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

Software Defined Networks. http://dx.doi.org/10.1016/B978-0-12-416675-2.00004-8

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

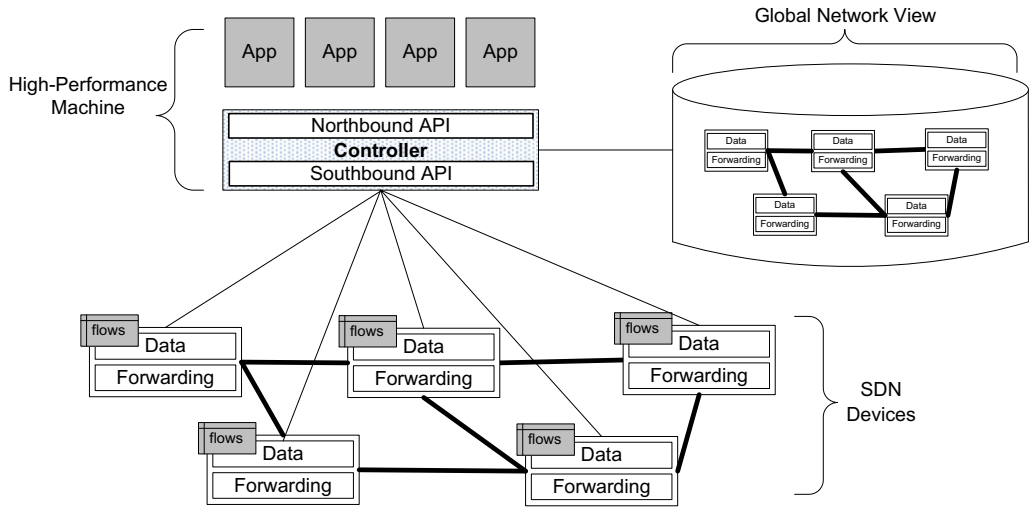

FIGURE 4.1

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

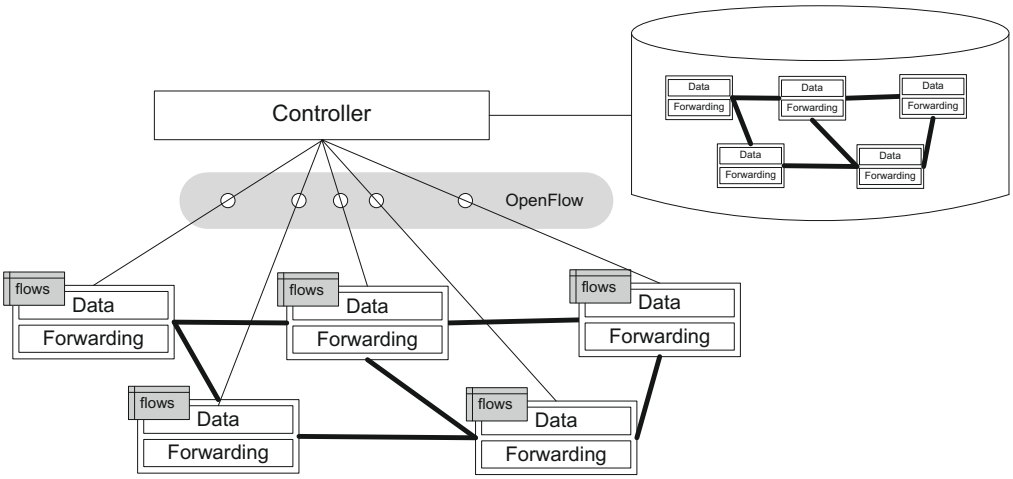

FIGURE 4.2

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.

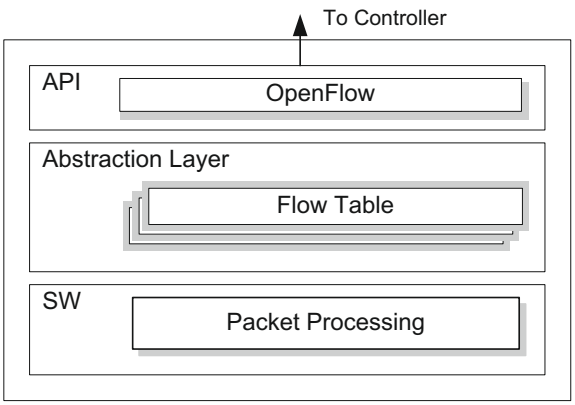

FIGURE 4.3

SDN software switch anatomy.

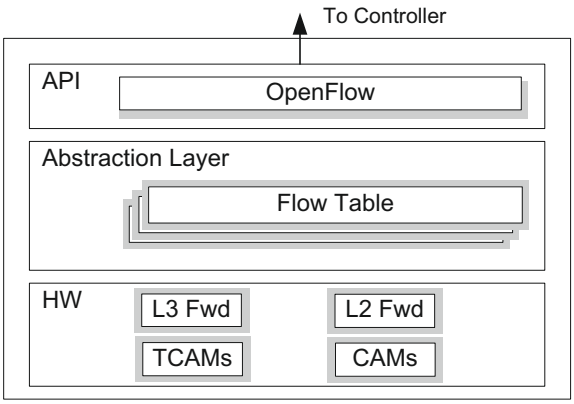

FIGURE 4.4

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.

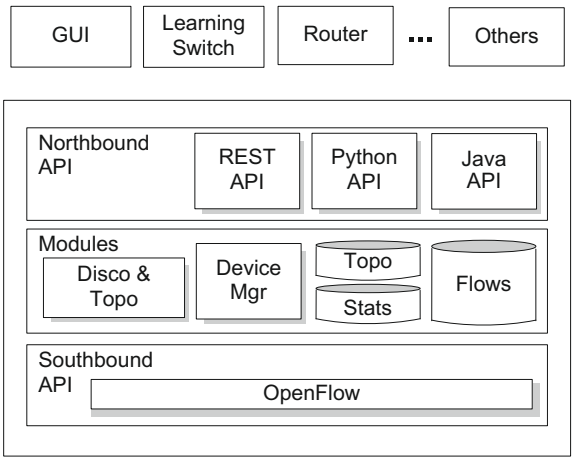

FIGURE 4.5

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.

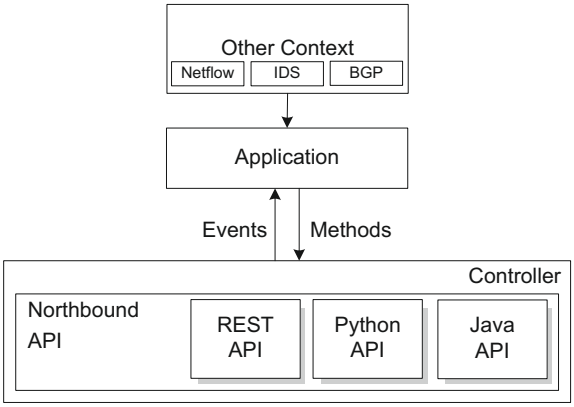

FIGURE 4.6

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.

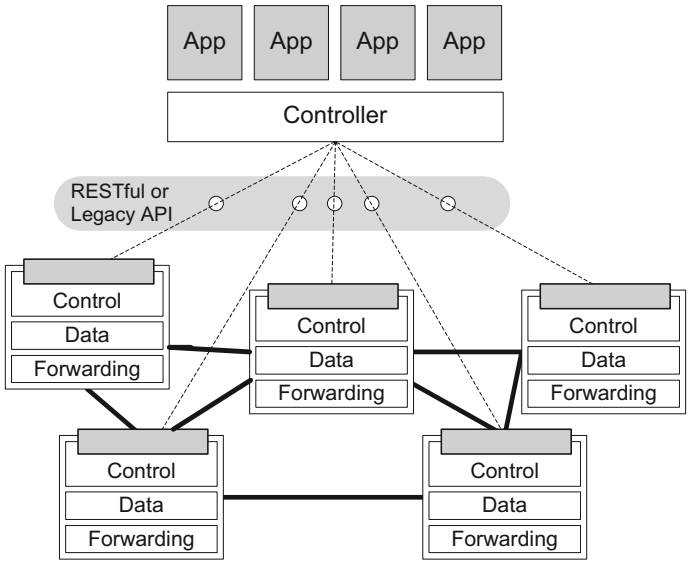

FIGURE 4.7

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

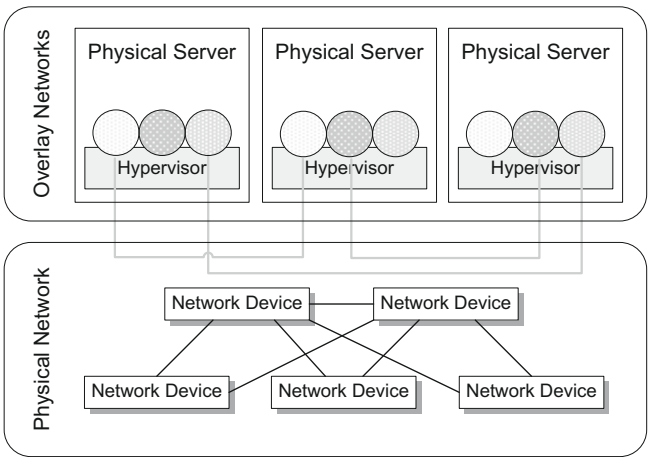

FIGURE 4.8

Virtualized networks.

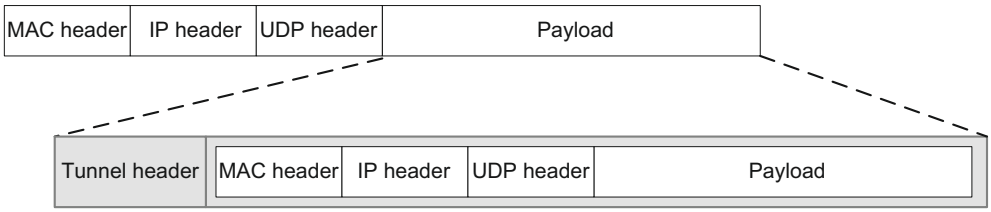

FIGURE 4.9

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

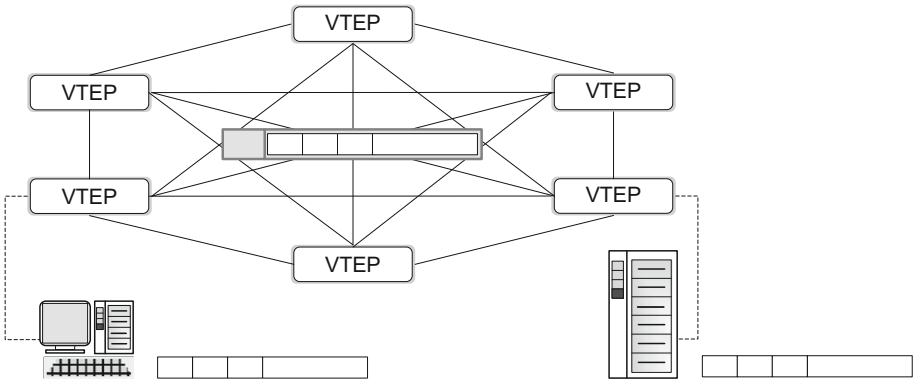

FIGURE 4.10

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