This section provided an overview of the composition of SDN devices and the considerations that must be taken into account during their development and their use as part of an SDN application. We provide further specifics on flow tables, flow entries, and actions in Chapter 5.

4.3.4 Existing SDN Device Implementations

A number of SDN device implementations are available today, both commercial and open source. Software SDN devices are predominantly open source. Currently, two main alternatives are available: *Open vSwitch* (OVS) [4] from Nicira and *Indigo* [5] from Big Switch. Incumbent *network equipment manufacturers* (NEMs), such as Cisco, HP, NEC, IBM, Juniper, and Extreme, have added OpenFlow support to some of their legacy switches. Generally, these switches may operate in both legacy mode as well as OpenFlow mode. There is also a new class of devices called *white-box switches*, which are minimalist in that they are built primarily from merchant silicon switching chips and a commodity CPU and memory by a low-cost *original device manufacturer* (ODM) lacking a well-known brand name. One of the premises of SDN is that the physical switching infrastructure may be built from OpenFlow-enabled white-box switches at far less direct cost than switches from established NEMs. Most legacy control plane software is absent from these devices, since this functionality is largely expected to be provided by a centralized controller. Such white-box devices often use the open source OVS or Indigo switch code for the OpenFlow logic, then map the packet-processing part of those switch implementations to their particular hardware.

4.3.5 Scaling the Number of Flows

The granularity of flow definitions will generally be more fine as the device holding them approaches the edge of the network and will be more general as the device approaches the core. At the edge, flows will permit different policies to be applied to individual users and even different traffic types of the same user. This will imply, in some cases, multiple flow entries for a single user. This level of flow granularity would simply not scale if it were applied closer to the network core, where large switches deal with the traffic for tens of thousands of users simultaneously. In those core devices, the flow definitions will be generally more coarse, with a single aggregated flow entry matching the traffic from a large number of users whose traffic is aggregated in some way, such as a tunnel, a VLAN, or a MPLS LSP. Policies applied deep into the network will likely not be user-centric policies but rather policies that apply to these aggregated flows. One positive result is that there will not be an explosion in the number of flow entries in the core switches due to handling the traffic emanating from thousands of flows in edge switches.

4.4 SDN Controller

We have noted that the controller maintains a view of the entire network, implements policy decisions, controls all the SDN devices that comprise the network infrastructure, and provides a northbound API for applications. When we have said that the controller implements policy decisions regarding routing, forwarding, redirecting, load balancing, and the like, these statements referred to both the controller and

the applications that make use of that controller. Controllers often come with their own set of common application modules, such as a learning switch, a router, a basic firewall, and a simple load balancer. These are really SDN applications, but they are often bundled with the controller. Here we focus strictly on the controller.

Figure 4.5 exposes the anatomy of an SDN controller. The figure depicts the modules that provide the controller's core functionality, both a northbound and a southbound API, and a few sample applications that might use the controller. As we described earlier, the southbound API is used to interface with the SDN devices. This API is OpenFlow in the case of Open SDN or some proprietary alternative in other SDN solutions. It is worth noting that in some product offerings, both OpenFlow and alternatives coexist on the same controller. Early work on the southbound API has resulted in more maturity of that interface with respect to its definition and standardization. OpenFlow itself is the best example of this maturity, but de facto standards such as the Cisco CLI and SNMP also represent standardization in the southbound-facing interface. OpenFlow's companion protocol, OF-Config [17], and Nicira's *Open vSwitch Database Management Protocol* (OVSDB) [18] are both open protocols for the southbound interface, though these are limited to configuration roles.

Unfortunately, there is currently no northbound counterpart to the southbound OpenFlow standard or even the de facto legacy standards. This lack of a standard for the controller-to-application interface is considered a current deficiency in SDN, and some bodies are developing proposals to standardize it. The absence of a standard notwithstanding, northbound interfaces have been implemented in a number of disparate forms. For example, the Floodlight controller [2] includes a Java API and a *Representational State Transfer* (RESTful) [13] API. The OpenDaylight controller [14] provides a RESTful API for applications running on separate machines. The northbound API represents an outstanding opportunity for innovation and collaboration among vendors and the open source community.

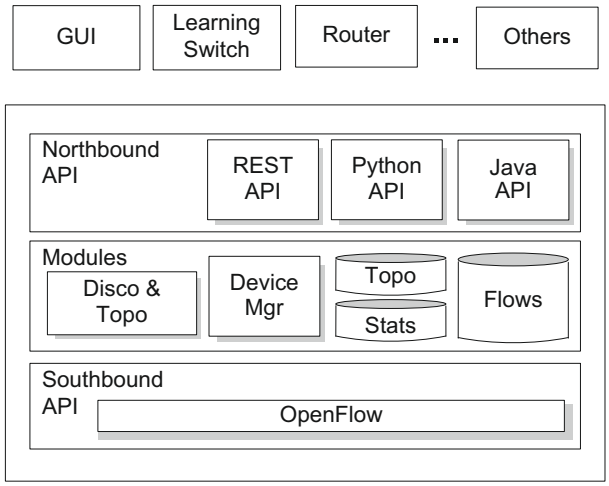

FIGURE 4.5

SDN controller anatomy.

4.4.1 SDN Controller Core Modules

The controller abstracts the details of the SDN controller-to-device protocol so that the applications above are able to communicate with those SDN devices without knowing their nuances. Figure 4.5 shows the API below the controller, which is OpenFlow in Open SDN, and the interface provided for applications. Every controller provides core functionality between these raw interfaces. Core features in the controller include:

- **End-user device discovery.** Discovery of end-user devices such as laptops, desktops, printers, mobile devices, and so on.
- **Network device discovery.** Discovery of network devices that comprise the infrastructure of the network, such as switches, routers, and wireless access points.
- **Network device topology management.** Maintain information about the interconnection details of the network devices to each other and to the end-user devices to which they are directly attached.
- **Flow management.** Maintain a database of the flows being managed by the controller and perform all necessary coordination with the devices to ensure synchronization of the device flow entries with that database.

The core functions of the controller are device and topology discovery and tracking, flow management, device management, and statistics tracking. These are all implemented by a set of modules internal to the controller. As shown in Figure 4.5, these modules need to maintain local databases containing the current topology and statistics. The controller tracks the topology by learning of the existence of switches (SDN devices) and end-user devices and tracking the connectivity between them. It maintains a *flow cache* that mirrors the flow tables on the various switches it controls. The controller locally maintains per-flow statistics that it has gathered from its switches. The controller may be designed such that functions are implemented via pluggable modules such that the feature set of the controller may be tailored to an individual network's requirements.

4.4.2 SDN Controller Interfaces

For SDN applications, a key function provided by the SDN controller is the API for accessing the network. In some cases, this northbound API is a low-level interface, providing access to the network devices in a common and consistent manner. In this case, that application is aware of individual devices but is shielded from their differences. In other instances the controller may provide high-level APIs that give an abstraction of the network itself, so that the application developer need not be concerned with individual devices but rather with the network as a whole.

Figure 4.6 takes a closer look at how the controller interfaces with applications. The controller informs the application of *events* that occur in the network. Events are communicated from the controller to the application. Events may pertain to an individual packet that has been received by the controller or some state change in the network topology, such as a link going down. Applications use different *methods* to affect the operation of the network. Such methods may be invoked in response to a received event and may result in a received packet being dropped, modified, and/or forwarded or the addition, deletion, or modification of a flow. The applications may also invoke methods independently, without the stimulus of an event from the controller, as we explain in Section 4.5.1. Such inputs are represented by the "Other Context" box in Figure 4.6.

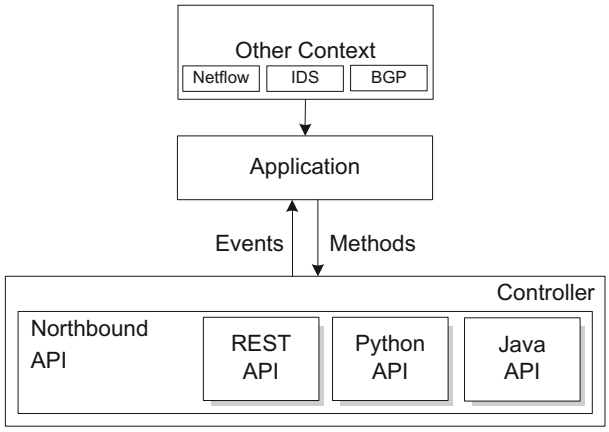

FIGURE 4.6

SDN controller northbound API.

4.4.3 Existing SDN Controller Implementations

There are a number of implementations of SDN controllers available on the market today. They include both open source SDN controllers and commercial SDN controllers. Open source SDN controllers come in many forms, from basic C-language controllers such as NOX [7] to Java-based versions such as Beacon [1] and Floodlight [2]. There is even a Ruby-based [8] controller called Trema [9]. Interfaces to these controllers may be offered in the language in which the controller is written or other alternatives, such as REST or Python. An open source controller called OpenDaylight [3] has been built by a consortium of vendors. Other vendors offer their own commercial versions of an SDN controller. Vendors such as NEC, IBM, and HP offer controllers that are primarily OpenFlow implementations. Most other NEMs offer vendor-specific and proprietary SDN controllers that include some level of OpenFlow support.

There are pros and cons to the proprietary alternative controllers. Although proprietary controllers are more closed than the nominally open systems, they do offer some of the automation and programmability advantages of SDN while providing a *buck stops here* level of support for the network equipment. They permit SDN-like operation of legacy switches, obviating the need to replace older switching equipment in order to begin the migration to SDN. They do constitute closed systems, however, which ostensibly violates one of the original tenets of SDN. They also may do little to offload control functionality from devices, resulting in the continued high cost of network devices. These proprietary alternative controllers are generally a component of the alternative SDN methodologies we introduce in Section 4.6.

4.4.4 Potential Issues with the SDN Controller

In general, the Open SDN controller suffers from the birthing pains common to any new technology. Although many important problems are addressed by the concept and architecture of the controller, there have been comparatively few large-scale commercial deployments thus far. As more commercial deployments scale, more real-life experience in large, demanding networks will be needed. In particular,

experience with a wider array of applications with a more heterogeneous mix of equipment types is needed before widespread confidence in this architecture is established. Achieving success in these varied deployments will require that a number of potential controller issues be adequately addressed. In some cases, these solutions will come in multiple forms from different vendors. In other cases, a standards body such as the ONF will have to mandate a standard. In Section 3.2.6 we stated that a centralized control architecture needed to grapple with the issues of latency, scale, high availability, and security. In addition to these more general issues, the centralized SDN controller will need to confront the challenges of *coordination between applications*, *the lack of a standard northbound API*, and *flow prioritization*.

There may be more than one SDN application running on a single controller. When this is the case, issues related to application prioritization and flow handling become important. Which application should receive an event first? Should the application be required to pass along this event to the next application in line, or can it deem the processing complete, in which case no other applications get a chance to examine and act on the received event?

The lack of an emerging standard for the northbound API is stymieing efforts to develop applications that will be reusable across a wide range of controllers. Early standardization efforts for OpenFlow generally assumed that such a northbound counterpart would emerge, and much of the efficiency gain assumed to come from a migration to SDN will be lost without it. Late in 2013 the ONF formed a workgroup that focuses on the standardization of the northbound API (see Table 3.2).

Flows in an SDN device are processed in priority order. The first flow that matches the incoming packet is acted upon. Within a single SDN application, it is critical for the flows on the SDN device to be prioritized correctly. If they are not, the resulting behavior will be incorrect. For example, the designer of an application will put more specific flows at a higher priority (e.g., match all packets from IP address 10.10.10.2 and TCP port 80) and the most general flows at the lowest priority (e.g., match everything else). This is relatively easy to do for a single application. However, when there are multiple SDN applications, flow entry prioritization becomes more difficult to manage. How does the controller appropriately interleave the flows from all applications? This is a challenge and requires special coordination between the applications.

4.5 SDN Applications

SDN applications run above the SDN controller, interfacing to the network via the controller's northbound API. SDN applications are ultimately responsible for managing the flow entries that are programmed on the network devices using the controller's API to manage flows. Through this API the applications are able to (1) configure the flows to route packets through the best path between two endpoints; (2) balance traffic loads across multiple paths or destined to a set of endpoints; (3) react to changes in the network topology such as link failures and the addition of new devices and paths, and (4) redirect traffic for purposes of inspection, authentication, segregation, and similar security-related tasks.

Figure 4.5 includes some standard applications, such as a *graphical user interface* (GUI) for managing the controller, a learning switch, and a routing application. The reader should note that even the basic functionality of a simple layer two learning switch is not obtained by simply pairing an SDN device with an SDN controller. Additional logic is necessary to react to the newly seen MAC address and update the forwarding tables in the SDN devices being controlled in such a way as to provide connectivity to that new MAC address throughout the network while avoiding switching loops. This additional logic is