CHAPTER

SDN APPLICATIONS 12

We now descend from the conceptual discussions about SDN use cases and take a deeper dive into how SDN applications are actually implemented. This chapter focuses on SDN application development in both an open SDN and an SDN via APIs environment. Most SDN via Overlays offerings today are bundled solutions which come ready-made with a full set of functionality, generally without the ability to add applications to those solutions. As such, we will not contemplate that type of application development.

In particular, we will examine attributes and considerations related to creating reactive versus proactive applications, as well as creating internal versus external applications. These different application types were introduced in Section 7.4.1. Since the creation of internal, reactive applications is a more ambitious endeavor, we present a sample internal, reactive application, running inside the Floodlight controller framework. Although Floodlight today generates less interest than some newer controllers, the principles for creating this type of application are fairly consistent across controllers in common use, and Floodlight is the most basic of these. That reason, combined with the fact that it evolved out of the original OpenFlowJ [1] libraries make Floodlight a suitable training environment.

It is worth noting that the crisp distinction between reactive and proactive applications that we make in this chapter stems from the original use of these terms within the OpenFlow paradigm. With some of the non-OpenFlow SDN alternatives introduced in Chapter 7, these distinctions may not be absolute. In particular, there may be applications that exhibit both reactive and proactive characteristics.

Many detailed functional block diagrams are included in this chapter, as well as a fair amount of sample source code which serve as helpful examples for the network programmer.

12.1 TERMINOLOGY

Consistent terminology is elusive in a dynamic technology such as SDN. Consequently, certain terms are used in different ways, depending on the context. To avoid confusion, in this chapter we will adhere to the following definitions:

- *Network device*: When referring generically to a device such as a router, switch, or wireless access point, in this chapter we will use the term *network device* or *switch*. These are the common terms used in the industry when referring to these objects. Note that the Floodlight controller uses the term *device* to refer to *end user device*. Hence, in sections of code using Floodlight or supporting text, the word *device* used alone means *end user node*.

- *End user node*: There are many examples of end user devices, including desktop computers, laptops, printers, tablets, and servers. The most common terms for these end user devices are *host*, and *end user node*; we will use *end user node* throughout this chapter.
- *Flow*: In the early days of SDN, the only controller-device protocol was OpenFlow, and the only way to affect the forwarding behavior of network devices was through setting flows. With the recent trends toward NETCONF and other protocols to effect forwarding behavior by setting routes and paths, there is now no single term for that which the controller changes in the device. The controller's actions decompose into the fundamentals of create, modify, and delete actions, but the entity acted upon may be a flow, a path, a route, or, as we shall see, an optical channel. For the sake of simplicity, we sometimes refer to all these things generically as *flows*. The reader should understand this to mean either setting flows, or configuring static routes, modifying the RIB, setting MPLS LSPs, among other things, depending on the context.

12.2 BEFORE YOU BEGIN

Before embarking on a project to build an SDN application, one must consider a number of questions. Answering these questions will help the developer to know how to construct the application, including what APIs are required, which controller might be best suited for the application, and what switch considerations should be taken into account. Some of these questions are:

- *What is the basic nature of the application?* Will it spend most of its time reacting to incoming packets which have been forwarded to it from the switches in the network? Or will it typically set flows proactively on the switches and only respond to changes in the network which require some type of reconfiguration?
- *What is the type and nature of network with which the application will be dealing?* For example, an access control application for a campus network might be designed differently than a similar application for a data center network.
- *What is the intended deployment of the controller with respect to the switches it controls?* A controller which is physically removed from the switches it controls (e.g., in the cloud) will clearly not be able to efficiently support an SDN application which is strongly reactive in nature because of the latency involved with forwarding packets to the controller and receiving a response.
- *Are the SDN needs purely related to the data center and virtualization?* If so, perhaps an SDN Overlay solution is sufficient to meet current needs and future needs related to the physical underlay network can be met later, as OpenFlow-related technologies mature.
- *Will the application run in a green-field environment or will it need to deal with legacy switches which do not support SDN?* Only in rare situations is one able to build an Open SDN application from the ground up. Typically, it is required to use existing equipment in the solution. It is important to understand if the existing switches can be upgraded to have OpenFlow capability and whether or not they will have hybrid capability, supporting both OpenFlow and legacy operation. If the switches do not support OpenFlow, then it is important to understand the level of SDN via APIs support that is available.
- *What type of APIs are available on legacy switches and routers? What is their level of capability?* If the application requires NETCONF, what YANG models does each type of switch or router

support, and are the different YANG models consistent? The answer to these questions will dictate the capabilities of your SDN application, and the extent to which you will be able to implement SDN capabilities in your legacy network.

- *What is the level of programming ability present in your development team(s)?* Only software teams with deep experience in the native language of the controller will be able to develop internal SDN applications; this may be especially true if OpenDaylight is your controller, since you must develop applications in a *model-driven service abstraction layer* (MD-SAL) environment. Since the native language of current mainstream controllers is Java, this currently implies that the team have deep experience with Java. If your software team is more comfortable developing applications in a language such as Python, then they will gravitate toward external applications.
- *How many switches will be managed by your application?* Scalability is more of an issue with reactive applications, since packets will periodically be forwarded to your application for processing, and this limits the number of switches or routers that can be controlled by a single application/controller instance.

The answers to these and similar questions will help the developer to understand the nature of the SDN application. Assuming that an Open SDN application will be developed, the developer will need to decide between two general styles of SDN applications, *reactive* and *proactive*.

DISCUSSION QUESTION

We have listed a number of factors to be considered when determining how to build your SDN application. Can you think of any other considerations? Do you believe that certain of these considerations are more important than others, and if so, which ones?

12.3 APPLICATION TYPES

In Section 7.4.1 we introduced the notion of reactive versus proactive applications, and internal vs external applications. In the sections that follow, we take a deeper look at these application types, and do so specifically from the perspective of SDN application development.

12.3.1 REACTIVE VS PROACTIVE APPLICATIONS

The reader should recall from Section 7.4.1 that the main difference between these two application paradigms is that reactive applications will periodically receive packets forwarded from a switch in order for the SDN application to process the packet. The reactive application then sends instructions back to the switch prescribing how to handle this packet, and whether flow changes are required. In the case of reactive applications, the communication between the switch and the controller will typically scale with the number of new flows injected into the network. The switches may often have relatively few flow entries in their flow tables, as the flow idle timers are set to match the expected duration of each flow in order to limit the growth of the flow table. This can result in the switch frequently receiving packets, which match no rules. In the reactive model, those packets are encapsulated in PACKET_IN messages and forwarded to the controller and thence to an application. The application inspects the packet and determines its disposition. The outcome is usually to program a new flow entry in the switch(es) so that the next time this type of packet arrives, it can be handled locally by the switch itself. The application will often program multiple switches at the same time so that each switch along the path of the flow will have a consistent set of flow entries for that flow.[1] The original packet will often be returned to the switch so that the switch can handle it via the newly installed flow entry. The kind of applications that naturally lend themselves to the reactive model are those that need to see and respond to new flows being created. Examples of such applications include per-user firewalling and security applications that need to identify and process new flows.

On the other hand, proactive SDN applications feature less communication emanating from the switch toward the controller. The proactive SDN application sets up the switches in the network with either flow entries or configuration attributes which are appropriate to deal with incoming traffic before it arrives at the switch. Events which trigger changes to the flow table or configuration changes to switches may come from mechanisms which are outside the scope of the switch-to-controller secure channel. For example, some external traffic monitoring module will generate an event which is received by the SDN application, which will then adjust the flows on the switches appropriately. Applications that naturally lend themselves to the proactive model usually need to manage or control the topology of the network. Examples of such applications include new alternatives to spanning tree and multipath forwarding. With the addition of support for legacy protocols via protocol plugins such as NETCONF and BGP-LS/PCE-P, configuration of paths, routes, and traffic shaping are possible via proactive applications.

Reactive programming may be more vulnerable to service disruption if connectivity to the controller is lost. Specifically, when the failure mode is set to *fail secure* the switch is more likely to be able to continue normal operation since many flow table entries will have been prepopulated in the proactive paradigm. Fail secure mode specifies that OpenFlow packet processing should continue despite the loss of connectivity to the controller.[2]

Another advantage of the proactive model is that there is no additional latency for flow setup as they are prepopulated. A drawback of the proactive model is that most flows will be wildcard-style, implying aggregated flows, and thus less-fine granularity of control. We discuss these issues in more detail in the next subsections.

12.3.2 INTERNAL VS EXTERNAL APPLICATIONS

We saw in Section 7.4.3 that internal applications are those that run *inside* the OSGi container of the controller. Conversely, external applications run outside that container, meaning they can run anywhere, and typically use the RESTful APIs provided by the controller. One of the easiest ways to compare internal and external applications is via a table of their limitations and capabilities. This comparison is summarized in Table 12.1.

[1]Proper synchronization of flow programming on a series of switches can be a challenge with OpenFlow. This is one of the reasons for the growing interest in the intents-based style of northbound API on the controller, as discussed in Section 7.3.5.
[2]See Section 5.4.5 for an explanation of *failure mode*. If the failure mode is set to *fail stand-alone* mode then loss of controller connectivity will cause the flow tables to be flushed independent of the application style.

Table 12.1 Internal vs External Applications

Feature	Internal	External
Programming language	Java	Any language
APIs used	Java	REST
Deployment	Inside controller	Anywhere
Performance	High	Low
Reactive apps	Yes	No
Proactive apps	Yes	Yes
Failure affects controller	Yes	No

The following points provide background for the entries in Table 12.1:

- Internal applications must be written in Java (assuming the controller's native language is Java, which is true of almost all controllers); external applications can be written in any language.
- Internal applications will use Java APIs on the controller; external applications will use RESTful APIs.
- Internal applications must be deployed inside the environment (typically OSGi) of the controller, running locally with the controller; external applications can run anywhere, even in the cloud if so desired.
- Internal applications are relatively faster; whereas external applications are slower due to: (1) using the REST API, (2) possibly being written in a slower language such as Python, and (3) potentially running remotely from the controller.
- Internal applications are capable of running reactive applications (receiving packets forwarded from an OpenFlow switch); external applications are not.
- Both internal and external applications are capable of running proactive applications.
- Internal applications are running inside the controller, and failures can have negative effects on the operation of the controller; for external applications, this danger is less severe.

12.3.3 DETAILS: REACTIVE SDN APPLICATIONS

We have seen that reactive applications are capable of receiving packets that have been forwarded to the controller from switches in the network. As of this writing, the only SDN protocol that performs that function is OpenFlow, hence, a reactive application is implicitly an OpenFlow application.

Since reactive applications must also run inside the controller in order to receive asynchronous notification of forwarded packets, they must also be internal applications, running inside the OSGi container of the controller. External applications using the RESTful APIs are unable to receive these asynchronous notifications, since REST is a unidirectional protocol. And while certain mechanisms can be put in place for bidirectional communication (e.g. web sockets), the latency involved is impractical for the prompt handling of the possibly huge number of incoming packets. Reactive applications have the ability to register listeners, which are able to receive notifications from the controller when certain events occur. In Section 12.4 we provide a brief historical overview of controllers that have played an important role in the evolution of SDN. Floodlight and Beacon are notable in this list. Some important listeners available in these two historically popular controller packages are:

- *SwitchListener*. Switch listeners receive notifications whenever a switch is added, removed or has a change in port status.
- *DeviceListener*. Device listeners are notified whenever a device (an end-user node) has been added, removed, moved (attached to another switch) or has changed its IP address or VLAN membership.
- *MessageListener*. Message listeners get notifications whenever a packet has been received by the controller and the application then has a chance to examine it and take appropriate action.

These listeners allow the SDN application to react to events which occur in the network and to take action based on those events.

When a reactive SDN application is informed of an event, such as a packet which has been forwarded to the controller, change of port state, or the entrance of a new network device or host into the network, the application has a chance to take some type of *action*. The most frequent event coming into the application would normally be a packet arriving at the controller from a switch, resulting in an action. Such actions include:

- *Packet-specific actions*. The controller can tell the switch to drop the packet, to flood the packet, to send the packet out a specific port, or to forward the packet through the NORMAL non-OpenFlow packet processing pipeline as described in Section 5.3.4.
- *Flow-specific actions*. The controller can program new flow entries on the switch, intended to allow the switch to handle certain future packets locally without requiring intervention by the controller.

Other actions are possible, some of which may take place outside the normal OpenFlow control path, but the packet-specific and flow-specific actions constitute the predominant behavior of a reactive SDN application.

Fig. 12.1 shows the general design of a reactive application. Notice that the controller has a listener interface that allows the application to provide listeners for switch, device (end user node), and message (incoming packet) events. Typically a reactive application will have a module to handle packets incoming to the controller which have been forwarded through the message listener. This packet processing can then act on the packet. Typical actions include returning the request to the switch, telling it what to do with the packet (e.g., forward out a specific port, forward NORMAL, or drop the packet). Other actions taken by the application can involve setting flows on the switch in response to the received packet, which will inform the switch what to do the next time it sees a packet of this nature.

For reactive applications, the last flow entry will normally be programmed to match any packet and to direct the switch to forward that otherwise unmatched packet to the controller. This methodology is precisely what makes the application *reactive*. When a packet not matching any existing rule is encountered, it is forwarded to the controller so that the controller can react to it via some appropriate action. A packet may also be forwarded to the controller in the event that it matches a flow entry and the associated action stipulates that the packet be passed to the controller.

In the reactive model the flow tables tend to continually evolve based on the packets being processed by the switch and by flows aging out. Performance considerations arise if the flows need to be reprogrammed too frequently, so care must be taken to appropriately set the idle timeouts for the flows. The point of an idle timer is to clear out flow entries after they have terminated or gone inactive, so it is important to consider the nature of the flow and what constitutes inactivity when setting the timeout.

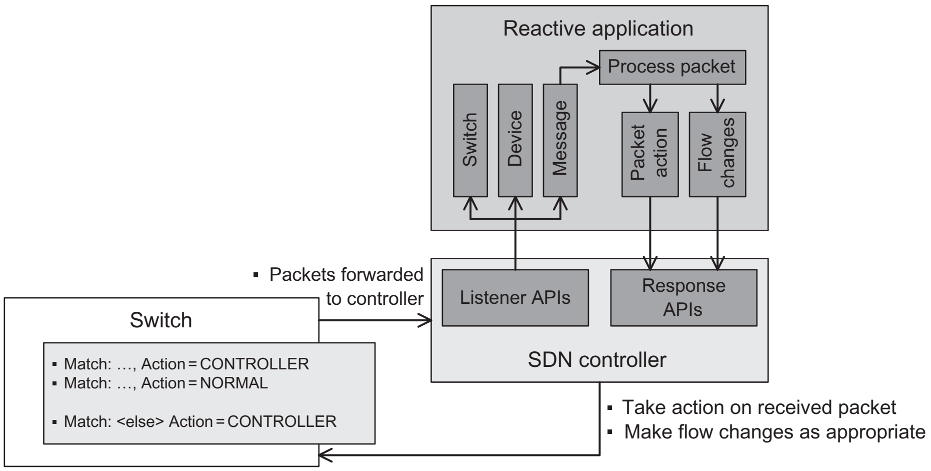

Reactive application
Process packet
Switch
Device
Message
Packet action
Flow changes
Listener APIs
Response APIs
SDN controller
▪ Packets forwarded to controller
Switch
▪ Match: …, Action = CONTROLLER
▪ Match: …, Action = NORMAL
▪ Match: <else> Action = CONTROLLER
▪ Take action on received packet
▪ Make flow changes as appropriate

FIG. 12.1

Reactive application design.

Unduly short idle timeouts will result in too many packets sent to the controller. Excessively long timeouts can result in flow table overflow. Knowing an appropriate value to set a timeout requires familiarity with the application driving the flow. As a rough rule of thumb, though, timeouts would be set on the order of tens of seconds rather than milliseconds or minutes.

12.3.4 DETAILS: PROACTIVE SDN APPLICATIONS

Proactive applications can be implemented using either the native API (e.g., Java) or using RESTful APIs, but most often these proactive applications are also external applications using the REST interface.

One important point to remember is that proactive applications are *dynamic*. That is, they can regularly make modifications to networking behavior. Fig. 12.2 shows that proactive applications take notifications from external events, and make changes to the behavior of network devices in the network. In the figure, we show the application setting flows at a relatively low level on the network device (i.e., the *forwarding information base* (FIB) level as described in Section 7.2.1). Proactive applications can also alter behavior of networking devices at the network-wide *routing information base* (RIB) and config layers as well.

In contrast to reactive APIs, these controller RESTful APIs can operate at different levels, depending on the northbound API being used. They can be primitive, basically allowing the application developer to configure OpenFlow flow entries on the switches, or to configure static routes directly on routers using NETCONF. These RESTful APIs can also be high-level, providing a level of abstraction above the network devices, so that the SDN application interacts with network components such as virtual networks, abstract network device models or with the RIB, rather than with the physical switches themselves.

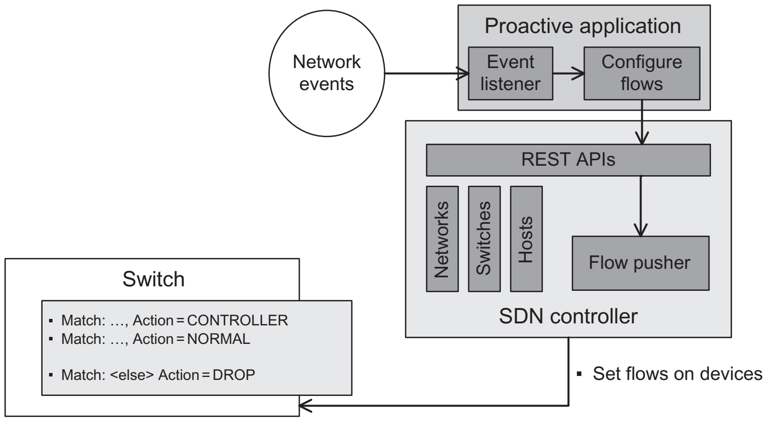


FIG. 12.2

Proactive application design.

Another difference between this type of application and a reactive one is that using RESTful APIs means there is no need for *listening* functionality. In the reactive application the controller asynchronously invokes methods in the SDN application stimulated by events such as incoming packets and the discovery of network devices. This type of asynchronous notification is not possible with a unidirectional RESTful API, which is invoked on the controller by the application rather than the other way around. While it is conceivable to implement an asynchronous notification mechanism from the controller toward a proactive application, generally, proactive applications are stimulated by sources external to the OpenFlow control path. The traffic monitor described in Section 12.12 is an example of such an external source.

Fig. 12.2 shows the general design of a purely proactive application. In such an application there are no *packet* listeners receiving packets that have been forwarded by switches in the network. Such listeners are not a natural fit with a one-way RESTful API, wherein the application makes periodic calls to the API and the API has no means of initiating communication back to the application. Proactive applications instead rely on stimulus from external *network* or *user events*. Such stimuli may originate from traffic monitors including an *intrusion detection system* (IDS) such as SNORT, or external applications like a server virtualization service, which notifies the application of the movement of a virtual machine from one physical server to another. Stimuli may also be in the form of user-initiated input which causes changes to the manner in which packets are handled—for example, a request for certain traffic types to be expedited or rerouted.

The RESTful APIs are still able to *retrieve* data about the network, such as domains, subnets, switches, and hosts, from the SDN controller. This information may be gathered by the controller through various protocols, including OpenFlow, *Link Layer Discovery Protocol* (LLDP), BGP, and BGP-LS. SDN controllers typically will collect and correlate this discovery and topology information, and make it available via RESTful APIs to SDN applications. One of the major advantages of running SDN applications on top of a controller's API is that the difficult but indispensable task of discovering topological information about a network is done by the controller on which your application is running.

As shown in Fig. 12.2, in an OpenFlow context the last flow entry will typically be to DROP unmatched packets. This is because proactive applications attempt to *anticipate* all traffic and program flow entries accordingly. As a consequence, packets which do not match the configured set of rules are discarded. We remind the reader that the match criteria for flow entries can be programmed such that most arriving packet types are expected and match some entry before this final DROP entry. If this were not the case, the purely proactive model could become an expensive exercise in dropping packets!

As mentioned earlier, there will be hybrid reactive-proactive applications which utilize a language such as Java for listeners and communicate via some external means to the proactive part of the application. The proactive component would then program new flow entries on the switch. As applications become more complex, it is likely that this hybrid model will become more common. The comment made above about arriving packets in a proactive application matching some flow entry before the final DROP entry is also germane to the case of hybrid applications. The proactive part of such hybrid applications will have programmed higher-priority flow entries so that only a reasonable number of packets match the final DROP entry. In such a hybrid model, the final entry will send all unmatched packets to the controller and so it is important to limit this to a rate that the controller can handle.

12.3.5 DETAILS: INTERNAL SDN APPLICATIONS

Internal SDN applications on the currently dominant controllers are all Java-based and share the following traits:

- *OSGi*: Internal applications on Java-based controllers will need to run in some type of Java environment supporting multiple modules, and OSGi is the most common framework. Apache Karaf [2] is the most common OSGi implementation for SDN controllers today.
- *Reactive application support*: Controllers providing capabilities today (e.g., Floodlight, HP, ODL, and ONOS) provide the ability to create *packet listeners* for receiving packets forwarded from switches.
- *Use of archetypes*: Considering the complexity of the predominant controllers' internal design, application development would be challenging without a software project management system tailored to such environments. Maven [3] is such a system, and is used for controller application development within ODL, ONOS, and the HP VAN controller. Maven includes the *archetype* [4] functionality to build a template project in order to bootstrap an application development effort.

Most current controllers allowing internal SDN applications share the same conceptual foundation as their common ancestor, Beacon. The OpenFlow APIs provided with these controllers have evolved to support newer Java idioms and current versions of OpenFlow. Examples of controllers falling into this category include Floodlight, the HP VAN SDN Controller, and ONOS. Even the original version of ODL, Hydrogen, followed this pattern. However, with the releases of Helium and Lithium, ODL embarked on the new MD-SAL internal architecture.

MD-SAL presents a new architecture for creating SDN applications and internal modules. From a software development point of view, the main aspects of MD-SAL are as follows:

- *YANG*: ODL has created the requirement that every MD-SAL application must define itself using a YANG model. We described YANG models in Section 7.2.2. Unlike the use of YANG by the

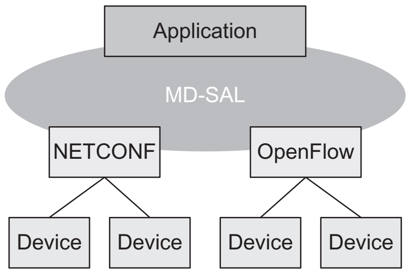


FIG. 12.3

MD-SAL overview.

NETCONF protocol, with MD-SAL YANG is used to define the application's data, for *remote procedure calls* (RPCs), and for notifications. Therefore, the first step in the process of application development is the definition of the relevant YANG model.

- *Model-based network device communication*: ODL incents application developers to interact with *models* of network devices rather than directly with the network devices themselves. This level of abstraction is intended to allow the controller APIs to be independent of the protocols beneath them. Fig. 12.3 shows a single application using APIs allowing communication via either NETCONF or OpenFlow, because it is communicating with the network device model, rather than the network device itself.
- *Standard APIs*: MD-SAL enforces a particular style of communication, using either an auto-generated RESTful API for the application, or through the use of a common YANG data tree. The RESTful APIs are created as part of the standard MD-SAL build process, and hence are common and open to other applications. The YANG data tree holds all of the information for the entire controller, and application data resides in one branch of that tree. Through this consistent use of YANG for all applications running in the controller, applications can access and make modifications to other applications' public data, providing a common and secure means of interapplication communication.

Because of the complexity of the MD-SAL environment, building internal applications on ODL confronts the developer with a relatively steep learning curve. More information on MD-SAL is available in [5,6]. It is important to consider the constraints and benefits presented previously when choosing between the internal and external application paradigms in ODL.

12.3.6 DETAILS: EXTERNAL SDN APPLICATIONS

External SDN applications are the simplest to create, since (a) they can be written in any programming language, (b) they can run in many places in the network, (c) they can use many types of southbound protocol plugins (e.g., OpenFlow, NETCONF, BGP-LS/PCE-P), and (d) they use simple RESTful APIs provided by the controller. External applications can be created more expeditiously than their internal counterparts and are useful for prototyping as well as for production use. There are no strict rules to follow regarding the design of an external SDN application. The application developer is entirely in control of the application's execution environment. The developer may choose to instantiate the application on the controller, on a remote system or even in the cloud. Hence, there are no detailed guidelines for external SDN application development. For these reasons, the external application paradigm has surpassed the internal model as a means of creating SDN solutions.

DISCUSSION QUESTION

In the beginning years of SDN, internal reactive applications attracted the most attention. Why do you think this is? Do you think that these types of applications are still the most important or the most interesting?

12.4 A BRIEF HISTORY OF SDN CONTROLLERS

The Beacon controller is truly a seminal controller in that much of the basic OpenFlow controller code in Floodlight and OpenDaylight was derived directly from Beacon. As we pointed out in Section 11.1.1, Beacon itself was based on early work performed at Stanford by David Erickson and Rob Sherwood. Both Beacon and Floodlight are based on OpenFlowJ for the core Java OpenFlow implementation. Floodlight is maintained by engineers from BigSwitch. While the distributions of Beacon and Floodlight are now distinct and different applications are packaged with each, Beacon remains close enough to Floodlight that we will not elaborate on it further at the detailed implementation level presented here.

We should point out that there are a number of other OpenFlow controllers available. For example, the NOX controller has been used in a considerable number of research projects. We provide basic information about a number of open source controller alternatives in Section 13.8. As of this writing, two controllers are predominant, OpenDaylight, and ONOS. We will explore these controllers in more detail in Section 12.7.

12.5 USING FLOODLIGHT FOR TRAINING PURPOSES

The application examples we provide in this chapter use the Floodlight controller. We have chosen Floodlight because it has historically been one of the most popular open source SDN controllers. Floodlight is not as popular today as it was in the earlier days of SDN, but its APIs are similar enough in nature to other Java OpenFlow APIs as to be useful as a training tool. It is possible to download the source code [7] and examine the details of the controller itself. Floodlight also comes with a number of sample applications, such as a learning switch and a load balancer, which provide excellent additional examples for the interested reader. The core modules of the Floodlight controller come from the seminal Beacon controller [8]. These components are denoted as *org.openflow* packages within the Floodlight source code. Since Beacon is the basis of the OpenFlow functionality present in a number of SDN controllers, the OpenFlow-related controller interaction described in the following sections would apply to those Beacon-derived controllers as well. With respect to ODL, while early versions incorporated much of the Beacon OpenFlow implementation, subsequent versions have evolved in order to implement OpenFlow as an MD-SAL module. For those later versions of ODL, the following sections would not directly apply.

In Section 12.10 we provide an example of proactive applications which use the RESTful *flow pusher* APIs. For this and other examples, remember that our use of *flows* does not restrict these designs to OpenFlow. As we explained in Section 12.1 we use the term flow in a generic sense, referring to the forwarding behavior of network devices, whether controlled by OpenFlow, NETCONF, PCE-P, or any other suitable protocol or protocol plugin.

In our first example, however, we will look in detail at the source code involved in writing a reactive SDN application. We will use this example as an opportunity to walk through SDN application source code in detail and explore how to implement the tight application-controller coupling that is characteristic of reactive applications.

12.6 A SIMPLE REACTIVE JAVA APPLICATION

In the common vernacular, a *blacklist* is *a list of people, products, or locations viewed with suspicion or disapproval*. In computer networking a blacklist is a list of hostnames or IP addresses, which are known or suspected to be malicious or undesirable in some way. Companies may have blacklists, which protect their employees from accessing dangerous sites. A school may have blacklists to protect children from attempting to visit undesirable web pages.

As implemented today, most blacklist products sit in the data path, examining all traffic at choke points in the network, looking for blacklisted hostnames or IP addresses. This clearly has disadvantages in terms of the latency it introduces to traffic as it all passes through that one blacklist appliance. There are also issues of scalability and performance if the number of packets or the size of the network is quite large. Cost would also be an issue since provisioning specialized equipment at choke points adds significant network equipment expense.

It would be preferable to examine traffic at the edge of the network, where the users connect. But putting blacklist-supporting appliances at every edge switch is impractical from a maintenance and cost perspective. A simple and cost-effective alternative is to use OpenFlow-supporting edge switches or wireless access points (APs), with a blacklist application running above the OpenFlow controller. In our simple application, we will examine both hostnames and IP addresses. We describe the application in the following section.

12.6.1 BLACKLISTING HOSTNAMES

We introduced an SDN solution to the blacklist problem in Section 9.3.3. We provide further details of that solution here. The first part of the blacklist application deals with hostnames. Checking for malicious or undesirable hostnames works as follows:

- A single default flow is set on the edge switches to forward all DNS traffic to the OpenFlow controller. (Note that this is an example of a proactive rule. Thus, while blacklist is primarily a reactive application, it does exhibit the hybrid reactive-proactive characteristic discussed in Section 12.3.4.)
- The blacklist application running on the controller listens for incoming DNS requests which have been forwarded to the controller.
- When a DNS request is received by the blacklist application it is parsed to extract the hostnames.

- The hostnames are compared against a database of known malicious or undesirable hosts.
- If any hostname in the DNS request is found to be bad, the switch is instructed to drop the packet.
- If all hostnames in the DNS request are deemed to be safe, then the switch is instructed to forward the DNS request normally.

In this way, with one simple flow entry in the edge switches and one simple SDN application on the controller, end users are protected from intentionally or inadvertently attempting to access bad hosts and malicious websites.

12.6.2 BLACKLISTING IP ADDRESSES

As described earlier in Section 9.3.3, websites sometimes have embedded IP addresses, or end users will attempt to access undesirable locations by providing IP-specific addresses directly, rather than a hostname, thus circumventing a DNS lookup. Our simple blacklist application works as follows:

- A single default flow is set on the edge switches, at a low priority, to forward all IP traffic that does not match other flows to the OpenFlow controller.
- The blacklist application listens for IP packets coming to the controller from the edge switches.
- When an IP packet is received by the blacklist application, the destination IP address is compared against a database of known malicious or undesirable IP addresses.
- If the destination IP address is found to be bad, the switch is instructed to drop the packet.
- If the destination IP address in the IP packet is deemed to be safe, then the switch or AP is instructed to forward the IP packet normally, and a higher-priority flow entry is placed in the switch which will explicitly allow IP packets destined for that IP address.
- The IP destination address flow entry was programmed with an idle-timeout which will cause it to be removed from the flow table after some amount of inactivity.

Note that a brute force approach to the blacklist problem would be to explicitly program rules for all known bad IP addresses such that any packet matching those addresses is directly dropped by the switch. While this works in principle, it is not realistic as the number of blacklisted IP addresses is potentially very large, and as we have pointed out, the number of available flow entries, while varying from switch to switch, is limited. This is exacerbated by the fact the flow entries for the blacklisted IP addresses generally should not expire, so they more or less permanently occupy valuable flow table space.

There are three major components to this application:

- *Listeners*. The module that listens for events from the controller.
- *Packet handler*. The module that handles incoming packet events and decides what to do with the received packet.
- *Flow manager*. The module that sets flows on the switches in response to the incoming packet.

Fig. 12.4 provides a more detailed view of the blacklist application introduced in Section 9.3.3. As a reactive application, Blacklist first programs the switch to forward unmatched packets to the controller. The figure depicts three listeners for network events: the switch listener, the device (end user node) listener, and the message listener. We show the device listener as it is one of the three basic

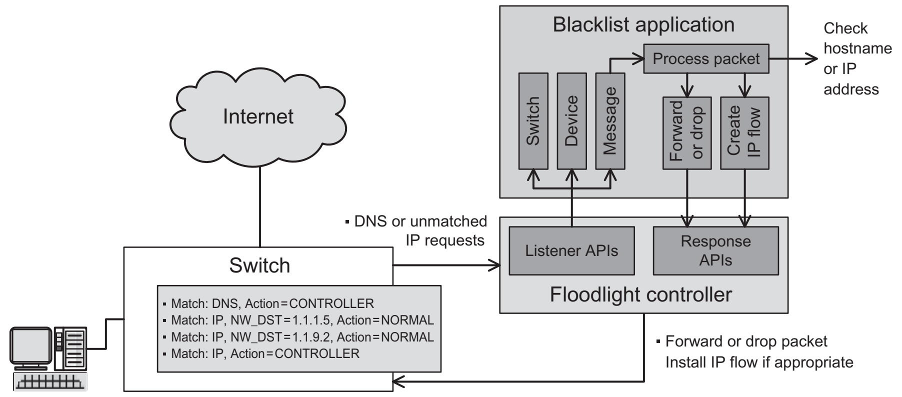

Blacklist application
Process packet
Check hostname or IP address
Switch
Device
Message
Forward or drop
Create IP flow
Internet
▪ DNS or unmatched IP requests
Listener APIs
Response APIs
Floodlight controller
Switch
▪ Match: DNS, Action=CONTROLLER
▪ Match: IP, NW_DST=1.1.1.5, Action=NORMAL
▪ Match: IP, NW_DST=1.1.9.2, Action=NORMAL
▪ Match: IP, Action=CONTROLLER
▪ Forward or drop packet
Install IP flow if appropriate

FIG. 12.4

Blacklist application design.

classes of listeners for reactive applications but as it has no special role in Blacklist we do not mention it further here. The *Message Listener* will forward incoming packets to the *Process Packet* module, which processes the packet and takes the appropriate action. Such actions include forwarding or dropping the packet and programming flow entries in the switch. We discuss the Message Listener and the Switch Listener in greater detail in Section 12.6.3.

The flow entries depicted in the switch in Fig. 12.4 show that DNS requests are always forwarded to the controller. If the destination IP address returned by DNS has already been checked, the application will program the appropriate high-priority flow entries. In our example, such entries are shown as the two flow entries matching 1.1.1.5 and 1.1.9.2. The reader should understand that the flows for destination 1.1.1.5 and 1.1.9.2 were programmed earlier by the controller when packets for these two destinations were first processed. If the IP address on the incoming packet has not been seen earlier or has aged out, then there will be no flow entry for that address and the IP request will be forwarded to the controller for blacklist processing.

Source code for the Listener, Packet Handler, and Flow Manager components is provided in Appendix B. Code snippets will also be provided throughout the following analysis. The reader should refer to the appendix to place these snippets in context. We include in the appendix the SDN-relevant source code for Blacklist in part to illustrate that a nontrivial network security application can be written in just 15 pages of Java, evidence of the relative ease and power of developing applications in the Open SDN model.

We now delve more deeply into the components of Blacklist.

12.6.3 BLACKLIST: LISTENERS

Our blacklist application uses two important listeners. The first is the *SwitchListener*. The *Switch-Listener* receives notifications from the controller whenever a switch is discovered or when it has disconnected. The *SwitchListener* also receives notifications when the switch's port configuration has changed (e.g., when ports have been added or removed). The exact frequency of such notifications depends on the particular switch's environment. Once steady-state operation has been achieved, however, it is unlikely that these notifications will occur more often than once every few seconds. The two most common SDN switch-type entities in this example are wired switches and APs.

When the application first learns that a switch has been discovered, the application programs it with two default rules, one to forward DNS requests to the controller and the other a low-priority flow entry to forward IP packets to the controller. For nonblacklisted IP addresses, this low-priority IP flow entry will be matched the first time that IP destination is seen. Subsequently, a higher priority flow is added that allows packets matching that description to be forwarded normally without involving the controller. If that flow entry idle timer expires, the flow entry is removed and the next packet sent to that IP address will once again be forwarded to the controller, starting this process anew.

Examination of the actual source code in Appendix B reveals that we add one other default flow for ARP requests, which are forwarded normally. ARP requests pose no threat from the perspective of the blacklist application. So, by default, all new switches will have three flows: DNS (send to controller), IP (send to controller), and ARP (forward NORMAL).

The other important listener is the *MessageListener*. The most important of the various methods in this listener is the *receive* method. The receive method is called whenever a packet arrives at the controller. The method then passes the packet to the blacklist SDN application. When the *PACKET_IN* message is received, our application invokes the *PacketHandler* to process the packet. In Section 5.3.5 we described how the switch uses the *PACKET_IN* message to communicate to the controller.

The code for the *MessageListener* module is listed in Appendix B.1. We examine part of that module below. First, we see that our class is derived from the Floodlight interface called *IOFMessageListener*:

```
//- - - - - - - - - - - - - - - - - - - - - - - - - - - -
public class MessageListener implements IOFMessageListener
//- - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

Next is the registration of our *MessageListener* method:

```
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
public void startUp()
{
    // Register class as MessageListener for PACKET_IN messages.
    mProvider.addOFMessageListener( OFType.PACKET_IN, this );
}
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

This code makes a call into Floodlight (*mProvider*), invoking the method *addOFMessageListener* and telling it we wish to listen for events of type *PACKET_IN*.

The next piece of code to consider is our *receive* method, which we provide to Floodlight to call when a packet is received by the listener. We are passed a reference to the switch, the message itself, and the Floodlight context reference.

```
//- - - - - - - - - - - - - - - - - - - - - - - - - - - -
@Override
public Command receive( final IOFSwitch         ofSwitch,
                        final OFMessage         msg,
                        final FloodlightContext context )
{
    switch( msg.getType() )
    {
    case PACKET_IN:  // Handle incoming packets here

       // Create packethandler object for receiving packet in
       PacketHandler ph = new PacketHandler( ofSwitch,
                                             msg, context);

       // Invoke processPacket() method of our packet handler
       // and return the value returned to us by processPacket
       return ph.processPacket();

    default: break;  // If not a PACKET_IN, just return

    }

    return Command.CONTINUE;
}
//- - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

In the code mentioned previously, we see a Java *switch* statement based on the message type. We are only interested in messages of type *PACKET_IN*. Upon receipt of that message, we create a *PacketHandler* object to process this incoming packet. We then invoke a method within that new object to process the packet.

12.6.4 BLACKLIST: PACKET HANDLERS

In our application there is one *PacketHandler* class which handles all types of incoming packets. Recall that our flow entries on the switches have been set up to forward DNS requests and unmatched IP packets to the controller. The *processPacket* method of the PacketHandler shown in Fig. 12.4 is called to handle these incoming packets, where it performs the following steps:

1. *IP packet*. Examine the destination IP address and, if it is on the blacklist, drop the packet. *Note that we do not attempt to explicitly program a DROP flow for this bad address. If another attempt is made to access this blacklisted address, the packet will be diverted to the controller again. This is a conscious design decision on our part, though there may be arguments that support the approach of programming blacklisted flows explicitly.*
2. *DNS request*. If the IP packet is a DNS request, examine the hostnames, and if any are on the blacklist, drop the packet.
3. *Forward packet*. If the packet has not been dropped, forward it.

4. *Create IP flow*. If it is not a DNS request, create a flow for this destination IP address. Note that there is no need to set a flow for the destination IP address of the DNS server because all DNS requests are caught by the higher-priority DNS flow and sent to the controller.

In our blacklist application, the *Create IP flow* step mentioned previously will actually invoke a method in the *FlowMgr* class which has a method for creating the Floodlight OpenFlow objects that are required to program flow entries on the switches.

The source code for the *PacketHandler* class can be found in Appendix B.2. We describe some of the more important portions of that code in the following paragraphs.

PacketHandler is a *Plain Old Java Object* (POJO), which means that it does not inherit from any Floodlight superclass. The object's constructor stores the objects passed to it, including: (1) references to the switch that sent the packet, (2) the message itself (the packet data), and (3) the floodlight context, which is necessary for telling Floodlight what to do with the packet once we have finished processing it.

The first piece of code is the beginning of the *processPacket* method, which was called by the *MessageListener*.

```
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
public Command processPacket()
{
    // First, get the OFMatch object from the incoming packet
    final OFMatch ofMatch = new OFMatch();
    ofMatch.loadFromPacket( mPacketIn.getPacketData(),
                            mPacketIn.getInPort()     );

//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

This is our first look at the *OFMatch* object provided by Floodlight, which derives from the same OpenFlow Java source code that can be found in Beacon and, thus, in OpenDaylight. In the code mentioned previously, we create an *OFMatch* object and then we load the object from the packet that we have received. This allows us to use our own *ofMatch* variable to examine the header portion of the packet we have just received.

In different parts of *processPacket* we look at various portions of the incoming packet header. The first example uses *ofMatch.getNetworkDestination()* to get the destination IP address, which is used to compare against our IP blacklist.

```
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
IPv4.toIPv4AddressBytes( ofMatch.getNetworkDestination() ) );
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

The next example determines whether this is a DNS request:

```
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
if( ofMatch.getNetworkProtocol()      == IPv4.PROTOCOL_UDP &&
    ofMatch.getTransportDestination() == DNS_QUERY_DEST_PORT )
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

Much of the other code in the module in Appendix B.2 deals with doing the work of the blacklist application. This includes checking the destination IP address, parsing the DNS request payload and checking those names as well. The most relevant SDN-related code is the creation of an action list to hand to the controller, which it will use to instruct the switch as to what to do with the packet that was received. In *forwardPacket()* we create the action list with the following code:

```
//- - - - - - - - - - - - - - - - - - - - - - - - - - - -
final List<OFAction> actions = new ArrayList<OFAction>();
actions.add( new OFActionOutput( outputPort ) );
//- - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

This code creates an empty list of actions, then adds one action to the list. That action is a Floodlight object called *OFActionOutput*, which instructs the switch to send the packet out the given port.

The *FlowManager* object is responsible for the actual interaction with Floodlight to perform both the sending of the packet out of the port as well as the possible creation of the new flow entry. We describe this object in the following section.

12.6.5 BLACKLIST: FLOW MANAGEMENT

The *FlowMgr* class deals with the details of the OpenFlow classes that are part of the Floodlight controller. These classes simplify the process of passing OpenFlow details to the controller when our application sends responses to the OpenFlow switches. These classes make it possible to use simpler Java class structures for communicating information to and from the controller. However, there is considerable detail that must be accounted for in these APIs, especially compared to their RESTful counterparts.

We now examine some of the more important Java objects that are used to convey information about flows to and from the controller:

- *IOFSwitch*. This interface represents an OpenFlow switch and, as such, it contains information about the switch, such as ports, port names, IP addresses, etc. The *IOFSwitch* object is passed to the *MessageListener* when a packet is received and it is used to invoke the *write* method when an OpenFlow packet is sent back to the switch in response to the request that was received.
- *OFMatch*. This object is part of the *PACKET_IN* message that is received from the controller that was originated by the switch. It is also used in the response that the application will send to the switch via the controller to establish a new flow entry. When used with the *PACKET_IN* message, the match fields represent the header fields of the incoming packet. When used with a reply to program a flow entry, they represent the fields to be matched and may include wildcards.
- *OFPacketIn and OFPacketOut*. These objects hold the actual packets received and sent, including payloads.
- *OFAction*. This object is used to send actions to the switch, such as NORMAL forwarding or FLOOD. The switch is passed a list of these actions so that multiple items such as *modify destination network address* and then NORMAL forwarding may be concatenated.

Code for the *FlowMgr* class can be found in Appendix B.3. We will walk the reader through some important logic from *FlowMgr* using the following code snippets. One of the methods in *FlowMgr* is *setDefaultFlows* which is called from the *SwitchListener* class whenever a new switch is discovered by Floodlight.

```
//- - - - - - - - - - - - - - - - - - - - - - - - -
public void setDefaultFlows(final IOFSwitch ofSwitch)
{
    // Set the intitial 'static' or 'proactive' flows
    setDnsQueryFlow( ofSwitch );
    setIpFlow( ofSwitch );
    setArpFlow( ofSwitch );
}
//- - - - - - - - - - - - - - - - - - - - - - - - -
```

As an example of these internal methods, consider the code to set the DNS default flow. In the following code, we see *OFMatch* and *OFActionOutput* that were introduced in Section 12.6.4. The *OFMatch* match object is set to match packets which are Ethernet, UDP, and destined for the DNS UDP port.

```
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
private void setDnsQueryFlow( final IOFSwitch ofSwitch )
{
    // Create match object to only match DNS requests
    OFMatch ofMatch = new OFMatch();
    ofMatch.setWildcards( allExclude( OFPFW_TP_DST,
                                      OFPFW_NW_PROTO,
                                      OFPFW_DL_TYPE ) )
               .setDataLayerType( Ethernet.TYPE_IPv4 )
               .setNetworkProtocol( IPv4.PROTOCOL_UDP )
               .setTransportDestination( DNS_QUERY_DEST_PORT );

    // Create output action to forward to controller.
    OFActionOutput ofAction  = new OFActionOutput(
                         OFPort.OFPP_CONTROLLER.getValue(),
                         (short) 65535                    );

    // Create our action list and add this action to it
    List<OFAction> ofActions = new ArrayList<OFAction>();
    ofActions.add(ofAction);

    sendFlowModMessage( ofSwitch,
                        OFFlowMod.OFPFC_ADD,
                        ofMatch,
                        ofActions,
                        PRIORITY_DNS_PACKETS,
                        NO_IDLE_TIMEOUT,
                        BUFFER_ID_NONE );
}
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

The flow we are setting instructs the switch to forward the packet to the controller by setting the output port to *OFPort.OFPP_CONTROLLER.getValue()*. Then we call the internal method *sendFlowModMessage*. The code for that method is shown as follows:

```
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
private void sendFlowModMessage( final IOFSwitch      ofSwitch,
                                 final short          command,
                                 final OFMatch        ofMatch,
                                 final List<OFAction> actions,
                                 final short          priority,
                                 final short          idleTimeout,
                                 final int            bufferId )
{
    // Get a flow modification message from factory.
    final OFFlowMod ofm = (OFFlowMod) mProvider
                                 .getOFMessageFactory()
                                 .getMessage(OFType.FLOW_MOD);

    // Set our new flow mod object with the values that have
    // been passed to us.
    ofm.setCommand( command ).setIdleTimeout( idleTimeout )
                             .setPriority( priority )
                             .setMatch( ofMatch.clone() )
                             .setBufferId( bufferId )
                             .setOutPort( OFPort.OFPP_NONE )
                             .setActions( actions )
                             .setXid( ofSwitch
                                .getNextTransactionId() );

    // Calculate the length of the request, and set it.
    int actionsLength = 0;
    for( final OFAction action : actions )
       { actionsLength += action.getLengthU(); }
    ofm.setLengthU(OFFlowMod.MINIMUM_LENGTH + actionsLength);

    // Now send out the flow mod message we have created.
    try
    {
        ofSwitch.write( ofm, null );
        ofSwitch.flush();
    }
    catch (final IOException e)
    {
        // Handle errors with the request
    }
}
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

Items to note in the code mentioned previously are as follows:

- The *OFFlowMod* object is created and used to hold the information regarding how to construct the flow, including items such as idle timeout, priority, match field, and actions.

- The length must still be calculated by this method. The length could be provided by the OpenFlow package, but as of this writing it is not.
- The switch is sent the flow modification message using the *write* method.

There is additional code available for the reader to examine in Appendix B.3. For example, consider the FlowMgr methods *sendPacketOut* and *createDataStreamFlow* for more examples of interacting with the Floodlight controller.

The next section looks at the two dominant controllers available today and examines the APIs that they make available to the SDN application developer.

DISCUSSION QUESTION

We have described an internal, reactive application, in some detail. Does the level of software development expertise required seem overwhelming or reasonable? What part of the blacklist application is the most difficult to understand, and why?

12.7 CONTROLLER CONSIDERATIONS

In the earlier days of SDN, controllers were created for open application development by researchers (e.g., Beacon at Stanford University), by enterprises (e.g., Ryu by NTT Communications), and by network vendors (e.g., XNC by Cisco, VAN SDN Controller by HP, Trema by NEC). As the SDN technology has evolved and matured, these controllers still exist but have less momentum. As evidenced by various open conferences on SDN such as the Open Networking Summit, the emphases for SDN application development have begun to focus on two controllers: ODL and ONOS. In this section we consider each of these controllers and their suitability for SDN application development.

12.7.1 OpenDaylight

With respect to the number of applications available on an SDN controller, ODL currently holds the lead over ONOS, as it has a longer history. As such, it has a head start on attracting implementation efforts by vendors (e.g., Cisco [9], Brocade [10]) and customers (e.g., AT&T [11], Fujitsu [12]) alike. From a development point of view, the main arguments for choosing ODL are as follows.

- *Multiprotocol*: You should consider ODL if your interests lie in creating SDN solutions based on protocols such as NETCONF, BGP, MPLS, or other existing protocols. ODL has the lead in supporting these protocols, and the key backers of ODL such as Cisco have a vested interest in creating SDN solutions that work with them. ODL also provides a platform for integrating your own protocol into the ODL environment, opening the door for applications written by customers to utilize ODL to control your network devices using your device-specific protocol. ONOS has projects working on existing protocol solutions, but is behind ODL in this area.
- *Minimal risk*: You should consider ODL if you are planning on creating SDN solutions which cause a minimum of disruption to the status quo of existing networks, as ODL helps pave the way for slower, less risky evolution toward an eventual SDN network.
- *Controller confidence*: ODL is a Linux Foundation project, is backed by a host of networking vendors, and the number of customers creating SDN solutions on ODL is growing, making it a

confident choice for the future. There is little risk that ODL is going to cease to exist, which is not true for many of the controllers cited in Section 12.4 that are no longer garnering investment of time and money. In addition, there is a very large community of developers of code, both for ODL itself and for applications running on the controller. This promotes added stability and helps guarantee the longevity of the controller.

ODL offers both RESTful APIs for creating external applications as well as the MD-SAL environment for creating internal applications. External, RESTful applications require limited software development expertise and are easy to develop. Internal, MD-SAL applications require significant Java software development expertise and present a fairly steep learning curve. Thus caution is called for before opting for the internal application model. The benefits of creating MD-SAL applications may, however, outweigh these considerations.

12.7.2 ONOS

ONOS is a relative newcomer, having first been released in late 2014, but it has gained traction with service providers such as AT&T [13] and NEMs like Ciena [14] and Fujitsu [15]. As such, ONOS warrants consideration by prospective SDN application developers.

- *OpenFlow*: An SDN network engineer should consider ONOS if his/her interests lie primarily in creating OpenFlow-based applications, as this is the focus of ONOS. ONOS is a project emanating from the research communities at Stanford and other universities, with more of a focus on OpenFlow-based experimentation and development.
- *Service provider*: ONOS is explicitly targeted at service providers, and has gone to lengths to recruit them to join the ONOS project.
- *Intents*: For the developer interested in creating applications which reside above a layer of intents-based abstraction, ONOS has been built with this type of API in mind and provides a rich environment for applications that operate at this level.

ONOS offers both RESTful APIs for creating external applications, as well as Java APIs for internal application development. Both the RESTful APIs and Java APIs follow the general models that have evolved alongside with the Beacon, Floodlight, and HP VAN SDN controllers, and their derivatives. SDN developers familiar with those APIs will find similarities with the ONOS APIs. However, ONOS additionally provides intents-based APIs, which are new to SDN in general, and will require some amount of adjustment to this new, albeit improved, model.

DISCUSSION QUESTION

We have discussed different types of SDN applications, and two significant controllers, ODL and ONOS. Which controller would be best suited for which type of application? Why might you be inclined to use one versus the other?

12.8 NETWORK DEVICE CONSIDERATIONS

One of the most important considerations when building an SDN application is the nature of the network devices that are to be controlled by the application. These considerations vary depending on your intended type of SDN application, as well as whether yours is an OpenFlow versus a non-OpenFlow-based strategy. We examine these in turn.

12.8.1 OpenFlow DEVICE CONSIDERATIONS

Creating applications for network devices using OpenFlow requires an understanding of the capabilities of the devices you wish to control. Some of the considerations in this area are:

- Do all the switches support OpenFlow? What version of OpenFlow do they claim to support? How much of that version do they in fact support?
- Is there a mix of OpenFlow and non-OpenFlow switches? If so, will some non-OpenFlow mechanism be used to provide the best simulation of an SDN environment? For example, do the non-OpenFlow switches support some type of SDN API which allows a controller to configure flows in the switch?
- For OpenFlow-supporting switches, are they hardware or software switches or both? If both, what differences in behavior and support will need to be considered?
- For hardware OpenFlow switches, what are their hardware flow-table sizes? If the switches are capable of handling flow table overflow in software, at what point are they forced to begin processing flows in software, and what is the performance impact of doing so? Is it even possible to process flows in software? What feature limitations are introduced because of the hardware?
- What is the mechanism by which the switch-controller communications channel is secured? Is this the responsibility of the network programmer?

Hardware switches vary in their flow capacity, feature limitations and performance, and these should be considerations when creating an SDN application. The major issue regarding flow capacity is how many flow entries the hardware ASIC is capable of holding. Some switches support less than 1000 flow entries, while some, such as the NEC PF5240 [16], support greater than 150,000. Note that for some switches the maximum number of flows depends on the nature of the flows supported. For example, the NEC PF5820 switch [17] can support 80,000 layer 2-only flows but only 750 full 12-tuple flows. (We described matching against the basic 12-tuple of input fields in Section 5.3.3.) A layer 2 flow entry only requires matching on the MAC address, so the amount of logic and memory occupied per flow entry is much less. Some applications, such as a traffic prioritization application or a TCP-port-specific application, may use relatively few flow table entries. Others, such as an access control application or a per-user application, will be much more flow-entry hungry. The topological placement of the network device may have a bearing on the maximum number of flow entries as well. For example, an access control application being applied in a switch or AP at the user edge of the network will require far fewer flow entries per network device than the same application controlling an upstream or core switch.

Hardware implementations often have feature limitations because, when the OpenFlow specification is mapped to the capabilities of the hardware, generally some features cannot be supported. For example, some ASICs are not able to perform NORMAL forwarding on the same packet that has had its header modified in some way. While it is true that the list of features supported by the network device can be learned via OpenFlow *Features Request* and *Features Reply* messages (see Section 5.3.5), this does not tell you the specific nuances and limitations that may be present in any one switch's implementation of each feature. Unfortunately, this must often be discovered via trial and error. Information about these issues may be available on the Internet sites of interested SDN developers. The application developer must be aware of these types of limitations before designing the solution.

Some hardware switches will implement some functions in hardware and others in software. It is wise to understand this distinction so that an application does not cause unexpected performance degradation by unknowingly manipulating flows such that the switch processes packets in software.

In general software implementations of OpenFlow are able to implement the full set of functionality defined by the OpenFlow specification for which they claim support. They implement flow tables in standard computer memory. Thus the feature limitations and table size issues described previously are not a major concern for application developers. However, performance will still be a consideration, since implementations in hardware will generally be faster.

12.8.2 NON-OpenFlow DEVICE CONSIDERATIONS

Creating applications for network devices using NETCONF or BGP-LS/PCE-P requires knowledge of the devices you wish to control. We discuss considerations for each of these protocols as follows.

NETCONF

The NETCONF protocol is standard and devices supporting it may be compatible. YANG also is a standard, and should be consistent among vendors and across product families. There are, however, potential pitfalls:

- *NETCONF without YANG*: There are devices which support an earlier version of NETCONF that does not support YANG models. These will likely be proprietary implementations and the data models for each must be understood in order to communicate successfully with each device. This is a particular challenge for service providers, as they tended to be early adopters of NETCONF and hence are forced to contend with pre-YANG versions of the protocol.
- *YANG models*: The critical issue regarding NETCONF/YANG control of network devices is the definition of the data models for individual functional areas such as routing, policy, and security. Today these models vary between vendors, and even vary among different products from a single vendor. It should be noted, however, that standardization efforts are underway in bodies such as the IETF for creating common YANG models for interfaces, routing, and other functional aspects of networking devices.

BGP-LS/PCE-P

The BGP-LS/PCE-P project in ODL includes plugins for BGP (which includes BGP-LS and BGP-FS) and PCE-P. OSPF or IS-IS topology information is received by the BGP plugin via the BGP-LS protocol. PCE-P is then employed to set the desired MPLS LSPs for routing traffic through the network. ODL supports other protocols as well. Some examples include the *Locator/ID Separation Protocol* (LISP), SNMP, and the *Interface to the Routing System* (I2RS).

We now will provide a number of high-level designs for some additional simple applications relevant to several different environments. These include the data center, the campus, and service provider environments. We restrict ourselves to high-level design issues in the following sections. It is our hope that the reader will be able to extend the detailed source code analysis performed for the blacklist application in Section 12.6 and apply those learnings to the cases we cover next.

12.9 CREATING NETWORK VIRTUALIZATION TUNNELS

As explained in Section 8.3, data center multitenancy, MAC-table overflow, and VLAN exhaustion are being addressed with new technology making use of tunnels, the most common being the MAC-in-IP tunnels provided by protocols such as VXLAN, NVGRE, and STT. Fig. 12.5 shows a possible software application design for creating and managing overlay tunnels in such an environment. We see in the figure the switch, device, and message listeners commonly associated with reactive applications. This reactive application runs on the controller and listens for IP packets destined for new IP destination addresses for which no flow entries exist. When the message listener receives such an unmatched packet, the application's tunnel manager is consulted to determine which endpoint switch is responsible for this destination host's IP address.

The information about the hosts and their tunnel endpoints is maintained by the application in the associated host and tunnel databases. The information itself can be gathered either outside of the domain of OpenFlow or it could be generated by observing source IP addresses at every switch. Whichever way this is done, the information needs to be available to the application.

When a request is processed, the tunnel information is retrieved from the database and, if the tunnel does not already exist, it will be created. Note that there may be other hosts communicating between these same two switches acting as tunnel endpoints, and so there may be a tunnel already in existence. The tunnel creation can be done in a number of ways, all of which are outside the scope of the OpenFlow specification. The tunnel creation is performed on the OpenFlow device and a mapping is established between that tunnel and a virtual port, which is how tunnels are represented in flow entries. These methods are often proprietary and specific to a particular vendor. For the reader interested in configuring VXLAN or NVGRE tunnels on the OVS switch, a useful guide can be found in [18].

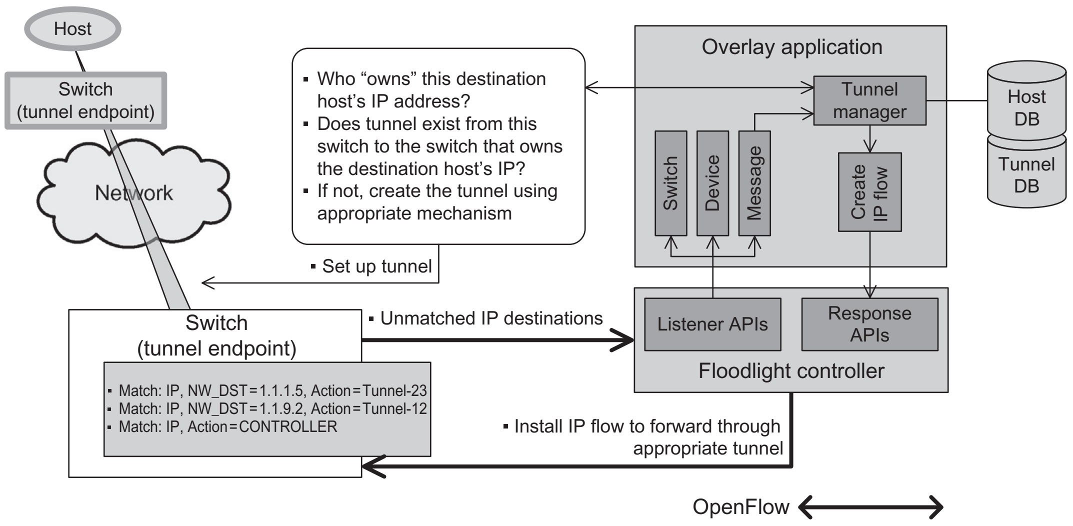

Host
Switch (tunnel endpoint)
Network
▪ Who "owns" this destination host's IP address?
▪ Does tunnel exist from this switch to the switch that owns the destination host's IP?
▪ If not, create the tunnel using appropriate mechanism
▪ Set up tunnel
Overlay application
Tunnel manager
Switch
Device
Message
Create IP flow
Host DB
Tunnel DB
Switch (tunnel endpoint)
▪ Unmatched IP destinations
Listener APIs
Response APIs
Floodlight controller
▪ Match: IP, NW_DST=1.1.1.5, Action=Tunnel-23
▪ Match: IP, NW_DST=1.1.9.2, Action=Tunnel-12
▪ Match: IP, Action=CONTROLLER
▪ Install IP flow to forward through appropriate tunnel
OpenFlow

FIG. 12.5

Tunnels application design.

Once the tunnel is either created or identified, the tunnel application can set up flows on the local switch to direct traffic heading toward that destination IP address into the appropriate tunnel. Once the virtual port exists, it is straightforward to create flow entries such as those shown in the figure where we see the tunneled packets directed to virtual ports TUNNEL-23 and TUNNEL-12. The complementary flow will need to be set on the switch hosting the remote tunnel endpoint as well.

This design attempts to minimize tunnel creation by only creating tunnels as needed, and removing them after they become idle. A different, proactive approach would be to create the tunnels a priori, obviating the need for *PACKET_IN* events.

Note that the preceding high-level design is unicast-centric. It does not explain the various mechanisms for dealing with broadcast packets such as ARPs. These could be dealt with by a global ARP service provided by the controller. The idea behind such a controller-based service is that the controller learns IP-MAC relationships by listening to ARP replies and, when it has already cached the answer to an ARP request, it does not forward the ARP request but instead replies automatically, thus suppressing ARPs from using up broadcast bandwidth. This process is fairly easy to embody in the SDN controller as it learns the IP and MAC addresses of unmatched packets which by default are forwarded to the controller. Thus the controller is often able to trap ARP requests and provide the reply without propagating the broadcast.

There are also ways of handling these requests via the VXLAN, NVGRE, and STT protocols. The methods of dealing with broadcasts and multicasts in the tunneling protocols are specific to each protocol and beyond the scope of this book. For details we refer the reader to the references we have previously provided for these three protocols.

12.10 OFFLOADING FLOWS IN THE DATA CENTER

One of the major problems in data centers is the elephant flows, which we introduced in Section 9.6.1. Such flows consume huge amounts of bandwidth, sometimes to the point of starving other flows of bandwidth. Fig. 12.6 depicts a high-level design of an offload application. This proactive application uses RESTful APIs to communicate with the controller. The general overview of the operation of such an application is as follows:

- There is a Flow Monitor, which monitors flows and detects the beginning of an elephant flow. (This is a nontrivial problem in itself. In Section 9.6.1 we provided simple examples as well as references for how elephant flows can be scheduled, predicted, or detected.)
- Once an elephant flow is detected, the Flow Monitor sends a notification message (possibly using the application's own REST API) to communicate details about the flow. This includes identifying the two endpoint ToR switches where the flow begins and ends as well as the IP addresses of the starting (1.1.1.5) and ending (2.2.2.10) hosts.
- The offload application has been previously configured with the knowledge of which ports on the ToR switches connect to the offload Switch.
- Armed with this topology information, the offload application is able to set flows appropriately on the ToR switches and the offload switch.
- As shown in Fig. 12.6 a new flow rule is added in this ToR switch. This is the flow shown in italics in the figure. It is important that this be a high-priority flow and match exactly on the source and

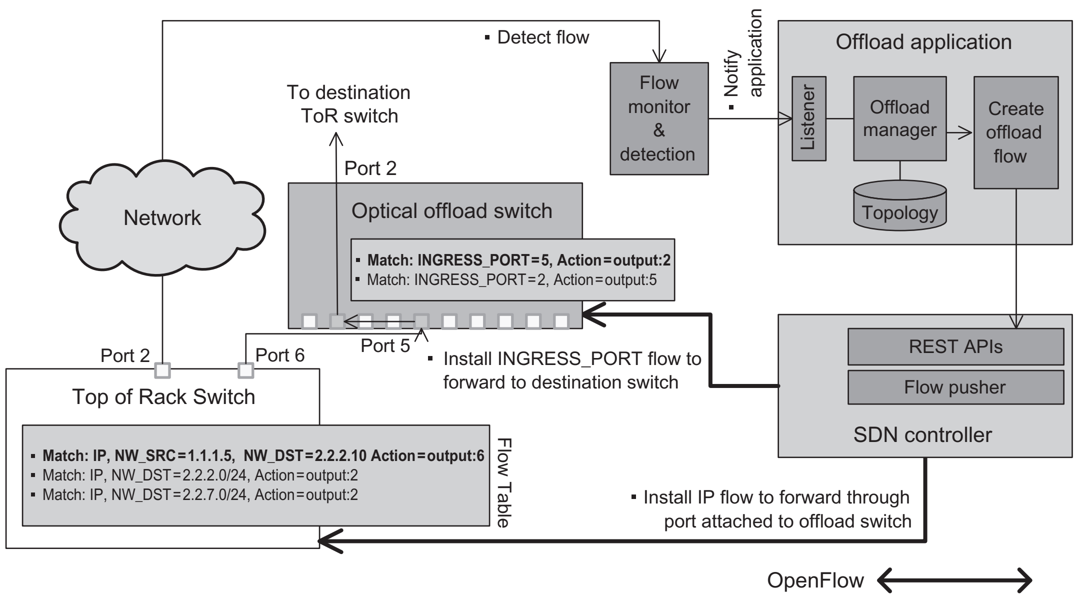

▪ Detect flow
Flow monitor & detection
▪ Notify application
Offload application
Listener
Offload manager
Create offload flow
Topology
To destination ToR switch
Port 2
Network
Optical offload switch
▪ Match: INGRESS_PORT=5, Action=output:2
▪ Match: INGRESS_PORT=2, Action=output:5
Port 2
Port 6
Port 5
▪ Install INGRESS_PORT flow to forward to destination switch
REST APIs
Flow pusher
SDN controller
Top of Rack Switch
▪ Match: IP, NW_SRC=1.1.1.5, NW_DST=2.2.2.10 Action=output:6
▪ Match: IP, NW_DST=2.2.2.0/24, Action=output:2
▪ Match: IP, NW_DST=2.2.7.0/24, Action=output:2
Flow Table
▪ Install IP flow to forward through port attached to offload switch
OpenFlow

FIG. 12.6

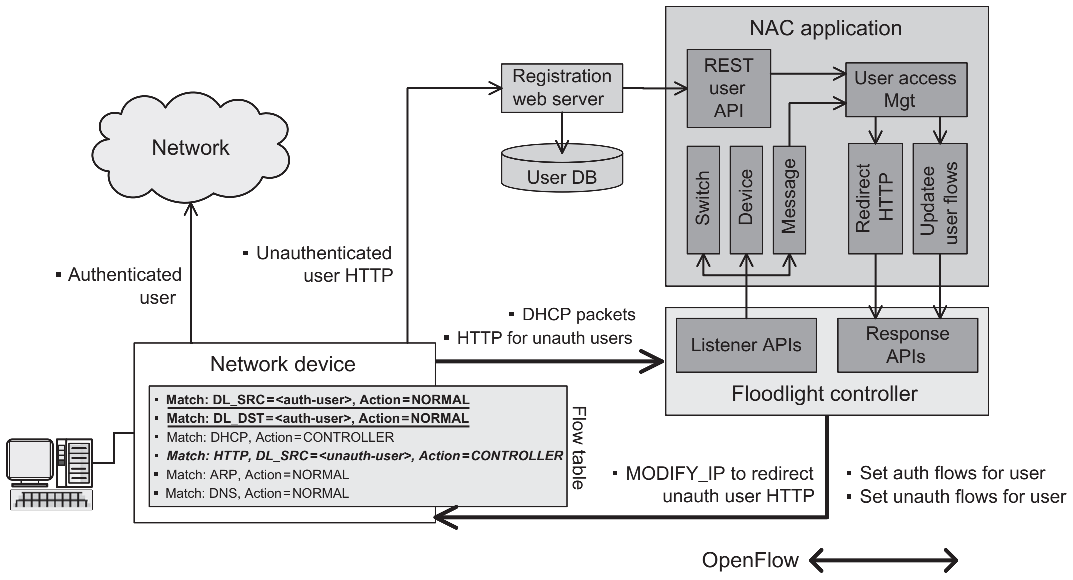
Offload application design.

destination host addresses. The next lower-priority flow entry matches on the *network* address 2.2.2.0/24, but, as that is a lower-priority entry, packets belonging to the elephant flow will match the higher priority entry first and those packets will be directed out port 6 to the optical offload switch. Note that traffic other than the elephant flow will match the lower-priority entries and will be transmitted via the normal port.

- On the optical offload switch, flow rules are programmed which connect the two ports which will be used to forward this elephant flow between the switches involved. For the sake of simplicity, we will assume that an elephant flow is unidirectional. If there is bulk traffic in both directions, the steps we describe here can be duplicated to instantiate a separate OTN flow in each direction.
- Once the elephant flow ceases, the offload application will be informed by the flow monitor and the special flows will be removed.

12.11 ACCESS CONTROL FOR THE CAMPUS

In Section 9.3.1 we introduced the idea of an SDN *network access control* (NAC) application for the campus. Fig. 12.7 shows a high-level design of a campus NAC application. It should be apparent from the diagram that this application is a reactive application because it needs to be notified of unauthenticated users attaching to the network. The initial state of the switch is to have flows which allow ARP and DNS, but which forward DHCP requests and responses to the controller. The DHCP responses will be used to build a database of MAC and IP addresses for the end user nodes attaching to the network.



- **Match: DL_SRC=<auth-user>, Action=NORMAL**
- **Match: DL_DST=<auth-user>, Action=NORMAL**
- Match: DHCP, Action=CONTROLLER
- ***Match: HTTP, DL_SRC=<unauth-user>, Action=CONTROLLER***
- Match: ARP, Action=NORMAL
- Match: DNS, Action=NORMAL

FIG. 12.7

NAC application design.

When the DHCP response is observed, if this is a new user, the NAC application adds the user to the database and the user's state is *unauthenticated*. The application then programs a flow entry in the switch based on the user's MAC address. This flow in the nonunderlined italics text in Fig. 12.7 is programmed to forward the user's HTTP requests to the controller. Thus, when a user opens a browser, in typical captive portal behavior, the application will cause the HTTP request to be forwarded to the captive portal web server where user authentication takes place. This is easily achieved by modifying the destination IP address (and possibly the destination MAC address as well) of the request so that when the switch forwards the packet it will go to the captive portal *registration web server* shown in Fig. 12.7 rather than to the intended destination.

Normal captive portal behavior is to force an HTTP redirect which sends the user's browser to the desired captive portal registration, authentication or guest login page. The controller will have programmed the switch with flows which will shunt HTTP traffic to the captive portal. No other traffic type needs to be forwarded as the authentication takes place via the user's web browser. Thus the user engages in an authentication exchange with the captive portal. This could entail payment, registration or some related mechanisms. When complete, the captive portal informs the NAC application that the user is authenticated. Note that the registration server uses the application's RESTful APIs to notify the NAC application of changes in the user's status. At this point the application removes the nonunderlined italics flow entry shown in Fig. 12.7 and replaces it with the two bold-underlined flow entries, thus allowing the user unfettered access to the network. When the user later connects to the network, if he is still registered, then he will be allowed into the network without needing to go through the registration/authentication step, unless these flow entries have timed out.

The reader should note that in this example and in the accompanying Fig. 12.7 we stop part-way through the complicated process of HTTP redirection. The full process entails sending an HTTP 302 message back to the user in order to redirect the user's browser to the registration server. These details are not germane to our example so we exclude them here.

12.12 TRAFFIC ENGINEERING FOR SERVICE PROVIDERS

In Section 9.2.1 we explained how traffic engineering is a key tool for service providers to optimize their return on investment in the large networks they operate for their customers. Fig. 12.8 shows a simplified design for a proactive traffic engineering application that takes traffic monitoring information into account. As can be seen in the figure, the flow rules in the router(s) are very simple. The design of the application likewise is simple. The complexity resides inside the path computation manager component doing the path calculations and optimizing traffic over the available links.

This problem can also be addressed using reactive application design. Fig. 12.9 reveals a high-level design of such an approach. The key difference between this design and the proactive one lies in the flow tables on the routers which feature a final flow entry that forwards unmatched packets to the controller. In this solution, initially there are no flows on the switch except for the CONTROLLER flow. The routers are not preconfigured with any specific flows other than to forward unmatched packets to the controller, as is typical for a reactive application. When these packets are received at the controller, the application's listener passes them to the path control manager. The path control manager then calculates the best path to the destination given the current traffic loads. Thus the flows are established in an on-demand manner. When traffic to that destination eventually stops, the flow is

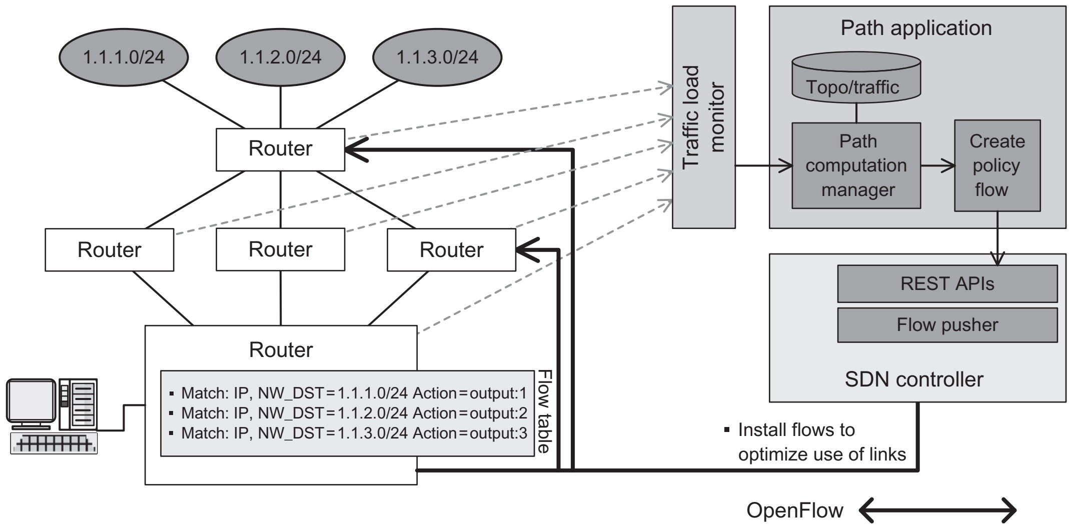


FIG. 12.8

Traffic engineering proactive application design.

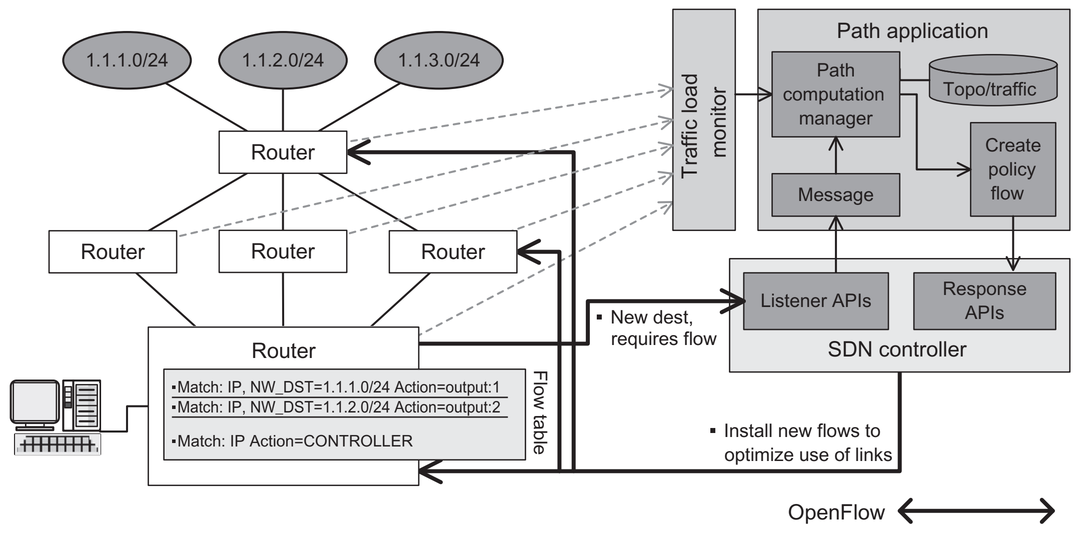


FIG. 12.9

Traffic engineering reactive application design.

removed. The flow could be removed implicitly by aging out or an external traffic monitoring function could explicitly inform the application that the flow has stopped and then the flow could be deleted by the application.

DISCUSSION QUESTION

Of the SDN application designs we have presented, which do you feel is the most innovative? Which would be the most pragmatic and easy to implement? Can you think of other designs that would be interesting to discuss? Can you think of any current networking problems that demand an SDN solution such as these?

12.13 CONCLUSION

In this chapter we bridge the OpenFlow specifics of Chapter 5 with the conceptual use cases of Chapters 8 and 9, grounded with concrete design and programming examples. Our examples are all based on applications of Open SDN. Certainly, network programmers familiar with SDN via APIs or SDN via Overlays would be able to address with their respective technologies some of the scenarios presented in this chapter. It is unlikely, though, in our opinion, that those technologies would be so easily and simply adapted to the full spectrum of applications presented in this chapter. Having said that, though, the viability of a technology depends not only on its extensibility from a programmer's perspective, but on other factors as well. Such issues include: (1) the cost, sophistication, and availability of ready-made solutions, (2) the market forces that push and pull on various alternatives, and (3) how rapidly needed future innovations are likely to emerge from different alternatives. We will spend the final three chapters of this book addressing these three aspects of SDN.

REFERENCES

[1] OpenFlowJ. Retrieved from: http://bitbucket.org/openflowj/openflowj.
[2] Apache Karaf. Retrieved from: http://karaf.apache.org.
[3] Apache Maven. Retrieved from: http://maven.apache.org.
[4] Maven archetypes. Retrieved from: http://maven.apache.org/archetype/.
[5] OpenDaylight Controller: MD-SAL. Retrieved from: https://wiki.opendaylight.org/view/OpenDaylight_Controller:MD-SAL.
[6] Controller core functionality tutorials. Retrieved from: https://wiki.opendaylight.org/view/Controller_Core_Functionality_Tutorials:Tutorials:Starting_A_Project.
[7] Project Floodlight Download. Project Floodlight. Retrieved from: http://www.projectfloodlight.org/download/.
[8] Erikson D. Beacon. OpenFlow @ Stanford; February 2013. Retrieved from: https://openflow.stanford.edu/display/Beacon/Home.
[9] Cisco Open SDN controller. Retrieved from: http://www.cisco.com/c/en/us/products/cloud-systems-management/open-sdn-controller/index.html.
[10] Brocade SDN controller. Retrieved from: http://www.brocade.com/en/products-services/software-networking/sdn-controllers-applications/sdn-controller.html.
[11] How AT&T is using OpenDaylight. Retrieved from: https://www.opendaylight.org/news/user-story/2015/05/how-att-using-opendaylight.
[12] Fujitsu Virtuora NC. Retrieved from: http://www.fujitsu.com/us/products/network/technologies/software-defined-networking-and-network-functions-virtualization/.
[13] ONOS and AT&T team up to deliver CORD. SDX Central. Retrieved from: https://www.sdxcentral.com/articles/news/cord-onos-att/2015/06/.
[14] ONOS framework builds out SDN ecosystem. Retrieved from: http://www.ciena.com/connect/blog/ONOS-framework-builds-out-SDN-ecosystem.html.
[15] Fujitsu successfully demonstrates ONOS interoperability. Retrieved from: http://www.fujitsu.com/us/about/resources/news/press-releases/2015/fnc-20150615.html.
[16] ProgrammableFlow PF5240 Switch. NEC datasheet; Retrieved from: http://www.necam.com/SDN/.
[17] ProgrammableFlow PF5820 Switch. NEC datasheet; Retrieved from: http://www.necam.com/SDN/.
[18] Salisbury B. Configuring VXLAN and GRE tunnels on OpenvSwitch. NetworkStatic; 2012. Retrieved from: http://networkstatic.net/configuring-vxlan-and-gre-tunnels-on-openvswitch/.