The source code included in this appendix may be downloaded from www.tallac.com/SDN/get-started/.

B.1 MessageListener

```
//=================================================================
//   MessageListener class for receiving openflow messages
//   from Floodlight controller.
//=================================================================

public class MessageListener implements IOFMessageListener
{
    private static final MessageListener INSTANCE =
                                          new MessageListener();

    private static IFloodlightProviderService mProvider;

    private MessageListener() {}  // private constructor

    public static MessageListener getInstance() {
        return INSTANCE;
    }

    //---------------------------------------------------------------
    public void init(final FloodlightModuleContext context)
    {
        if( mProvider != null ) throw new RuntimeException(
                "BlackList Message listener already initialized" );

        mProvider = context.getServiceImpl(
                            IFloodlightProviderService.class );
    }

    //--------------------------------------------------------------
    public void startUp()
    {
        // Register class as MessageListener for PACKET_IN messages.
        mProvider.addOFMessageListener( OFType.PACKET_IN, this );
    }

    //----------------------------------------------------------------
    @Override
    public String getName() { return BlackListModule.NAME; }

    //----------------------------------------------------------------
    @Override
    public boolean isCallbackOrderingPrereq( final OFType type,
                                             final String name )
    {
        return( type.equals( OFType.PACKET_IN ) &&
                ( name.equals("topology") ||
                  name.equals("devicemanager") ) );
    }

    //----------------------------------------------------------------
    @Override
    public boolean isCallbackOrderingPostreq( final OFType type,
                                              final String name )
    {
        return( type.equals( OFType.PACKET_IN ) &&
                name.equals( "forwarding" ) );
    }

    //----------------------------------------------------------------
    @Override
    public Command receive( final IOFSwitch         ofSwitch,
                            final OFMessage         msg,
                            final FloodlightContext context )
    {
        switch( msg.getType() )
        {
        case PACKET_IN:  // Handle incoming packets here

           // Create packethandler object for receiving packet in
           PacketHandler ph = new PacketHandler( ofSwitch,
                                                 msg, context);

           // Invoke processPacket() method of our packet handler
           // and return the value returned to us by processPacket
           return ph.processPacket();

        default: break;  // If not a PACKET_IN, just return
        }

        return Command.CONTINUE;
    }

}
```

B.2 PacketHandler

```
//================================================================
//   PacketHandler class for processing packets receives
//   from Floodlight controller.
//================================================================

public class PacketHandler
{
    public static final short   TYPE_IPv4 = 0x0800;
    public static final short   TYPE_8021Q = (short) 0x8100;

    private final IOFSwitch         mOfSwitch;
    private final OFPacketIn        mPacketIn;
    private final FloodlightContext mContext;
    private       boolean           isDnsPacket;

    //-------------------------------------------------------------
    public PacketHandler( final IOFSwitch         ofSwitch,
                          final OFMessage         msg,
                          final FloodlightContext context )
    {
        mOfSwitch   = ofSwitch;
        mPacketIn   = (OFPacketIn) msg;
        mContext    = context;
        isDnsPacket = false;
    }

    //-------------------------------------------------------------
    public Command processPacket()
    {
        // First, get the OFMatch object from the incoming packet
        final OFMatch ofMatch = new OFMatch();
        ofMatch.loadFromPacket( mPacketIn.getPacketData(),
                                mPacketIn.getInPort()     );

        // If the packet isn't IPv4, ignore.
```

```
if( ofMatch.getDataLayerType() != Ethernet.TYPE_IPv4 )
{
    return Command.CONTINUE;
}

//---- First handle all IP packets -----------------------
// We have an IPv4 packet, so check the
// destination IPv4 address against IPv4 blacklist.
try
{
    // Get the IP address
    InetAddress ipAddr = InetAddress.getByAddress(
              IPv4.toIPv4AddressBytes(
                    ofMatch.getNetworkDestination() ) );

    // Check the IP address against our blacklist
    if( BlacklistMgr.getInstance()
                        .checkIpv4Blacklist( ipAddr ) )
    {
        // It's on the blacklist, so update stats...
        StatisticsMgr.getInstance()
                            .updateIpv4Stats( mOfSwitch,
                                              ofMatch,
                                              ipAddr    );

        // ... and drop the packet so it doesn't
        // go through to the destination.
        FlowMgr.getInstance().dropPacket( mOfSwitch,
                                          mContext,
                                          mPacketIn );

        return Command.STOP; // Done with this packet,
                             // don't let somebody else
                             // change our DROP
    }

}

catch( UnknownHostException e1 )
{
    // If we had an error with something, bad IP or some
    return Command.CONTINUE;
}

//---- Now handle DNS packets ----------------------------

// Is it DNS?
```

```
if( ofMatch.getNetworkProtocol() == IPv4.PROTOCOL_UDP &&
    ofMatch.getTransportDestination()
                          == FlowMgr.DNS_QUERY_DEST_PORT )
{
    // Prepare data structure to hold DNS hostnames
    // we extract from the request
    final byte[] pkt = mPacketIn.getPacketData();
    Collection<String> domainNames;

    isDnsPacket = true;

    // Get the domain names from the DNS request
    try
    {
        domainNames = parseDnsPacket( pkt );
    }
    catch( IOException e )  // Got here if there was an
                            // exception in parsing the
                            // domain names.
    {
        return Command.CONTINUE; // Just return and
                                 // allow other apps
                                 // to handle the
                                 // request.
    }

    // If there were not any domain names,
    // no checking required, just forward the
    // packet and return.
    if( domainNames == null ) {
        forwardPacket();
        return Command.STOP;
    }

    // If there are domain names, process them.
    for( String domainName : domainNames )
    {
        //  If the current domainName is in
        //  the blacklist, drop the packet and return.
        if( BlacklistMgr.getInstance()
                       .checkDnsBlacklist( domainName ) )
        {
            // Update statistics about the dropped packet.
            StatisticsMgr.getInstance()
                            .updateDnsStats( mOfSwitch,
                                             ofMatch,
                                             domainName );
```

```
                // Drop the packet.
                FlowMgr.getInstance().dropPacket( mOfSwitch,
                                                  mContext,
                                                  mPacketIn );

                return Command.STOP; // Note that we are
                                     // dropping the whole
                                     // DNS request, even if
                                     // only one hostname is bad.
            }
        }
    }

    // If we made it here, everything is okay, so call the
    // method to forward the packet and set up flows for
    // the IP destination, if appropriate.
    forwardPacket();
    return Command.STOP;
}

//--------------------------------------------------------------
private void forwardPacket()
{
    // Get the output port for this destination IP address.
    short outputPort = FlowMgr.getInstance()
              .getOutputPort( mOfSwitch, mContext, mPacketIn );

    // If we can't get a valid output port for this
    // destination IP address, we have to drop it.
    if( outputPort == OFPort.OFPP_NONE.getValue() )
               FlowMgr.getInstance().dropPacket( mOfSwitch,
                                                 mContext,
                                                 mPacketIn );

    // Else if we should flood the packet, do so.
    else if( outputPort == OFPort.OFPP_FLOOD.getValue() ) {
               FlowMgr.getInstance().floodPacket( mOfSwitch,
                                                  mContext,
                                                  mPacketIn );
    }

    // Else we have a port to send this packet out on, so do it.
    else
    {
        final List<OFAction> actions = new ArrayList<OFAction>();

        // Add the action for forward the packet out outputPort
```

```
            actions.add( new OFActionOutput( outputPort ) );

            // Note that for DNS requests,
            // we don't need to set flows up on the switch.
            // Otherwise we must set flows so that subsequent
            // packets to this IP dest will get forwarded
            // locally w/o the controller.
            if( !isDnsPacket )
               FlowMgr.getInstance().createDataStreamFlow( mOfSwitch,
                                                           mContext,
                                                           mPacketIn,
                                                           actions );

            // In all cases, we have the switch forward the packet.
            FlowMgr.getInstance().sendPacketOut( mOfSwitch,
                                                 mContext,
                                                 mPacketIn,
                                                 actions    );

        }

    }

    //-------------------------------------------------------------
    private Collection<String> parseDnsPacket(byte[] pkt) throws IOException
    {
        // Code to parse DNS are return
        // a collection of hostnames
        // that were in the DNS request
    }

}
```

B.3 FlowManager

```
//=================================================================
//   FlowManager class for handling interactions with Floodlight
//   regarding setting and unsetting flows
//=================================================================

public class FlowMgr
{
    private static final FlowMgr INSTANCE = new FlowMgr();
    private static IFloodlightProviderService mProvider;
    private static ITopologyService mTopology;

    public static final short PRIORITY_NORMAL      = 10;
    public static final short PRIORITY_IP_PACKETS  = 1000;
    public static final short PRIORITY_DNS_PACKETS = 2000;
    public static final short PRIORITY_IP_FLOWS    = 1500;
    public static final short PRIORITY_ARP_PACKETS = 1500;

    public static final short IP_FLOW_IDLE_TIMEOUT = 15;
    public static final short NO_IDLE_TIMEOUT = 0;
    public static final int BUFFER_ID_NONE = 0xffffffff;

    public static final short DNS_QUERY_DEST_PORT = 53;

    //-------------------------------------------------------------
    private FlowMgr()
    {
        // private constructor - prevent external instantiation
    }

    //-------------------------------------------------------------
    public static FlowMgr getInstance()
    {
        return INSTANCE;
    }

    //-------------------------------------------------------------
    public void init( final FloodlightModuleContext context )
    {
        mProvider = context.getServiceImpl(IFloodlightProviderService.class);
        mTopology = context.getServiceImpl(ITopologyService.class);
    }

    //-------------------------------------------------------------
    public void setDefaultFlows(final IOFSwitch ofSwitch)
    {
        // Note: this method is called whenever a switch is
        //     discovered by Floodlight, in our SwitchListener
        //     class (not included in this appendix).

        // Set the intitial 'static' or 'proactive' flows
        setDnsQueryFlow(ofSwitch);
        setIpFlow(ofSwitch);
        setArpFlow(ofSwitch);
    }
```

```
//--------------------------------------------------------------
public void sendPacketOut( final IOFSwitch         ofSwitch,
                           final FloodlightContext cntx,
                           final OFPacketIn        packetIn,
                           final List<OFAction>    actions)
{
    // Create a packet out from factory.
    final OFPacketOut packetOut = (OFPacketOut) mProvider
                                .getOFMessageFactory()
                                .getMessage(OFType.PACKET_OUT);

    // Set the actions based on what has been passed to us.
    packetOut.setActions(actions);

    // Calculate and set the action length.
    int actionsLength = 0;
    for (final OFAction action : actions)
        { actionsLength += action.getLengthU(); }
    packetOut.setActionsLength((short) actionsLength);

    // Set the length based on what we’ve calculated.
    short poLength = (short) (packetOut.getActionsLength()
                              + OFPacketOut.MINIMUM_LENGTH);

    // Set the buffer and in port based on the packet in.
    packetOut.setBufferId( packetIn.getBufferId() );
    packetOut.setInPort(   packetIn.getInPort() );

    // If the buffer ID is not present, copy and send back
    // the complete packet including payload to the switch.
    if (packetIn.getBufferId() == OFPacketOut.BUFFER_ID_NONE)
    {
        final byte[] packetData = packetIn.getPacketData();
        poLength += packetData.length;
        packetOut.setPacketData(packetData);
    }

    // Set the complete length of the packet we are sending.
    packetOut.setLength( poLength );

    // Now we actually send out the packet.
    try
    {
        ofSwitch.write( packetOut, cntx );
        ofSwitch.flush();
    }
    catch (final IOException e)
```

```
        {
            // Handle errors in sending packet out.
        }
    }

    //-------------------------------------------------------------
    public void dropPacket( final IOFSwitch         ofSwitch,
                            final FloodlightContext cntx,
                                  OFPacketIn        packetIn)
    {
        LOG.debug("Drop packet");

        final List<OFAction> flActions = new ArrayList<OFAction>();
        sendPacketOut( ofSwitch, cntx, packetIn, flActions );
    }

    //-------------------------------------------------------------
    public void createDataStreamFlow(
                              final IOFSwitch         ofSwitch,
                              final FloodlightContext context,
                              final OFPacketIn        packetIn,
                                    List<OFAction>    actions)
    {
        final OFMatch match = new OFMatch();
        match.loadFromPacket( packetIn.getPacketData(),
                              packetIn.getInPort() );

        // Ignore packet if it is an ARP, or has not source/dest, or is not IPv4.
        if( ( match.getDataLayerType() == Ethernet.TYPE_ARP) ||
            ( match.getNetworkDestination() == 0 )           ||
            ( match.getNetworkSource() == 0 )                ||
            ( match.getDataLayerType() != Ethernet.TYPE_IPv4 ) )
           return;

        // Set up the wildcard object for IP address.
        match.setWildcards( allExclude( OFMatch.OFPFW_NW_DST_MASK,
                                        OFMatch.OFPFW_DL_TYPE ) );

        // Send out the data stream flow mod message
        sendFlowModMessage( ofSwitch,
                            OFFlowMod.OFPFC_ADD,
                            match,
                            actions,
                            PRIORITY_IP_FLOWS,
                            IP_FLOW_IDLE_TIMEOUT,
                            packetIn.getBufferId() );
```

```
}

//-------------------------------------------------------------
private void deleteFlow( final IOFSwitch ofSwitch,
                         final OFMatch   match )
{
    // Remember that an empty action list means 'drop'
    final List<OFAction> actions = new ArrayList<OFAction>();

    // Send out our empty action list.
    sendFlowModMessage( ofSwitch,
                        OFFlowMod.OFPFC_DELETE,
                        match,
                        actions,
                        PRIORITY_IP_FLOWS,
                        IP_FLOW_IDLE_TIMEOUT,
                        BUFFER_ID_NONE );
}

//-------------------------------------------------------------
public void floodPacket(final IOFSwitch ofSwitch,
                        final FloodlightContext context,
                        final OFPacketIn packetIn)
{
    // Create action flood/all
    final List<OFAction> actions = new ArrayList<OFAction>();

    // If the switch supports the 'FLOOD' action...
    if (ofSwitch.hasAttribute( IOFSwitch.PROP_SUPPORTS_OFPP_FLOOD) )
    {
        actions.add(new OFActionOutput(
                               OFPort.OFPP_FLOOD.getValue()));
    }
    // ...otherwise tell it to send it to 'ALL'.
    else
    {
        actions.add( new OFActionOutput(
                               OFPort.OFPP_ALL.getValue() ) );
    }

    // Call our method to send the packet out.
    sendPacketOut( ofSwitch,
                   context,
                   packetIn,
                   actions );
}
```

```
//-------------------------------------------------------------
public short getOutputPort( final IOFSwitch ofSwitch,
                            final FloodlightContext context,
                            final OFPacketIn packetIn )
{
    // return the output port for sending out the packet
    // that has been approved
}

//-------------------------------------------------------------
private static int allExclude(final int... flags)
{
    // Utility routine for assistance in setting wildcard

    int wildcard = OFPFW_ALL;
    for( final int flag : flags ) { wildcard &= ~flag; }

    return wildcard;
}

//-------------------------------------------------------------
private void sendFlowModMessage( final IOFSwitch      ofSwitch,
                                 final short          command,
                                 final OFMatch        ofMatch,
                                 final List<OFAction> actions,
                                 final short          priority,
                                 final short          idleTimeout,
                                 final int            bufferId )
{
    // Get a flow modification message from factory.
    final OFFlowMod ofm = (OFFlowMod) mProvider
                                 .getOFMessageFactory()
                                 .getMessage(OFType.FLOW_MOD);

    // Set our new flow mod object with the values that have
    // been passed to us.
    ofm.setCommand( command ).setIdleTimeout( idleTimeout )
                             .setPriority( priority )
                             .setMatch( ofMatch.clone() )
                             .setBufferId( bufferId )
                             .setOutPort( OFPort.OFPP_NONE )
                             .setActions( actions )
                             .setXid( ofSwitch
                                .getNextTransactionId() );

    // Calculate the length of the request, and set it.
    int actionsLength = 0;
    for( final OFAction action : actions )
             { actionsLength += action.getLengthU(); }
    ofm.setLengthU(OFFlowMod.MINIMUM_LENGTH + actionsLength);

    // Now send out the flow mod message we have created.
    try
    {
        ofSwitch.write( ofm, null );
        ofSwitch.flush();
    }
    catch (final IOException e)
    {
        // Handle errors with the request
    }
}

//--------------------------------------------------------------
private void setDnsQueryFlow( final IOFSwitch ofSwitch )
{
    // Create match object to only match DNS requests
    OFMatch ofMatch = new OFMatch();
    ofMatch.setWildcards( allExclude( OFPFW_TP_DST,
                                      OFPFW_NW_PROTO,
                                      OFPFW_DL_TYPE ) )
               .setDataLayerType( Ethernet.TYPE_IPv4 )
               .setNetworkProtocol( IPv4.PROTOCOL_UDP )
               .setTransportDestination( DNS_QUERY_DEST_PORT );

    // Create output action to forward to controller.
    OFActionOutput ofAction  = new OFActionOutput(
                             OFPort.OFPP_CONTROLLER.getValue(),
                             (short) 65535                    );

    // Create our action list and add this action to it
    List<OFAction> ofActions = new ArrayList<OFAction>();
    ofActions.add(ofAction);

    sendFlowModMessage( ofSwitch,
                        OFFlowMod.OFPFC_ADD,
                        ofMatch,
                        ofActions,
                        PRIORITY_DNS_PACKETS,
                        NO_IDLE_TIMEOUT,
                        BUFFER_ID_NONE );
}

//--------------------------------------------------------------
```

```
private void setIpFlow( final IOFSwitch ofSwitch )
{
    // Create match object to only match all IPv4 packets
    OFMatch ofMatch = new OFMatch();
    ofMatch.setWildcards( allExclude( OFPFW_DL_TYPE ) )
           .setDataLayerType( Ethernet.TYPE_IPv4 );

    // Create output action to forward to controller.
    OFActionOutput ofAction  = new OFActionOutput(
                        OFPort.OFPP_CONTROLLER.getValue(),
                        (short) 65535                    );

    // Create our action list and add this action to it.
    List<OFAction> ofActions = new ArrayList<OFAction>();
    ofActions.add(ofAction);

    // Send this flow modification message to the switch.
    sendFlowModMessage( ofSwitch,
                        OFFlowMod.OFPFC_ADD,
                        ofMatch,
                        ofActions,
                        PRIORITY_IP_PACKETS,
                        NO_IDLE_TIMEOUT,
                        BUFFER_ID_NONE );
}

//-------------------------------------------------------------
private void setArpFlow( final IOFSwitch ofSwitch )
{
    // Create match object match arp packets
    OFMatch ofMatch = new OFMatch();
    ofMatch.setWildcards( allExclude(OFPFW_DL_TYPE) )
                        .setDataLayerType( Ethernet.TYPE_ARP );

    // Create output action to forward normally
    OFActionOutput ofAction  = new OFActionOutput(
                            OFPort.OFPP_NORMAL.getValue(),
                            (short) 65535                );

    // Create our action list and add this action to it.
    List<OFAction> ofActions = new ArrayList<OFAction>();
    ofActions.add(ofAction);

    // Send this flow modification message to the switch.
    sendFlowModMessage( ofSwitch,
                        OFFlowMod.OFPFC_ADD,
                        ofMatch,
                                      ofActions,
                                      PRIORITY_ARP_PACKETS,
                                      NO_IDLE_TIMEOUT,
                                      BUFFER_ID_NONE );
         }

     }
```

This page is intentionally left blank

Index

This page is intentionally left blank