CHAPTER

ALTERNATIVE DEFINITIONS OF SDN 6

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 and 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 of 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, there may be a tendency by some vendors 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 remains a bit subjective, as our litmus test is not absolute. Some of the alternatives do exhibit some of our defined SDN traits, but not all. In addition, we will 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

While the advent of the idea of Software Defined Networking has been met with enthusiasm by many, 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, which we examine in this chapter.

Software Defined Networks. http://dx.doi.org/10.1016/B978-0-12-804555-8.00006-5

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 then this point is moot as 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 with a sound migration plan.*
- **Too Risky.** A complete change to the networking process is too risky because the technology is new, 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. While 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 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. Fig. 6.1 shows a simple example of such phasing. First, the SDN technology is tested in a lab

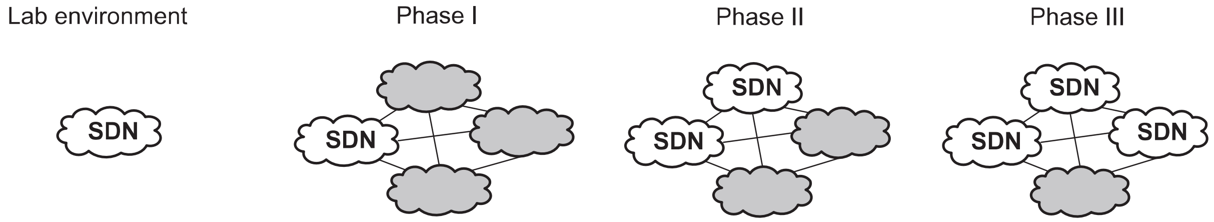

FIG. 6.1

Phased deployment of SDN.

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 re-training of skilled networking personnel. Nevertheless, for those areas 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, then 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.

Fig. 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. Both flows are routed via the same alternate route.

Fig. 6.3 shows a recovery from the failure of a single node in an SDN network. 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 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 Fig. 6.2. Finding optimal paths for different flows is relatively straightforward within

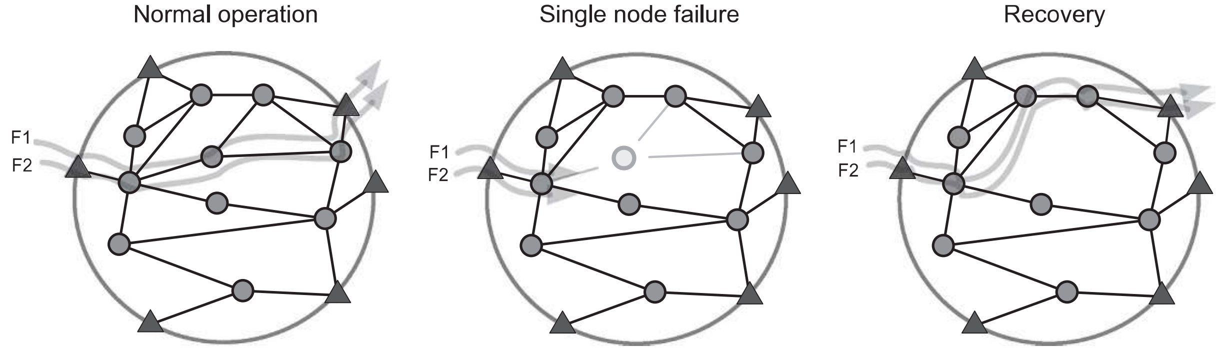

FIG. 6.2

Traditional network failure recovery.

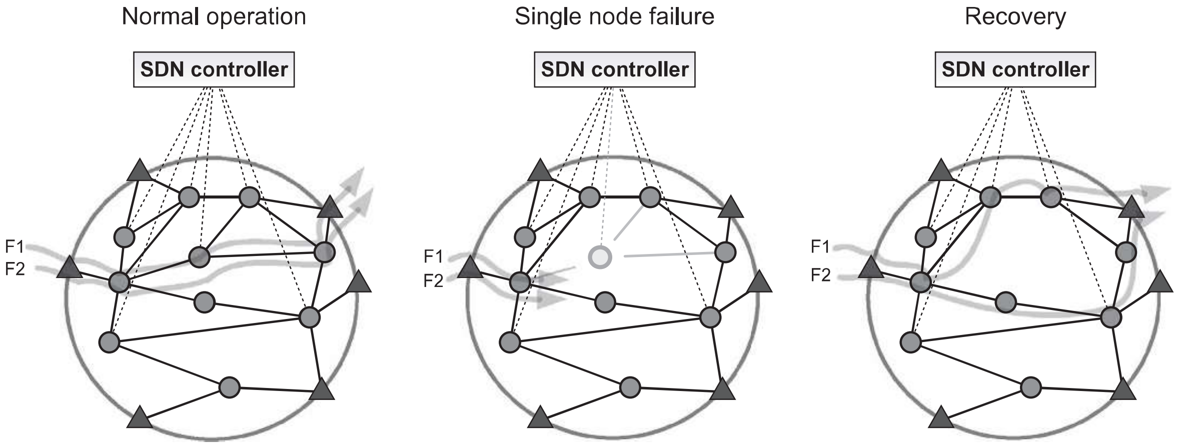

FIG. 6.3

SDN network failure recovery.

the Open SDN model, yet is very challenging to implement in traditional networking with autonomous devices. We revisit this topic via a specific use case in Section 9.1.2.

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

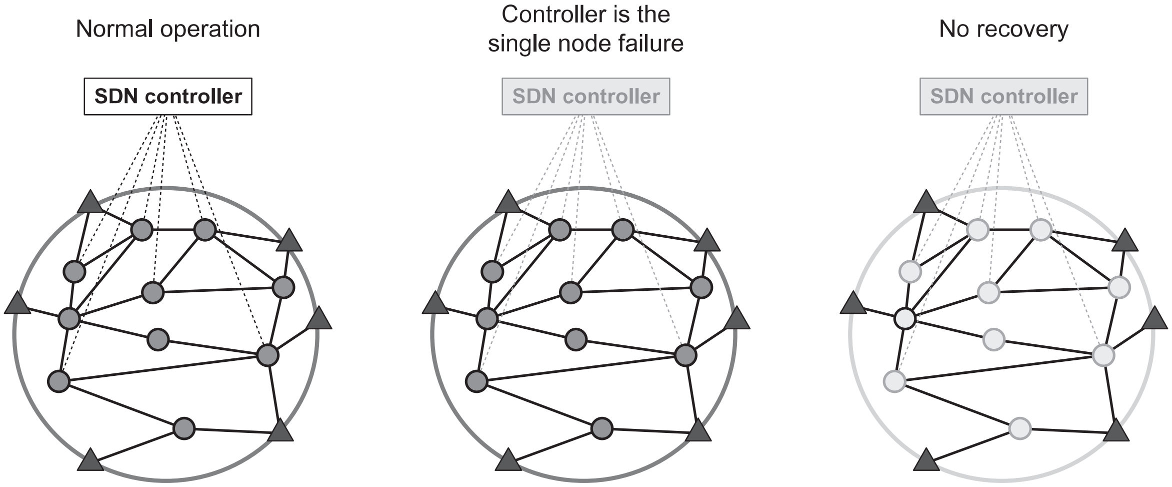

FIG. 6.4

SDN controller as single point of failure.

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!

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, and, 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 which can cause the failure of the controller. A single SDN controller is an attractive target for malicious individuals wishing to attack an organization's networking infrastructure. *Denial of Service* (DoS) attacks, worms, bots, and similar malware can wreak havoc on an SDN controller which 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 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 which 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. Fig. 6.5 shows an example of a well-designed controller configuration with redundant controllers and redundant controller databases. Redundancy techniques ranges from $N + 1$ 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. Fig. 6.5 depicts a monitor which 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.

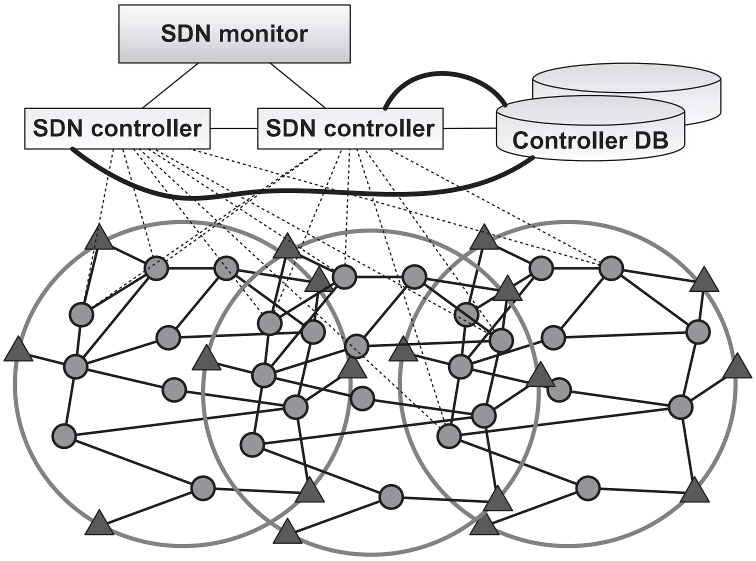

FIG. 6.5

Controller high availability and monitoring.

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 is vulnerable. This is in fact true, and the strongest security measures available 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 naive 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 network there might be ten to twenty SDN controllers. Assuming that the compromising of any control node can bring down the entire network, that means that the SDN network presents two orders of magnitude fewer control nodes susceptible for attack as compared to its traditional equivalent. Through hardening to attack both the controllers and the secure channels, network designers can focus extra protection around these few islands of intelligence in order to keep malicious attacks at bay.

In summary, while 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.

Resilience in SDN is an active research area. In addition to dealing with controller availability, this research crosses a set of disciplines, including survivability, dependability, traffic tolerance, and security. Security encompasses not only protecting against DDoS attacks, but also issues of *confidentiality* and *integrity*. There are also research efforts dealing specifically with the consistency between the network controller and redundant backups, and others investigating placement strategies of the network controller. This last item is known as *the controller placement problem*. The goal of this research is to find optimal places in the topology to place a controller in order to minimize connectivity issues in case a network partition occurs. A comprehensive discussion of this work on resilience in SDN can be found in [1].

6.1.3 PERFORMANCE AND SCALE

As the saying goes, "Many hands make light work." Applied to networking, this 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 which 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 of 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 while 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 [2]. 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. While 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 amongst the group of devices. In many legacy situations 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 above, 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 does not approach 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 under their control. Fig. 6.6 shows three controllers spreading the load of the management of a large number of devices. While the details of how to implement clusters of controllers is currently out of 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 have published guidelines for how multiple wireless controllers for the same network should interact, notably Cisco in [3] and HP in [4].

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 the off-loading of some of the burden from the *leaf* controllers to be handled by the *root* controller. The concept of controller hierarchies for SDN networks is explored in detail in [5]. Fig. 6.7 shows a two-level hierarchy of controllers, with the root controller managing the coordination between the

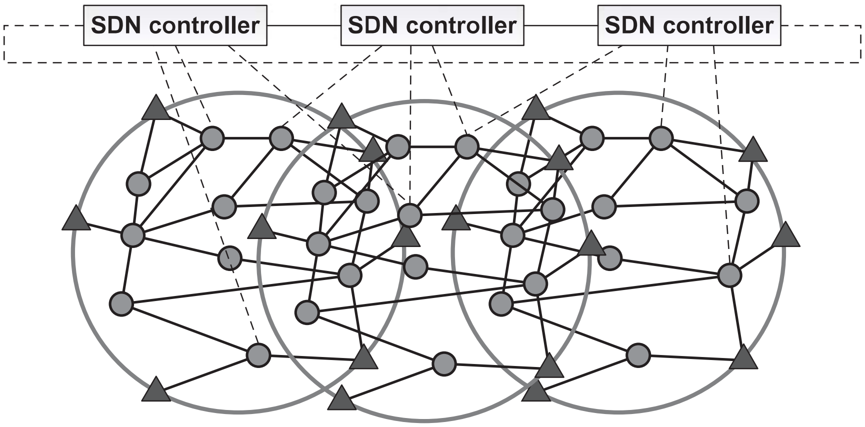

FIG. 6.6

Controller cluster.

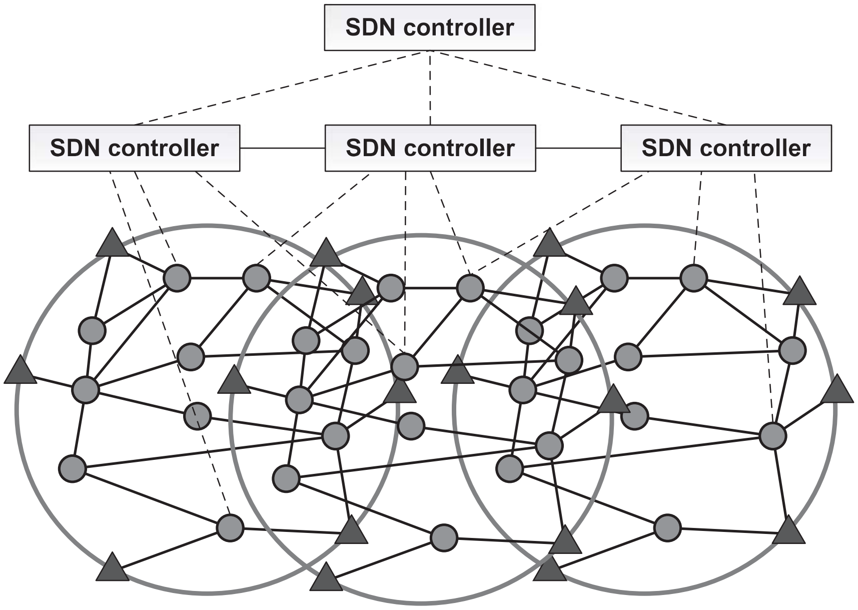

FIG. 6.7

Controller hierarchy.

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.

As mentioned in Section 5.10.4, controller hierarchies are not yet well defined within the OpenFlow family of specifications. This topic has been identified as one urgently needing to be addressed.

To summarize, performance issues can arise in large Open SDN networks, just as they can in non-SDN networks. Most of these issues are due to the logically centralized controller and the possible path latency between the switch and the controller. Engineering solutions are feasible with the scale of today's Open SDN networks but as these scale additional work is needed. There are a number of research efforts focused on performance issues in Open SDN. One of these is *DevoFlow* [6], whose goal it is to reduce the amount of communication between the control and data planes. Another is found in [7], where the authors study the scalability of the SDN controller. In [8], the authors examine compression methods to reduce the amount of redundant rules in order to expedite traffic management. Finally, the performance of the *Northbound API* (NBI) is studied in [9].

6.1.4 DEEP PACKET INSPECTION

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

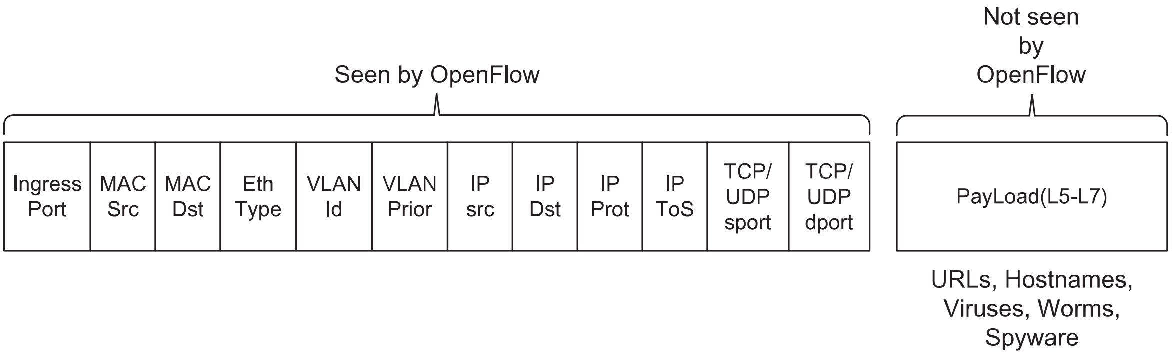

FIG. 6.8

Deep packet inspection.

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 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 which match on appropriate fields, then 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 for 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. Fig. 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. Also, 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.

There are two distinct challenges brought to light in the scenario described above. 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. Secondly, even if the device can determine that a packet needs to be examined for further scrutiny, 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 [10].

In any event, as of this writing, deep packet inspection is still absent from the latest version of the OpenFlow standard. As 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 of this would be encrypting the entire payload of a layer two frame. Obviously, this obscures the layer three headers 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 all of the twelve-tuple of packet header fields normally available as criteria. 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 1.3 device examines every incoming packet independently, without consideration of packets arriving earlier which 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 1.3 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 and UDP port 53, *Dynamic Host Configuration Protocol* (DHCP) uses UDP ports 67 and 68, and so on. These are easy to track and to establish appropriate OpenFlow rules. However, there are certain protocols, for example FTP and *Voice over IP* (VoIP), which start on a standard and fixed port (twenty-one for FTP, 5060 for the VoIP *Session Initiation Protocol* (SIP) protocol) for the initial phase of the conversation, but which 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 which match these dynamic and variable ports. Fig. 6.9 shows an example of stateful awareness where 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 wish 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 in order 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. Fig. 6.10 shows a sequence

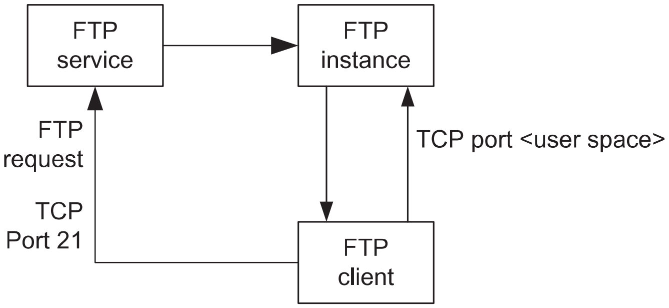

FIG. 6.9

Stateful awareness for special protocols.

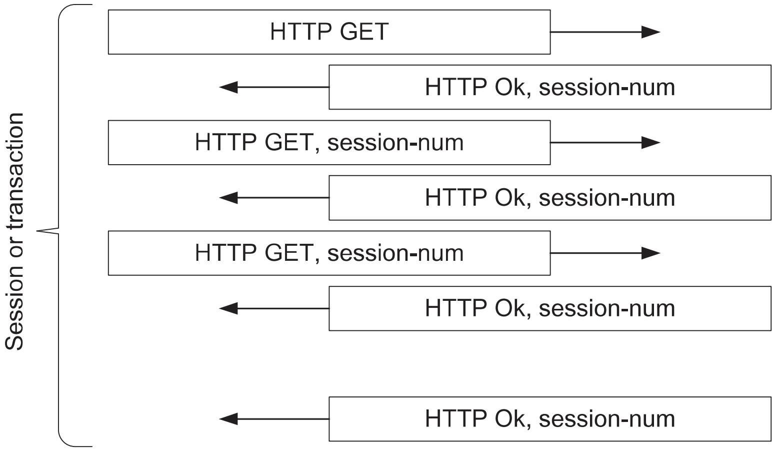

FIG. 6.10

Stateful awareness of application-level transactions.

of HTTP requests which 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 which could replace certain 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.

For this reason, OpenFlow 1.5 contemplates the possibility of maintaining flow state across packets. We provided a brief review of this new feature in Section 5.8.1. As of this writing, there are few details available about implementations based on this feature, so for the time being, in those instances where 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 have reviewed a number of often-cited shortcomings of Open SDN. In some cases, there are documented Open SDN strategies for dealing with the issue. 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 will examine alternative SDN technologies in greater depth than we have previously. For each, we will 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 will also assess how well that alternative meets our criteria to be considered an SDN technology.

DISCUSSION QUESTION:

We have discussed potential drawbacks and limitations of SDN. Were the arguments convincing, or do you feel that these SDN limitations are debilitating for the adoption of SDN? What other drawbacks might there be that we have not mentioned?

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 an attempt by incumbent network vendors to protect their turf against encroachment by radical new technologies. But we would do it a disservice if we did not consider the positive elements of this solution. One such positive element is the movement toward controller-driven networks.

This brings us to the first of two artificial constraints we will 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* [11] provides a rich set of network device programmability and was initially released before SDN became known to the industry. It is not controller-based, as 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 we will exclude them from consideration in this category. Looking back at our introduction to SDN via APIs in Chapter 4, Figs. 4.8 and 4.9 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 wish to delineate the Open SDN 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*. While

[1] For the sake of brevity, we will use the term *SDN via APIs* in the balance of this book.

that 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 now 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, as 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 which allow for a certain amount of configuration. There are a number of network device APIs in current use:

- The *Simple Network Management Protocol* (SNMP) first was standardized in 1988 and it has been the standard for management interactions with networking devices since that time. The general mechanism involved a simple protocol allowing the controlling function to GET and SET objects on a device, which 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. While SNMP does provide both GET and SET primitives, 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 only accessible via CLI (not SNMP). Consequently, management software attempting to access this data must emulate a user logging into the CLI and 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.
- *Remote Authentication Dial In User Service* (RADIUS), which we introduced in Section 3.2.2, has been widely used in residential broadband and has been used to push policy, ACLs and VLANs onto devices.
- The *Transaction Language 1* (TL1) was developed for telecommunications equipment in the 1980s and is similar in some respects to SNMP in that its intent is to provide a language for communication between machines.
- The *TR-069 CPE WAN Management Protocol* [12] is intended to control the communication between *Customer Premises Equipment* (CPE) and an *Auto-Configuration Server* (ACS). In our context, the CPE is the network device. The ACS is a novel contribution of TR-069. The ACS provides secure auto-configuration of the network device, and also incorporates other device management functions into a unified framework.

These APIs have been used for years primarily by network management applications. While we consider these to be legacy APIs, 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 [13]. 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.

Three examples of APIs that seem to scale well as SDN APIs are NETCONF, the BGP-LS/PCE-P plugin, and the *Representational State Transfer* (REST) API. We discuss these in the following sections.

6.2.2 NETCONF/YANG

In Chapter 7 we will delve into the details of the emerging trend of using the *Network Configuration Protocol* (NETCONF) as the southbound API for SDN applications and how network management-style SDN is gaining favor as an evolutionary path toward a more complete SDN solution. In this section we explore the advantages NETCONF provides over SNMP as a protocol for implementing SDN-style behavior in a network. 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. There are a number of strengths that NETCONF exhibits versus SNMP as a protocol for both network management and for SDN:

- **Security**: NETCONF has operated over a secure channel from its beginning, while SNMP had to evolve in order to support rudimentary secure exchanges of data with the management server.
- **Organization**: NETCONF separates *configuration* and *operational* data, making organization and control over different parts of the data much simpler.
- **Announcement of capabilities**: The first exchange between a NETCONF server on the device, and the client on the management station, is the announcement by the device of all the YANG models that it supports. YANG, which we describe in Section 7.2.2, provides a formal language for expressing the device's capabilities. Thus there is no ambiguity regarding what functionality the device supports.
- **Operations**: NETCONF *operations* are *Remote Procedure Calls* (RPCs). This capability allows the SDN controller to instruct the device to take a particular action, passing a set of parameters to the RPC. This enables the controller to more easily manipulate forwarding behavior on the device.

As an SDN API, NETCONF has enjoyed particular popularity amongst large service providers. Service providers have long been lobbying networking vendors to support NETCONF on their large routers. Vendors such as Juniper and Cisco have thus been compelled to implement NETCONF on devices targeted at these customers. This has created a groundswell of support for NETCONF in these environments.

One drawback is that NETCONF suffers from a paucity of standard YANG models. Consider SNMP, which from an early period had MIB definitions for many critical pieces of data on switches and routers. In comparison, the industry still struggles to create standard YANG models for routing, switching, interfaces, ACLs, and other fundamental network components. This hinders SDN application developers, who must write device-specific code in order to control the many different YANG models that exist for the same functionality. We revisit this YANG-related hurdle in Section 7.2.2. Nevertheless, the evidence is that NETCONF will be one of the primary protocols used by future SDN application developers.

6.2.3 BGP-LS/PCE-P

In this section we discuss how the BGP-LS/PCE-P plugin is used to create a more functional and extensive SDN solution, using existing protocols. In Sections 7.2.1, 7.2.4 and 7.2.5 we will delve into greater detail about how this plugin works and its role in an emerging trend in SDN solutions, particularly on the *OpenDaylight* (ODL) controller. At a broad-brush level, we divide the tasks performed by this plugin into two categories, *gathering topology information* and *setting paths and routes*.

Via this plugin, the SDN controller uses the *Border Gateway Protocol* (BGP) to gather exterior routing topology, the *BGP Link State* (BGP-LS) protocol to retrieve interior routing topology, and the *Path Computation Element Protocol* (PCE-P) to acquire topology information related to MPLS. The controller gathers this information to provide various topology views about the network to requesters, thus allowing participating applications to make intelligent decisions about routes and paths.

If so requested by an application, the BGP-LS/PCE-P plugin can program routes in the network by emulating a BGP node and using the BGP protocol to insert routes into the network devices. Similarly, the plugin uses the PCE-P protocol to set up and tear down MPLS *Label Switched Paths* (LSPs) per the application's requests.

We contrast this ODL plugin with other API-based solutions where the API is provided directly on the device or on a controller which then interprets those requests into the appropriate protocol to talk directly to devices. BGP-LS/PCE-P operates at a higher level of abstraction, providing topology, path and routing APIs to applications. This genre of plugin is relatively new but is part of the general momentum toward more abstract APIs, relieving the SDN application of the burden of dealing directly with a large collection of devices.

6.2.4 REST

A new technology being successfully used as an SDN API 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 one of the means for manipulating SDN networks 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 recompilation 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 by 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 some other methods.

A related use of REST is in the RESTCONF protocol. While more often used as a northbound API to the controller, RESTCONF can also be used as a device API. RESTCONF is an implementation of NETCONF/YANG using JSON or XML over HTTP rather than XML RPCs over SSH.

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 also 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.

6.2.5 EXAMPLES OF SDN VIA APIs

Cisco

Cisco originally had its own proprietary SDN program called *Cisco onePK* [14] that consists of a broad set of APIs in the devices. These APIs provide SDN functionality on legacy Cisco switches. Cisco onePK, currently de-emphasized by Cisco, 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. The more recent *Cisco Extensible Network Controller* (XNC) provided the original controller foundation for the ODL controller, discussed below.

Cisco has two other SDN-via-API controllers, the *Application Policy Infrastructure Controller—Enterprise Module* (APIC-EM) and the *Application Policy Infrastructure Controller—Data Center* (APIC-DC). The APIC-EM solution is targeted at enterprise customers who wish to control access and prioritize traffic between end users and resources such as servers. APIC-EM is unique in that its southbound protocol is actually the CLI. The controller takes higher-level policy information about users, resources, and access rights, and converts those into CLI settings on the switches and routers in the enterprise network. The APIC-DC solution is targeted at data centers, and it is built using an entirely new protocol called *OpFlex*. The OpFlex protocol is policy-based, meaning that the controller exchanges policy-level information with the devices in the network. The devices interpret the policy instructions and make the appropriate configuration modifications.

OpenDaylight

A hybrid approach is the ODL open source project [15]. ODL supports OpenFlow as one of its family of southbound APIs for controlling network devices. Used in this way, ODL clearly can fit into the Open SDN camp. The ODL controller also supports southbound APIs that program the legacy control plane on network devices, using protocol plugins such as NETCONF and BGP-LS/PCE-P, presented in Sections 6.2.2 and 6.2.3, respectively. When used with those legacy southbound APIs, it falls into the controller-based SDN via APIs category. Due to this chameleon-like role, we will not categorize it as exclusively either SDN via APIs or Open SDN. We discuss the ODL program in more detail in Chapters 7 and 13.

Juniper

Juniper has created an API-based controller called *Contrail*. This controller uses NETCONF and *Extensible Messaging and Presence Protocol* (XMPP) to create a data center solution which has parallels in the service provider space, using MPLS-like features to route traffic through the network. As Juniper has historically been a significant force in service provider markets, this provides a convenient segue for them into data centers. Juniper has open-sourced the base Contrail controller, calling this version *OpenContrail*, encouraging customers and vendors to contribute code to the project. At the time of this writing, there seems to be a smaller community of open source developers committed to OpenContrail than with ODL.

Arista

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 [16]. Arista is basing its company's messaging squarely around this API-centric definition of SDN. Arista claims that they support both centralized controllers and an alternative of distributed controllers that may reside on devices themselves.

6.2.6 RANKING SDN VIA APIs

In Table 6.1 we provide a ranking of SDN via APIs versus the other technologies discussed in this chapter. As we said in the introduction to this chapter, 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 may consider important. We also acknowledge that there may be criteria that are extremely important for some network professionals or users that are entirely absent from our list. For this, we apologize and 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, one criterion where SDN via APIs seems to exemplify a true SDN technology is in the area of improved network automation and virtualization. Networking devices which are only changed 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. As we stated at the start of Section 6.2, our discussion of SDN via APIs presumes use of a centralized controller model. For this reason, we give it a grade of high for centralized controller. The APIs defined by vendors are generally unique to their own devices, and, hence, usually proprietary.[2] 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. While it may be true that the Cisco APIs have been mimicked by its competitors in order to make their devices interoperable with Cisco-based management tools, this is not true interoperability. Cisco continually creates new APIs so as to differentiate themselves from competitors, which puts its competitors in constant catch-up mode. This is not the case with NETCONF and BGP-LS/PCE-P, however, as both of these are open specifications. While these two specifications are open, the lack of standardization of YANG models limits this openness somewhat. Thus, the grade for openness for SDN via APIs will be a dual ranking of $[Low, Medium]$ depending on whether the API is a proprietary or open protocol. 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, there is neither too much change nor a single point of failure. We rank it medium with respect to performance and scale, as 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 to 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.

DISCUSSION QUESTION:

SDN via APIs help customers to protect existing investments in their legacy equipment, but they can also help vendors protect their profit margins. Which do you think may have been the primary motivation? In what ways does SDN via APIs limit or even hinder technological progress in the networking domain?

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 was depicted in Fig. 4.12. 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 which are separate and independent of each other, even when VMs are running on the same server. As was shown in Fig. 4.12, a single physical server hosting multiple VMs can have each VM be a member of a separate virtual network. This VM

[2] 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.

can communicate with other VMs in its virtual network, but usually does not cross boundaries and talk to VMs which are part of a different virtual network. When it is desirable to have VMs in different networks communicate, they do so by 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 amongst 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 just described. The virtual ports of these virtual switches correspond to the *Virtual Tunnel Endpoints* (VTEPs) defined in Section 4.6.3. 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 Fig. 4.13. We will provide a detailed discussion about MAC-in-IP encapsulation in Chapter 8.

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 a virtual switch 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 which serve to 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 if 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). While Host A uses its ARP broadcast mechanism to resolve Host B's MAC address, 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 where 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.
5. The destination VTEP receives the packet, strips off the outer encapsulation information, and forwards the original frame to Host B.

Fig. 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 [17],[3] 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.

[3] Since we focus on the NVP component in this work, we will usually refer to that component by its original, preacquisition name, rather than NSX.

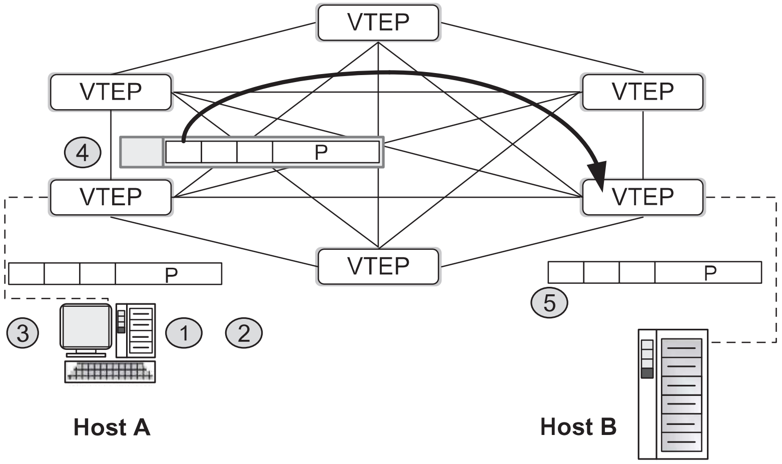

FIG. 6.11

Overlay operation.

Juniper's JunosV Contrail [18], however, does not use OpenFlow. We introduced Contrail in Section 6.2.5 as an example of the use of NETCONF as SDN via APIs. While that is indeed the case, it is also true that Contrail is a controller-based virtual overlay approach to network virtualization. It uses XMPP to program the control plane of the virtual switches in its overlay networks. Because the southbound APIs are NETCONF and XMPP, it is indeed a standards-based approach. Since the control plane still resides on those virtual switches and Contrail does not directly program the data plane, we do not consider it Open SDN. It is clearly an SDN via Hypervisor-Based Overlays approach, though, as well as a NETCONF-based SDN via APIs approach.

Another important SDN via Hypervisor-Based Overlays offering is IBM's *Software Defined Network for Virtual Environments* (SDN VE) [19]. 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. As 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 will give this a ranking of medium with respect to device simplification.

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	High	High
Network Automation and Virtualization	High	High	High
Openness	High	[Low, Medium][b]	[Medium, High][c]
Too Much Change	Low	High	Medium
Single Point of Failure	Low	Medium	N/A[d]
Performance and Scale	Medium	Medium	Medium
Deep Packet Inspection	Low	Low	Medium
Stateful Flow Awareness	Low[a]	Low	Medium
MAC Forwarding Table Overflow	High	Low	High
VLAN Exhaustion	High	Low	High

[a] *While OpenFlow V.1.5 describes the possibility of maintaining flow state across packets, there are few details and no commercial implementations that we are aware of so we leave this ranked low. (see Section 6.1.5).*

[b] *When the API is a proprietary protocol, we do not consider this open, hence it gets a low ranking. As we point out in Section 6.2.6, some APIs such as NETCONF are somewhat more open, and in those cases this deserves a medium ranking.*

[c] *Either medium or high depending on the openness of the particular implementation (see Section 6.3.4).*

[d] *Since SDN via Hypervisor-Based Overlays may or may not be based on the centralized controller concept we feel that ranking is not applicable (see Section 6.3.4).*

A sidebar 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 a 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 this 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 only deals with 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 overrely on VLANs for virtualization.

To summarize, as SDN via Hypervisor-Based Overlay Networks was specifically designed for the data center, it was not tailored to other environments, such as the Campus, Service Provider, Carrier, and Transport Networks. For this reason, while 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 9, 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. While SDN via Hypervisor-Based Overlays is an important solution gaining considerable traction in data center environments, it does not necessarily offer the device simplification or openness of Open SDN.

DISCUSSION QUESTION:

SDN via Overlays run across existing hardware and utilize a software-only design. In what ways is this solution is superior to Open SDN and SDN via APIs? Do you think that future networking devices and solutions will attempt to create environments that are software-only?

6.4 SDN VIA OPENING UP THE DEVICE

There is a another approach to Software Defined Networking which 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 which 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 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 which *are themselves open*, capable of running software which has been written by the open source community.

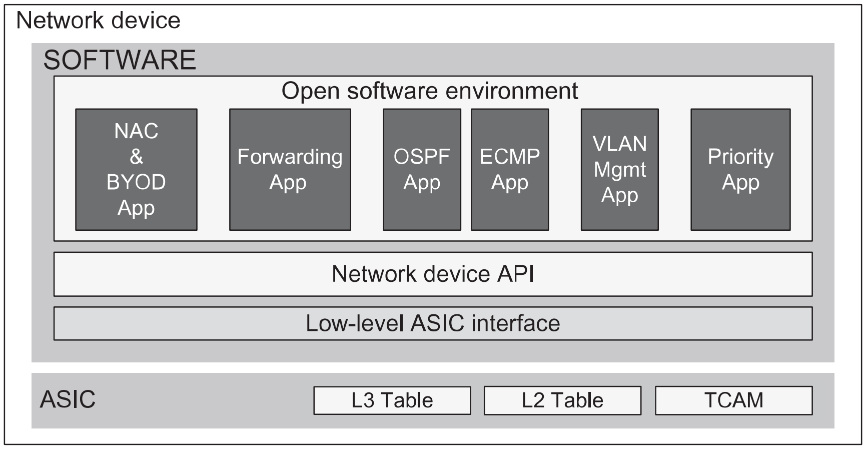

FIG. 6.12

Opening up the device.

This *Open Device* concept is represented in Fig. 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 which 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. In order to be successful, there needs to be some level of standardization of the APIs which 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 which 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 only possible 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 Device 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 [20] 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 [20]. 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. While 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 their open source *Data Plane Development Kit* (DPDK) [21,22]. 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 becoming a reality.

DISCUSSION QUESTION:

There is a push towards creating networking devices that are open, or run Linux inside, such that software can be written to extend or enhance the capabilities of devices. What are the ways in which this is a good idea? And are there ways in which this is a bad idea?

6.5 NETWORK FUNCTIONS VIRTUALIZATION

We cannot complete our chapter on SDN alternatives without a mention of *Network Functions Virtualization* (NFV), as there is confusion in the industry as to whether or not NFV is a form of SDN [23]. 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 CPUs are the norm in data centers. While 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 [24] 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. While NFV is not just another approach to SDN, the relationship between the two is sufficiently intimate that we dedicate the entire Chapter 10 to NFV.

6.6 ALTERNATIVES OVERLAP AND RANKING

While we have tried to define a clear framework and delineated four distinct SDN alternatives, the lines between them 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 Open SDN.
- As explained in Section 6.3.3, Juniper's Contrail is both an SDN via Hypervisor-Based Overlays approach as well as a NETCONF-based SDN via APIs approach.

In Fig. 6.13 we attempt to depict the possible overlap between our four major SDN approaches. The category SDN via Opening Up the Device is a difficult one for comparison. As Fig. 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 to stick with any current or future paradigm. For example, Big Switch's Indigo provides OpenFlow switch software freely to device vendors who wish to use it. This should encourage ODMs to build standard hardware and software so

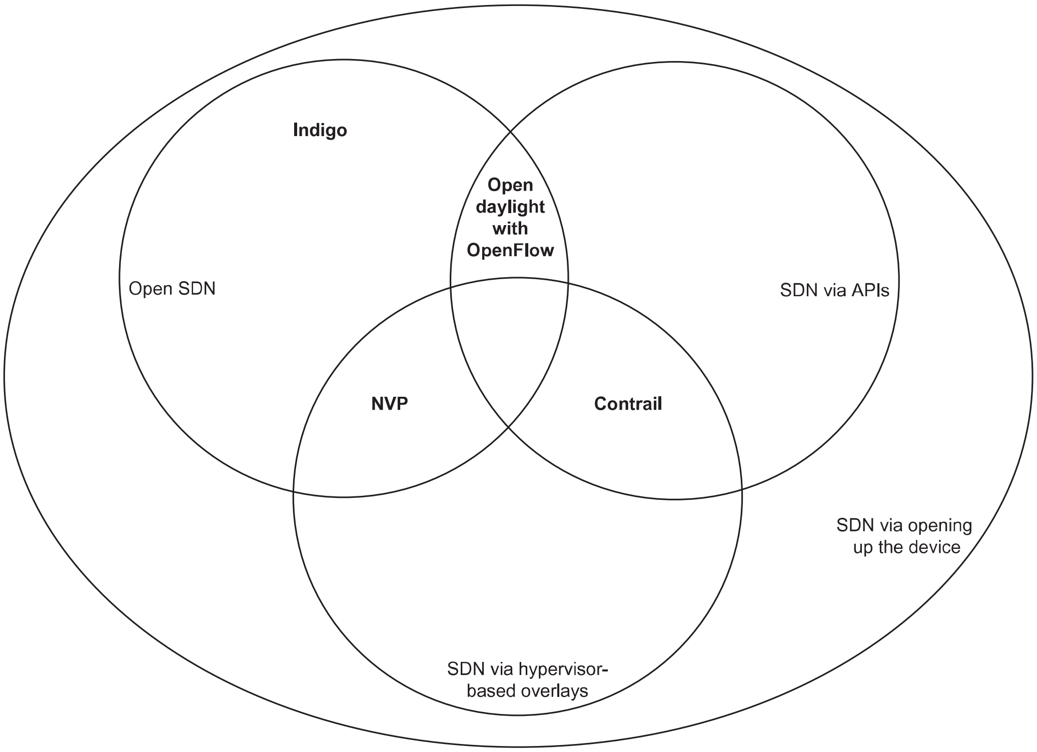
Indigo
Open daylight with OpenFlow
Open SDN
SDN via APIs
NVP
Contrail
SDN via opening up the device
SDN via hypervisor-based overlays

FIG. 6.13

SDN alternatives overlap.

that they can use software like 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 Open SDN as it is an implementation of OpenFlow. For this reason, we exclude SDN via Opening Up the Device from Table 6.1, as 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 where 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 is using Open SDN and, when appropriate, the ranking will reflect those two subcategories (i.e., [*notOpenSDN*, *OpenSDN*]).

DISCUSSION QUESTION:

We have described different alternatives to SDN. Which SDN alternative do you think is the most promising for the future of networking? The most revolutionary? The most pragmatic? The most likely to succeed in the long run?

6.7 CONCLUSION

In this chapter we have examined some well-publicized drawbacks of Open SDN and looked at three alternative definitions of SDN. We have 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 Opening Up the Device, has strengths and areas where they address 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 Chapters 8 and 9 will be how Open SDN can be used to address these real life problems. We will consider these alternative definitions where they have proven to be particularly effective at addressing a given use case. Before that, in the next chapter we survey new models of SDN protocols, controllers and applications that have emerged since this book was first published.

REFERENCES

[1] Silva AS, Smith P, Mauthe A, Schaeffer-Filho A. Resilience support in software-defined networking: a survey. Comput Netw 2015;92(1):189–207.

[2] Casado M. Scalability and reliability of logically centralized controller. Stanford CIO Summit, June 2010.

[3] Wireless LAN Controller (WLC) Mobility Groups FAQ. Cisco Systems. Retrieved from: http://www.cisco.com/en/US/products/ps6366.

[4] HP MSM7xx Controllers Configuration Guide. Hewlett-Packard, March 2013.

[5] Yeganeh S, Ganjali Y. Kandoo: a framework for efficient and scalable offloading of control applications. In: Hot Topics in Software Defined Networking (HotSDN), ACM SIGCOMM, August 2012, Helsinki; 2012.

[6] Curtis AR, Mogul JC, Tourrilhes J, Yalagandula P, Sharma P, Banerjee S. Devoflow: scaling flow management for high-performance networks. In: Proceedings of the ACM SIGCOMM 2011 conference, SIGCOMM'11. New York, NY, USA: ACM; 2011. p. 254–65.

[7] Veisllari R, Stol N, Bjornstad S, Raffaelli C. Scalability analysis of SDN-controlled optical ring MAN with hybrid traffic. In: IEEE international conference on communications (ICC); 2014. p. 3283–8. http://dx.doi.org/10.1109/ICC.2014.6883827.

[8] Zhang Y, Natarajan S, Huang X, Beheshti N, Manghirmalani R. A compressive method for maintaining forwarding states in SDN controller. In: Proceedings of the third workshop on hot topics in software defined networking, HotSDN'14. New York, NY, USA: ACM; 2014. p. 139–44. http://dx.doi.org/10.1145/2620728.2620759.

[9] Zhou W, Li L, Chou W. SDN northbound REST API with efficient caches. In: IEEE international conference on web services (ICWS). 2014. p. 257–64. http://dx.doi.org/10.1109/ICWS.2014.46.

[10] 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. In: European Workshop on Software Defined Networking (EWSDN), October 25–26, Darmstadt; 2012.

[11] Creating innovative embedded applications in the network with the Junos SDK. Juniper Networks, 2011. Retrieved from: http://www.juniper.net/us/en/local/pdf/whitepapers/2000378-en.pdf.

[12] TR-069 CPE WAN Management Protocol. Issue 1, Amendment 4, Protocol Version 1.3, July 2011. The Broadband Forum—Technical Report; 2011.

[13] Configuring Policy-Based Routing. Cisco Systems. Retrieved from: http://www.cisco.com/en/US/docs/switches/lan/catalyst4500/12.2/20ew/configuration/guide/pbroute.html.

[14] Cisco onePK Developer Program. Cisco Systems. Retrieved from: http://developer.cisco.com/web/onepk-developer.

[15] Open Daylight Technical Overview. Retrieved from: http://www.opendaylight.org/project/technical-overview.

[16] Bringing SDN to reality. Arista Networks, March 2013. Retrieved from: http://www.aristanetworks.com.

[17] Nsx. VMware. Retrieved from: https://www.vmware.com/products/nsx/.

[18] McGillicuddy S. Contrail: the Juniper SDN controller for virtual overlay network. Techtarget, May 9, 2013. Retrieved from: http://searchsdn.techtarget.com/news/2240183812/Contrail-The-Juniper-SDN-controller-for-virtual-overlay-network.

[19] Chua R. IBM DOVE Takes Flight with New SDN Overlay, Fixes VXLAN Scaling Issues. SDN Central, March 26, 2013. Retrieved from: http://www.sdncentral.com/news/ibm-dove-sdn-ve-vxlan-overlay/2013/03.

[20] OpenWRT—Wireless Freedom. Retrieved from: https://openwrt.org/.

[21] Intel DPDK: Data Plane Development Kit. Retrieved from: http://www.dpdk.org.

[22] McGillicuddy S. Intel DPDK, switch and server ref designs push SDN ecosystem forward. Techtarget, April 23, 2013. Retrieved from: http://searchsdn.techtarget.com/news/2240182264/Intel-DPDK-switch-and-server-ref-designs-push-SDN-ecosystem-forward.

[23] Pate P. NFV and SDN: what's the difference? SDN Central, March 30, 2013. Retrieved from: http://www.sdncentral.com/technology/nfv-and-sdn-whats-the-difference/2013/03.

[24] Jacobs D. Network functions virtualization primer: software devices take over. Techtarget, March 8, 2013. Retrieved from: http://searchsdn.techtarget.com/tip/Network-functions-virtualization-primer-Software-devices-take-over.