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

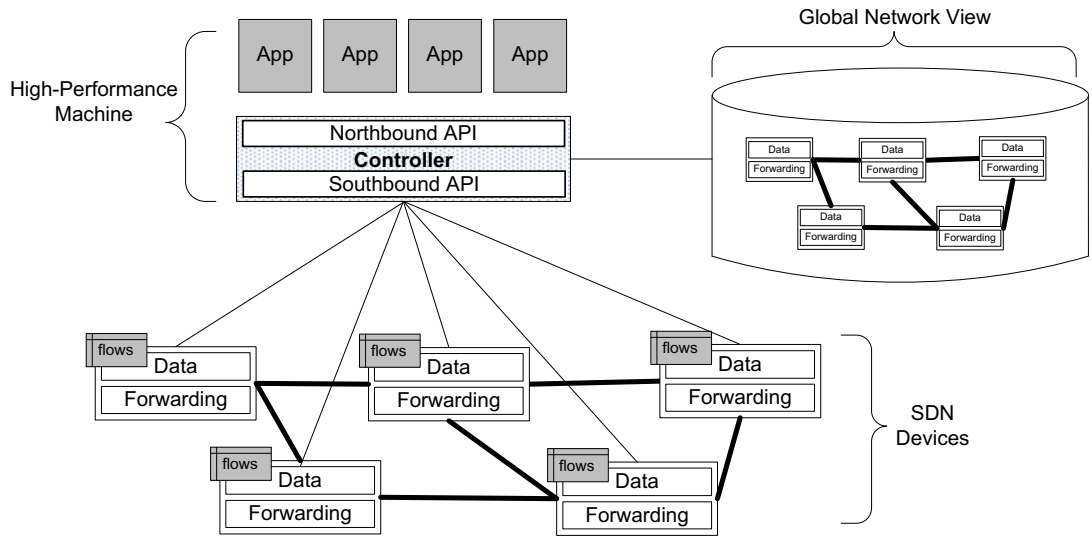

FIGURE 4.1

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

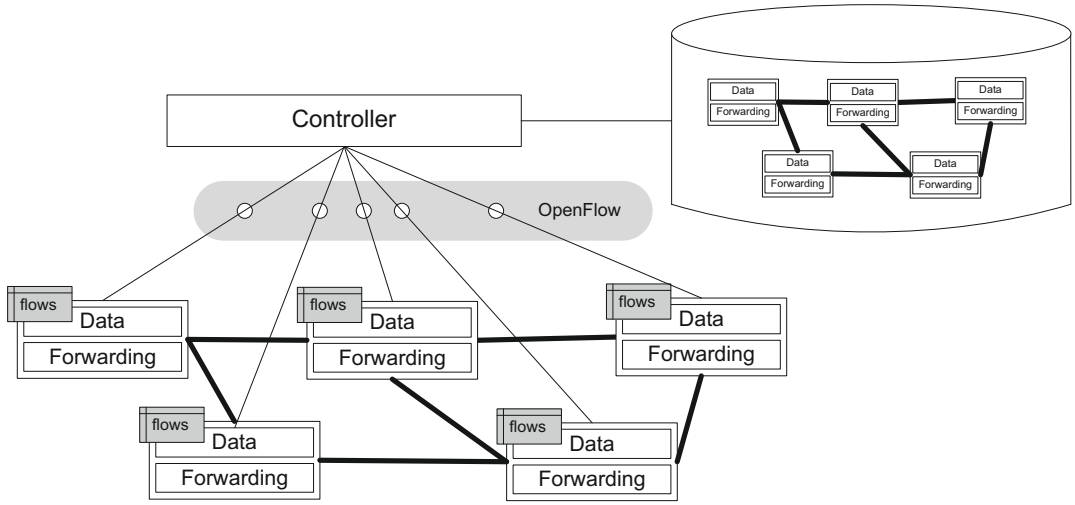

FIGURE 4.2

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.

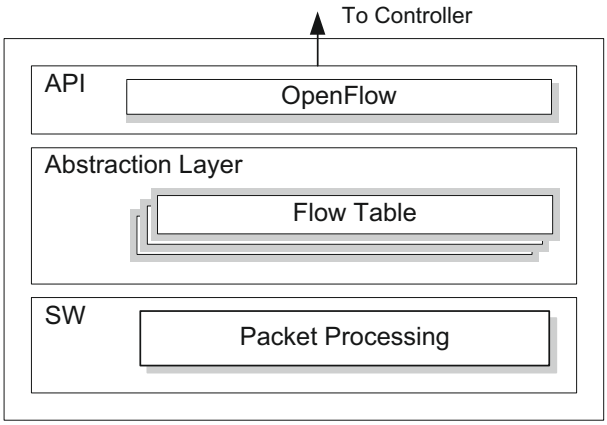

FIGURE 4.3

SDN software switch anatomy.

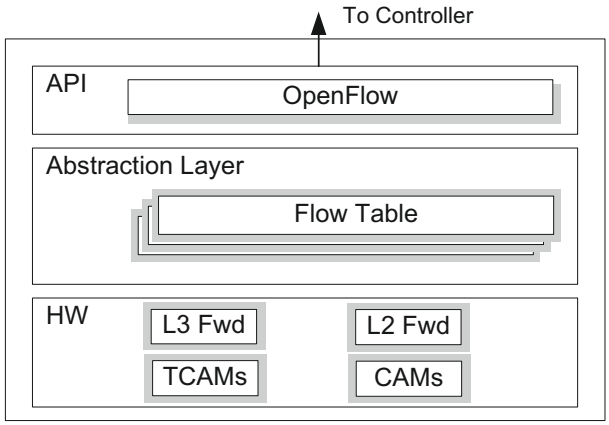

FIGURE 4.4

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,