CHAPTER

SDN Applications

10

We now descend from the conceptual discussions about SDN use cases and take a deeper dive into how SDN applications are actually implemented. This chapter focuses on SDN application development in an Open SDN environment. Most SDN-via-overlay offerings today are bundled solutions that come ready-made with a full set of functionality, generally without the ability to add applications to those solutions. SDN-via-API solutions in general allow for some degree of user-level programming but do not provide the network-wide view and abstraction layer afforded by the Open SDN controller. OpenDaylight is a platform that forms the basis for hybrid application development, allowing applications to control non-OpenFlow-enabled switches as well as OpenFlow-enabled switches. Many detailed functional block diagrams are included in this chapter, as is a fair amount of sample source code that serves as helpful examples for the network programmer.

10.1 Before You Begin

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

- *What is the basic nature of the application?* Will it spend most of its time reacting to incoming packets that have been forwarded to it from the switches in the network? Or will it typically set flows proactively on the switches and only respond to changes in the network that require some type of reconfiguration?
- *What is the type and nature of network with which the application will be dealing?* For example, a campus network application geared toward access control will have a different design than a campus application that is concerned with general network policies such as the prioritization of different types of traffic.
- *What is the intended deployment of the controller with respect to the switches it controls?* A controller that is physically removed from the switches it controls (e.g., in the cloud) will clearly not be able to efficiently support an SDN application that is strongly reactive in nature because of the latency involved in forwarding packets to the controller and receiving a response.
- *Are the SDN needs purely related to the data center and virtualization?* If so, perhaps an SDN overlay solution is sufficient to meet current needs, and future needs related to the physical underlay network can be met later, as OpenFlow-related technologies mature.

Software Defined Networks. http://dx.doi.org/10.1016/B978-0-12-416675-2.00010-3
© 2014 Elsevier Inc. All rights reserved.

- *Will the application run in a greenfield environment, or will it need to deal with legacy switches that do not support SDN?* Only in rare situations is one able to build an SDN application from the ground up. Typically, it is required to use existing equipment in the solution. It is important to understand whether the existing switches can be upgraded to have OpenFlow capability and whether or not they will have hybrid capability, supporting both OpenFlow and legacy operation. If the switches do not support OpenFlow, it is important to understand the level of SDN-via-APIs support that is available.

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

10.2 Reactive versus Proactive Applications

Typically, in the case of reactive SDN applications, the communication between the switch and the controller will scale with the number of new flows injected into the network. The switches may often have relatively few flow entries in their flow tables, since the flow timers are set to match the expected duration of each flow in order to keep the flow table size small. This results in the switch frequently receiving packets that match no rules. In the reactive model, those packets are encapsulated in PACKET_IN messages and forwarded to the controller and thence to an application. The application inspects the packet and determines its disposition. The outcome is usually to program a new flow entry in the switch(es) so that the next time this type of packet arrives, it can be handled locally by the switch itself. The application will often program multiple switches at the same time so that each switch along the path of the flow will have a consistent set of flow entries for that flow. The original packet will often be returned to the switch so that the switch can handle it via the newly installed flow entry. The kind of applications that naturally lend themselves to the reactive model are those that need to see and respond to new flows being created. Examples of such applications include per-user firewalling and security applications that need to identify and process new flows.

On the other hand, proactive SDN applications feature less communication emanating from the switch to the controller. The proactive SDN application sets up the switches in the network with the flow entries that are appropriate to deal with incoming traffic before it arrives at the switch. Events that trigger changes to the flow table configurations of switches may come from mechanisms that are outside the scope of the switch-to-controller secure channel. For example, some external traffic-monitoring module will generate an event which is received by the SDN application, which will then adjust the flows on the switches appropriately. Applications that naturally lend themselves to the proactive model usually need to manage or control the topology of the network. Examples of such applications include new alternatives to spanning tree and multipath forwarding.

Reactive programming may be more vulnerable to service disruption if connectivity to the controller is lost. Loss of the controller will have less impact in the proactive model if the failure mode specifies that operation should continue. (If the failure mode configuration is set to *fail closed*, then loss of controller connectivity will cause the flow tables to be flushed independently of the style of the application.) With the proactive model, there is no additional latency for flow setup, because they are prepopulated. A drawback of the proactive model is that most flows will be wildcard-style, implying aggregated flows and thus less fine granularity of control.

We discuss these issues in more detail in the next subsections.

10.2.1 Terminology

To maintain consistency with the terminology used in the software libraries referenced in this section, the term *device* always refers to an end-user node, and the term *switch* always refers to a networking node, such as a wireless access point, switch, or router. In general in this book, the term *SDN device* includes SDN switches. In the vernacular of the particular Java APIs discussed in this section, *device* refers to the end-user node. Since we want to name the APIs faithfully to the actual source code implementation, we adhere to this convention for the term *device only* in Sections 10.2, 10.3, and 10.4.

10.2.2 Reactive SDN Applications

Reactive applications need to be asynchronously notified of incoming packets that have been forwarded to the controller from OpenFlow switches. In Section 6.2.3 we explained why RESTful APIs have become a popular method for manipulating SDN networks. The type of asynchronous notification required here is not performed via RESTful APIs, however. This is because a RESTful API is asymmetric, with a requester (the application) and a responder (the controller). Under the RESTful paradigm, it is not straightforward for the controller (responder) to asynchronously notify the application of an event such as the arrival of a packet from a switch. To accomplish this sort of asynchronous notification via a RESTful interface an additional, reverse-direction control relationship would need to be established, with the controller being the requester and the application being the responder. Though this is feasible, there is additional latency with this approach compared to the *listener* approach described below. With the Java APIs one can register a listener and then receive callbacks from the controller when packets arrive. These callbacks come with passed arguments including the packet and associated metadata, such as which switch forwarded the packet and the port on which the packet arrived. Consequently, reactive applications tend to be written in the native language of the controller, which, in the examples in this chapter, is Java. Note that C and Ruby controllers are also available, and the same designs and mechanisms described here apply to those languages as well.

Reactive applications have the ability to register listeners, which are able to receive notifications from the controller when certain events occur. Some important listeners available in the popular Floodlight and Beacon controller packages are:

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

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

When a reactive SDN application is informed of an event, such as a packet that has been forwarded to the controller, a change of port state, or the entrance of a new switch or device into the network, the application has a chance to take some type of *action*. The most frequent event coming into the

application would normally be a packet arriving at the controller from a switch, resulting in an action. Such actions include:

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

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

Figure 10.1 shows the general design of a reactive application. Notice that the controller has a listener interface that allows the application to provide listeners for switch, device, and message (incoming packet) events. Typically, a reactive application will have a module to handle packets incoming to the controller that have been forwarded through the message listener. This packet processing can then act on the packet. Typical actions include returning the request to the switch, telling it what to do with the packet (e.g., forward out a specific port, forward NORMAL or drop the packet). Other actions taken by the application can involve setting flows on the switch in response to the received packet, which will inform the switch what to do the next time it sees a packet of this nature.

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

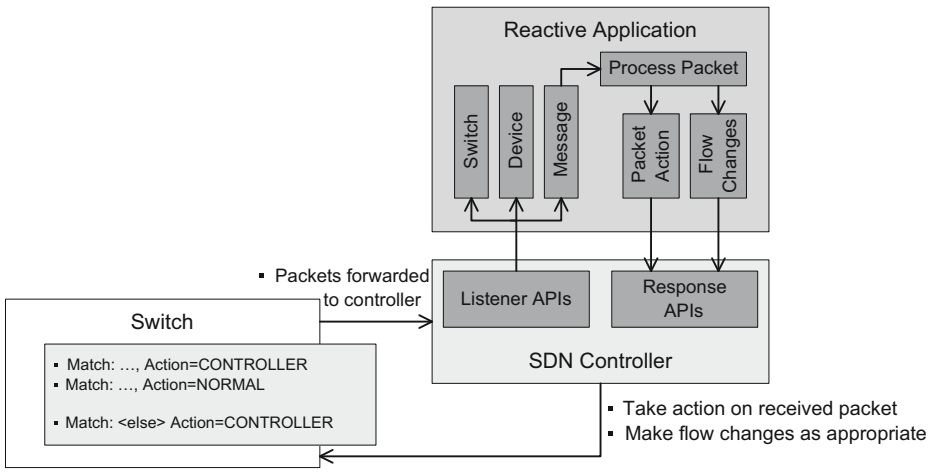


FIGURE 10.1

Reactive application design.

In the reactive model, the flow tables tend to continually evolve based on the packets being processed by the switch and by flows aging out. Performance considerations arise if the flows need to be reprogrammed too frequently, so care must be taken to appropriately set the idle timeouts for the flows. The point of an idle timer is to clear out flow entries after they have terminated or gone inactive, so it is important to consider the nature of the flow and what constitutes inactivity when setting the timeout. Unduly short idle timeouts will result in too many packets sent to the controller. Excessively long timeouts can result in flow table overflow. Knowing an appropriate value to set a timeout requires familiarity with the application driving the flow. As a rough rule of thumb, though, timeouts would be set on the order of tens of seconds rather than milliseconds or minutes.

10.2.3 Proactive SDN Applications

Proactive applications can be implemented using either the native API (e.g., Java) or using RESTful APIs. The reader should note that such RESTful APIs are not the ones resident in the switch that we described as part of SDN via APIs in Section 6.2.3. The RESTful APIs we describe here reside on the Open SDN controller and are called by the application. In contrast to reactive APIs, which are low-level and deal almost directly with the OpenFlow protocol below them, these controller RESTful APIs can operate at different levels, depending on the northbound API being used. They can be low-level, basically allowing the application developer to configure flow entries on the switches themselves. These RESTful APIs can also be high-level, providing a level of abstraction above the network switches so that the SDN application interacts with network components such as virtual networks rather than with the switches themselves.

Another difference between this type of application and a reactive one is that using RESTful APIs means there is no need for *listening* functionality. In the reactive application the controller can asynchronously invoke methods in the SDN application stimulated by events such as incoming packets and switches that have been discovered. This type of asynchronous notification is not possible with a unidirectional RESTful API, which is invoked on the controller by the application rather than the other way around. It is conceivable to implement an asynchronous notification mechanism from the controller toward a proactive application, but generally, proactive applications are stimulated by sources external to the OpenFlow control path. The traffic monitor described in Section 10.14 is an example of such an external source.

Figure 10.2 shows the general design of a purely proactive application. In such an application no listeners are receiving from the controller events about switches, devices, and messages. Such listeners are not a natural fit with a one-way RESTful API, wherein the application makes periodic calls to the API and the API has no means of initiating communication back to the application. As shown in Figure 10.2, proactive applications rely on stimulus from external *network events*. Such stimuli may originate from a number of sources, such as traffic or switch monitors using SNMP or SNORT, or external applications like a server virtualization service, which notifies the application of the movement of a virtual machine from one physical server to another. As we explained in Section 10.2, applications that deal with the network topology will be passed control protocol packets, such as DNS, DHCP, and BGP, by rules programmed to do just that.

The RESTful APIs are still able to *retrieve* data about the network, such as domains, subnets, switches, and hosts. RESTful APIs generally also provide an interface, often referred to as a *flow pusher*, which allows the application to set flows on switches. Thus the proactive application has the

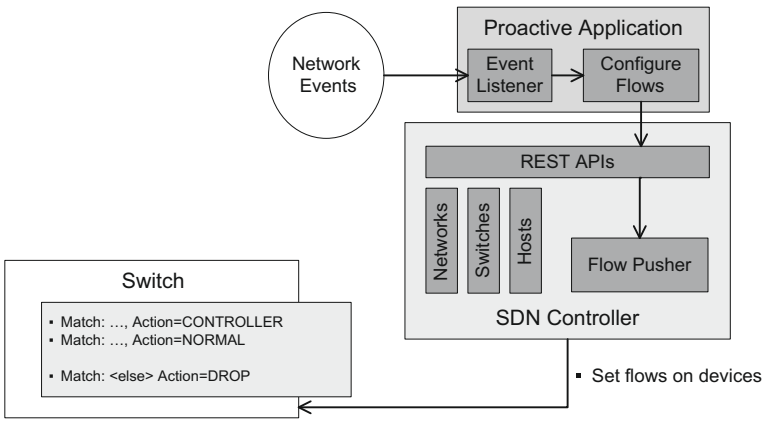


FIGURE 10.2

Proactive application design.

same ability to program flows as the reactive application but will tend to use wildcard matches more frequently.

As shown in Figure 10.2, the last flow entry will typically be to DROP unmatched packets. This is because proactive applications attempt to *anticipate* all traffic and program flow entries accordingly. As a consequence, packets that do not match the configured set of rules are discarded. We remind the reader that the match criteria for flow entries can be programmed such that most arriving packet types are expected and match some entry before this final DROP entry. If this were not the case, the purely proactive model could become an expensive exercise in dropping packets!

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

10.3 Analyzing Simple SDN Applications

Proactive, RESTful-based applications can run either on the same system as the controller or on a remote system, since HTTP allows that flexibility. In contrast, reactive applications are tightly bound to the controller as they access the controller via the native APIs (e.g., Java or C), and they run in the same container or execution space. In Section 10.12 we provide an example of proactive applications that use the RESTful *flow pusher* APIs. First, though, in Section 10.4 we look in detail at the source code

involved in writing a reactive SDN application. We use this example as an opportunity to walk through SDN application source code in detail and explore how to implement the tight application-controller coupling that is characteristic of reactive applications.

The application examples we provide in this chapter use the Floodlight controller. We have chosen Floodlight because it is one of the most popular open source SDN controllers. It is possible to download the source code [1] and examine the details of the controller itself. Floodlight also comes with a number of sample applications, such as a learning switch and a load balancer, which provide excellent additional examples for the interested reader. The core modules of the Floodlight controller come from the seminal Beacon controller [2]. These components are denoted as *org.openflow* packages within the Floodlight source code. Since Beacon is the basis of the OpenFlow functionality present in OpenDaylight and other SDN controllers, the OpenFlow-related controller interaction described in the following sections would apply to those Beacon-derived controllers as well. We provide further description of the Beacon and Floodlight controllers in Sections 10.5 and 10.6.

10.4 A Simple Reactive Java Application

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

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

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

10.4.1 Blacklisting Hostnames

We introduced an SDN solution to the blacklist problem in Section 8.3.3. We provide further details of that solution here. The first part of the Blacklist application deals with hostnames. Checking for malicious or undesirable hostnames works as follows:

- A single default flow is set on the edge switches to forward all DNS traffic to the OpenFlow controller. (Note that this is an example of a proactive rule. Thus, though a blacklist is primarily a reactive application, it does exhibit the hybrid reactive-proactive characteristic discussed in Section 10.2.3.)
- The Blacklist application running on the controller listens for incoming DNS requests that have been forwarded to the controller.

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

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

10.4.2 Blacklisting IP Addresses

As described earlier in Section 8.3.3, websites sometimes have embedded IP addresses, or end users will attempt to access undesirable locations by providing IP-specific addresses directly rather than a hostname, thus circumventing a DNS lookup. Our simple Blacklist application works as follows:

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

Note that a brute-force approach to the blacklist problem would be to explicitly program rules for all known bad IP addresses such that any packet matching those addresses is directly dropped by the switch. Though this solution works in principle, it is not realistic, because the number of blacklisted IP addresses is potentially very large and, as we have pointed out, the number of available flow entries, though varying from switch to switch, is limited.

There are three major components to this application:

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

Figure 10.3 provides a more detailed view of the Blacklist application introduced in Section 8.3.3. As a reactive application, blacklist first programs the switch to forward unmatched packets to the controller. The figure depicts three listeners for network events: the switch listener, the device listener, and the message listener. We show the device listener because it is one of the three basic classes of listeners for reactive applications, but since it has no special role in Blacklist, we do not mention it further here. The *message listener* will forward incoming packets to the *process packet* module, which processes

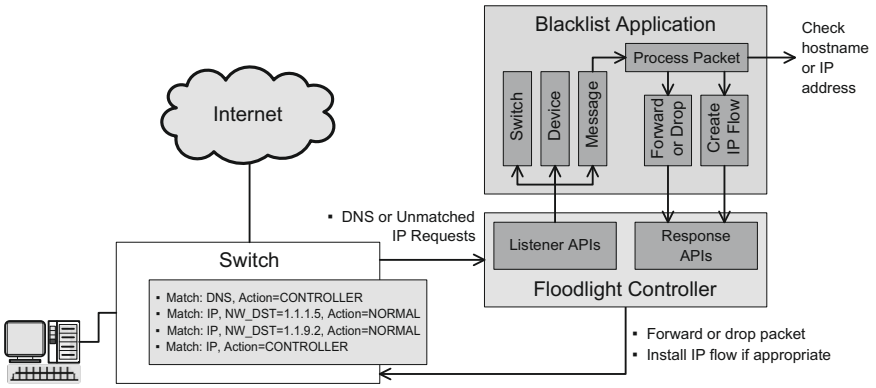


FIGURE 10.3

Blacklist application design.

the packet and takes the appropriate action. Such actions include forwarding or dropping the packet and programming flow entries in the switch. We discuss the message listener and the switch listener in greater detail in Section 10.4.3.

The flow entries depicted in the switch in Figure 10.3 show that DNS requests are always forwarded to the controller. If the destination IP address returned by DNS has already been checked, the application will program the appropriate high-priority flow entries. In our example, such entries are shown as the two flow entries matching 1.1.1.5 and 1.1.9.2. The reader should understand that the flows for destination 1.1.1.5 and 1.1.9.2 were programmed earlier by the controller when packets for these two destinations were first processed. If the IP address on the incoming packet has not been seen earlier or has aged out, there will be no flow entry for that address and the IP request will be forwarded to the controller for blacklist processing.

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

We now delve more deeply into the components of Blacklist.

10.4.3 Blacklist: Listeners

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

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

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

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

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

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

Next is the registration of our *MessageListener* method:

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

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

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

```
//-------------------------------------------------------------
@Override
public Command receive( final IOFSwitch         ofSwitch,
                        final OFMessage         msg,
                        final FloodlightContext context )
{
    switch( msg.getType() )
    {
```

```
        case PACKET_IN:  // Handle incoming packets here

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

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

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

        }

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

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

10.4.4 Blacklist: Packet Handlers

In our application there is one *PacketHandler* class that handles all types of incoming packets. Recall that our flow entries on the switches have been set up to forward DNS requests and unmatched IP packets to the controller. The *processPacket* method of the *PacketHandler* shown in Figure 10.3 is called to handle these incoming packets, where it performs the following steps:

1. **IP packet.** Examine the destination IP address and, if it is on the blacklist, drop the packet. *Note that we do not attempt to explicitly program a DROP flow for this bad address. If another attempt is made to access this blacklisted address, the packet will be diverted to the controller again. This is a conscious design decision on our part, though there may be arguments that support the approach of programming blacklisted flows explicitly.*
2. **DNS request.** If the IP packet is a DNS request, examine the hostnames, and if any are on the blacklist, drop the packet.
3. **Forward packet.** If the packet has not been dropped, forward it.
4. **Create IP flow.** If it is not a DNS request, create a flow for this destination IP address. Note that there is no need to set a flow for the destination IP address of the DNS server, because all DNS requests are caught by the higher-priority DNS flow and sent to the controller.

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

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

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

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

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

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

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

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

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

The next example determines whether this is a DNS request:

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

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

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

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

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

10.4.5 Blacklist: Flow Management

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

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

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

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

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

As an example of these internal methods, consider the code to set the DNS default flow. In the following code, we see *OFMatch* and *OFActionOutput*, which were introduced in Section 10.4.4. The *OFMatch* match object is set to match packets that are Ethernet, UDP, and destined for the DNS UDP port.

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

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

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

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

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

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

```
                                    .getOFMessageFactory()
                                    .getMessage(OFType.FLOW_MOD);

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

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

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

Items to note in the code above are:

- The *OFFlowMod* object is created and used to hold the information regarding how to construct the flow, including items such as idle timeout, priority, match field, and actions.
- The length must still be calculated by this method. The length could be provided by the OpenFlow package, but as of this writing it is not.
- The switch is sent the flow modification message using the *write* method.

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

The next section looks at some of the open source and commercial controllers that are available today and examines the APIs that they make available to the SDN application developer.

10.5 Background on Controllers

The Beacon controller is truly a seminal controller in that much of the basic OpenFlow controller code in Floodlight and OpenDaylight was derived directly from Beacon. As we pointed out in Section 9.1.1, Beacon itself was based on early work performed at Stanford by David Erickson and Rob Sherwood. Both Beacon and Floodlight are based on OpenFlowJ for the core Java OpenFlow implementation. Floodlight is maintained by engineers from Big Switch. The distributions of Beacon and Floodlight are now distinct and different applications are packaged with each, but Beacon remains close enough to Floodlight that we will not elaborate on it further at the detailed implementation level presented here.

We should point out that a number of other OpenFlow controllers are available. For example, the NOX controller has been used in a considerable number of research projects. We provide basic information about a number of open source controller alternatives in Section 11.6. The subset of controllers that we present in this chapter represents those controllers that we feel an application developer today would be most likely to use.

It is also important to note that as of this writing there is no standard for the northbound API to the controller. The Java APIs that we have documented in the preceding sections may not be exposed to the application programmers in every one of the controllers that we present in this chapter, including those based on the original Beacon code. There may be a higher-level abstraction that is exposed that is not so tightly coupled to the OpenFlow protocol. This appears to be the direction in which OpenDaylight may be headed. The application API is an interface that may translate to a number of different southbound APIs, OpenFlow being just one of them.

10.6 Using the Floodlight Controller

Floodlight is arguably the most prevalent open source SDN controller available today. It has a number of packages denoted by *org.openflow*, which make up some of the core functionality. There are also many packages from Floodlight that implement components such as the web GUI, the device manager, link discovery, a learning switch, a load balancer, and even a virtual network package.

As we said earlier, the Floodlight controller is freely downloadable. The code and packages may be viewed, and it is fairly straightforward to begin experimenting with it. Once it is configured and running it will take control of the OpenFlow switches that have been configured to use it as their controller. By default it will operate its Learning Switch SDN application, setting flow entries on switches to forward packets across a layer two network, even one with loops and a mesh topology.

Floodlight provides both reactive Java APIs and proactive RESTful APIs. We look at these two classes of APIs next.

10.6.1 Java APIs

In Section 10.4 we provided detailed examples of the Floodlight Java APIs. There are APIs to register listeners for various events, such as switch and device changes, and to receive incoming packets. There are the OpenFlow primitive classes and interfaces such as *IOFSwitch*, *OFMatch*, *OFAction*, and others. These APIs are fundamentally the same in Floodlight, Beacon, and OpenDaylight. The Floodlight GUI is extensible and can be used to incorporate one's own application into the Floodlight GUI framework.

10.6.2 RESTful APIs

Proactive Floodlight applications can use the RESTful APIs to gather data about the network, to program flows in the network, and even to create virtual networks and manipulate a firewall. The following list of RESTful APIs for Floodlight provides an indication of the functionality available:

- **Switch** provides APIs to retrieve statistics for all or for a single switch.
- **Controller** provides APIs to retrieve a list of all switches being managed by this controller.
- **Counter** provides APIs to retrieve a list of traffic counters per switch.
- **Memory** provides APIs to retrieve the current memory usage by the controller.
- **Topology** provides APIs to retrieve a list of interswitch links, external switch links, and switch clusters.
- **Device** provides APIs to retrieve a list of all devices.
- **StaticFlowPusher** provides APIs to add or remove static flows from switches and to retrieve a list of static flows on a switch.
- **Network Service (aka Virtual Network)** provides APIs to create, modify, or remove a virtual network, to attach or remove a host from a virtual network, and to retrieve a list of virtual networks and their hosts.

The extensible RESTful API uses the Restlet framework [3] and includes a small web server that allows external applications and GUIs to communicate with the SDN application. Adding a RESTful interface to the SDN application is straightforward:

- Add the RESTful API and the associated class for handling calls to the API by instantiating a *RestRoutable* object, identifying the RESTful URL and the handling class.
- Create the RESTful resource handler class by extending the *ServerResource* class.
- Use Jackson [4] or a similar tool to translate back and forth between the data portions (often in JSON [5] format) of the RESTful requests and replies.

Since the resource handler is part of the Floodlight application, it has access to the rest of the application and its data. In addition, there are a number of Floodlight modules that can be used for various purposes. For example, Floodlight has information about the OpenFlow switches and devices (which are hosts or end nodes in the Floodlight jargon). It is possible to query the switch to find out about its ports and to query the devices about their attachment point (switch and port) to the network.

A fine example of how to write a Floodlight application is provided in [6], which should be of assistance in creating one's own application to control a network of OpenFlow switches.

10.7 Using the OpenDaylight Controller

OpenDaylight is a project intended to produce a common controller to be used as a basis for SDN applications for vendors and enterprises alike. At the time of this writing, the OpenDaylight controller is in prerelease form. The OpenFlow functionality in the OpenDaylight controller is derived from the Beacon code. The lion's share of the controller was contributed to the project by Cisco. This can be confirmed by inspection of the packages documented in [7], which reveals a small number of the *org.openflow* packages discussed earlier and a large number of *org.opendaylight.controller* packages, most of which have come from Cisco.

There is a Java API in the controller for performing both OpenFlow and non-OpenFlow management of the network and the switches. Like Floodlight, there is also a RESTful API available.

10.7.1 Java APIs

The OpenFlow functionality provided by the OpenFlow packages in the Floodlight, Beacon, and OpenDaylight controllers is essentially the same. Applications that run on Floodlight run with little modification on Beacon and OpenDaylight.

OpenDaylight provides an additional abstraction called the *Service Abstraction Layer* (SAL). SAL purportedly allows network programmers to write applications similarly to the way they would be written calling the OpenFlow Java APIs, with the advantage that SAL can translate the API requests into either OpenFlow or another non-OpenFlow API mechanism that may be available on the target switch. It should thus be possible to write applications that work with both OpenFlow and non-OpenFlow switches.

10.7.2 RESTful APIs

The RESTful APIs for OpenDaylight provide functionality that is similar to that provided by Floodlight and other controllers. Here is a summary of the categories of API:

- **Topology** provides APIs to retrieve the interswitch links in the network.
- **Host Tracker** provides APIs to retrieve the hosts (end nodes) in the network.
- **Flow Programmer** provides APIs for reading and writing flows on specific switches in the network.
- **Static Routing** provides APIs for reading and writing static routes (e.g., next-hop rules) on switches in the network.
- **Statistics** provides APIs for retrieving statistics for flows, ports, tables, and switches.
- **Subnets** provides APIs for retrieving information about subnets.
- **Switch Manager** provides APIs for retrieving information about switches.

10.8 Using the Cisco XNC Controller

We have mentioned that the bulk of the OpenDaylight controller implementation has come from Cisco. The Cisco *Extensible Network Controller* (XNC) was originally known as the Cisco ONE controller. The XNC controller is Cisco's commercial SDN controller, and there is considerable overlap between XNC and OpenDaylight. For example, the user interfaces for OpenDaylight and for Cisco XNC are very similar. However, there is some functionality that is part of the Cisco XNC controller that is not available with OpenDaylight. We list those next.

10.8.1 RESTful APIs

The APIs available in OpenDaylight are also available in XNC. In addition, two categories of RESTful APIs available with Cisco's XNC controller that are not part of OpenDaylight are:

- **Slices.** Slices addresses the need for administrative domains that provide different views, depending on the administrator that is using the controller. This allows the network to be partitioned into

separate domains based on the administrator's role or based on the network entities that are under the control of specific administrators.

- **Topology Independent Forwarding (TIF).** The TIF functionality in the Cisco controller allows an administrator to create paths between two endpoints without taking into consideration the intermediate hops between one end and the other.

10.9 Using the Hewlett-Packard Controller

HP is an important vendor to SDN not only because of its networking products but also because it has a large share of the server market for data centers. HP has a major portion of the services market as well, expanding significantly with its purchase of Electronic Data Systems (EDS) in 2008. HP will likely have a significant SDN presence due to SDN's projected impact in data centers, where HP servers and services are prevalent.

HP's controller is expected to be released in late 2013. Early indications of the controller are that it will have both Java and RESTful APIs.

10.9.1 Java APIs

The HP Java API has diverged from what was prevalent in the earlier controllers (Beacon, Floodlight, etc.), in order to provide support for OpenFlow 1.3, including multiple tables, TLVs, and the like and to provide a richer software development environment.

10.9.2 RESTful APIs

The HP RESTful API includes functionality such as the following:

- **Nodes** retrieves information about end nodes in the networks.
- **Datapaths** retrieves information about switches in the network and for setting flows on these devices.
- **Clusters** retrieves information about groups of switches that form a broadcast domain.
- **Regions** retrieves information about groups of clusters.
- **Forward Path** returns the shortest path between a source and destination switch.
- **Teams** retrieves information about groups of controllers working in concert for high availability (HA).

10.10 Switch Considerations

One of the most important considerations in building an SDN application is the nature of the switches that are to be controlled by the application. Some of the relevant questions are:

- Do all the switches support OpenFlow? What version of OpenFlow do they support? How much of that version do they in fact support?
- Is there a mix of OpenFlow and non-OpenFlow switches? If so, will some non-OpenFlow mechanism be used to provide the best simulation of an SDN environment? For example, do the non-OpenFlow switches support some type of SDN API that allows a controller to configure flows in the switch?

- For OpenFlow-supporting switches, are they hardware or software switches, or both? If both, what differences in behavior and support will need to be considered?
- For hardware OpenFlow switches, what are their hardware flow-table sizes? If the switches are capable of handling flow table overflow in software, at what point are they forced to begin processing flows in software, and what is the performance impact of doing so? Is it even possible to process flows in software? What feature limitations are introduced because of the hardware?
- What is the mechanism by which the switch-controller communications channel is secured? Is this the responsibility of the network programmer?

Hardware switches vary in their flow capacity, feature limitations, and performance, and these should be considerations in creating an SDN application. The major issue regarding flow capacity is how many flow entries the hardware ASIC is capable of holding. Some switches support fewer than 1,000 flow entries, whereas other, such as the NEC PF5240 [8], support greater than 150,000. Note that for some switches the maximum number of flows depends on the nature of the flows supported. For example, the NEC PF5820 switch [9] can support 80,000 layer-two-only flows but only 750 full 12-tuple flows. (We described matching against the basic 12-tuple of input fields in Section 5.3.3.) A layer two flow entry only requires matching on the MAC address, so the amount of logic and memory occupied per flow entry is much less. Some applications, such as a traffic prioritization application or a TCP port-specific application, may use relatively few flow table entries. Others, such as an access control application or a per-user application, will be much more hungry for flow entries. The topological placement of the application logic may have a bearing on the maximum number of flow entries as well. For example, an access control application being applied in a switch or AP at the user edge of the network will require far fewer flow entries per device than the same application controlling an upstream or core switch.

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

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

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