CHAPTER

THE OpenFlow SPECIFICATION

5

Casual followers of networking news can experience confusion as to whether SDN and OpenFlow are one and the same thing. Indeed, they are not. OpenFlow is definitely a distinct subset of the technologies included under the big tent of SDN. An important tool and catalyst for innovation, OpenFlow defines both the communications protocol between the SDN data plane and the SDN control plane, as well as part of the behavior of the data plane. It does not describe the behavior of the controller itself. While there are other approaches to SDN, OpenFlow is today the only nonproprietary, general-purpose protocol for programming the forwarding plane of SDN switches. As such, this chapter will focus only on the protocol and behaviors dictated by OpenFlow, as there are no directly competing alternatives.

An OpenFlow system consists of an *OpenFlow controller* that communicates to one or more *OpenFlow Switches*. The OpenFlow protocol defines the specific messages and message formats exchanged between controller (control plane) and device (data plane). The OpenFlow behavior specifies how the device should react in various situations, and how it should respond to commands from the controller. There have been various versions of OpenFlow, which we examine in detail in this chapter. We will also consider some of the potential drawbacks and limitations of the OpenFlow specification.

Note that our goal in this chapter is not to provide an alternative to a detailed reading of the specification itself. We hope to provide the reader with a rudimentary understanding of the elements of OpenFlow and how they interoperate to provide basic switch and routing functions, as well as an experimental platform that allows each version to serve as a springboard for innovations that may appear in future versions of OpenFlow. In particular, while we provide tables that list the different ports, messages, instructions and actions that make up OpenFlow, we do not attempt to explain each in detail, as this would essentially require that we replicate the specification here. We will use the most important of these in clear examples so that the reader is left with a basic understanding of OpenFlow operation. We provide a reference to each of the four versions of OpenFlow in the section where it is introduced.

5.1 CHAPTER-SPECIFIC TERMINOLOGY

The following new operations are introduced for this chapter.

To *pop* an item is to remove it from a *Last-In-First-Out* (LIFO) ordered list of like items.

To *push* an item is to add it to a LIFO ordered list of like items.

These terms are frequently used in conjunction with the term *stack* which is a computer science term for a LIFO ordered list. If one pictures a stack of items such as books, and three books are added to that stack, the order in which those books are normally retrieved from the pile is a LIFO order. That is, the first book pulled off the top of the pile is the last one that was added to it. If we alter slightly the perception of the stack of books to imagine that the stack exists in a spring-loaded container such

Software Defined Networks. http://dx.doi.org/10.1016/B978-0-12-804555-8.00005-3

that the top-most book is level with the top of the container, the use of the terms pop and push become apparent. When we add a book to such a stack we *push* it onto the top and later *pop* it off to retrieve it. The OpenFlow specification includes several definitions that use these concepts. In this context, the push involves adding a new header element to the existing packet header, such as an MPLS label. Since multiple such labels may be pushed on before any are popped off, and they are popped off in LIFO order, the stack analogy is apt.

5.2 OpenFlow OVERVIEW

The OpenFlow specification has been evolving for a number of years. The nonprofit Internet organization, *openflow.org* was created in 2008 as a mooring to promote and support OpenFlow. While openflow.org existed formally on the Internet, in the early years the physical organization was really just a group of people that met informally at Stanford University. From its inception OpenFlow was intended to belong to the research community to serve as a platform for open network switching experimentation, with an eye on commercial use through commercial implementations of this public specification. The first release Version 1.0.0 appeared in December 31, 2009, though numerous point prereleases existed before then that were made available for experimental purposes as the specification evolved. At this point and continuing up through release 1.1.0, development and management of the specification was performed under the auspices of openflow.org. On March 21, 2011 the *Open Network Foundation* (ONF) was created for the express purpose of accelerating the delivery and commercialization of SDN. As we will explain in Chapter 6, there are a number of proponents of SDN that offer SDN solutions that are not based on OpenFlow. For the ONF, however, OpenFlow remains at the core of their SDN vision for the future. For this reason the ONF has become the responsible entity for the evolving OpenFlow specification. Starting after the release of V.1.1, revisions to the OpenFlow specification are released and managed by the ONF.

One may get the impression from the fanfare surrounding OpenFlow that the advent of this technology has been accompanied by concomitant innovation in switching hardware. The reality is a bit more complicated. The OpenFlow designers realized a number of years ago that many switches were really built around ASICs controlled by rules encoded in tables that could be programmed. Over time, fewer home-grown versions of these switching chips were being developed, and there was greater consolidation in the semiconductor industry. More manufacturers' switches were based on ever-consolidating switching architecture and programmability, with ever-increasing use of programmable switching chips from a relatively small number of merchant silicon vendors. OpenFlow is an attempt to allow the programming, in a generic way, of the different implementations of switches that conform to this new paradigm. OpenFlow attempts to exploit the table-driven design extant in many of the current silicon solutions. As the number of silicon vendors consolidates, there should be greater possibility for alignment with future OpenFlow versions.

It is worth pausing here to remark on the fact that we are talking a lot about ASICs for a technology called *Software* Defined Networking. Hardware must be part of the discussion since it is necessary to use this specialized silicon in order to switch packets at high line rates. We explained in Chapter 4 that while pure software SDN implementations exist, they cannot switch packets at sufficiently high rates to keep up with high-speed interfaces. What is really meant by the word *software* in the name SDN, then, is that the SDN devices are fully programmable, not that everything is done using software running on a traditional CPU.

In the sections that follow we will introduce the formal terminology used by OpenFlow and provide basic background that will allow us to explore the details of the different versions of the OpenFlow specification that have been released up to the time of the writing of this book.

5.2.1 THE OpenFlow SWITCH

Fig. 5.1 depicts the basic functions of an OpenFlow V.1.0 switch and its relationship to a controller. As would be expected in a packet switch, we see that the core function is to take packets that arrive on one port (path *X* on port 2 in the figure) and forward it through another port (port *N* in the figure), making any necessary packet modifications along the way. A unique aspect of the OpenFlow switch is embodied in the *packet matching function* shown in Fig. 5.1. The adjacent table is a *Flow Table* and we will give separate treatment to this below in Section 5.3.2. The wide, grey, double arrow in Fig. 5.1 starts in the decision logic, shows a match with a particular entry in that table, and directs the now-matched packet to an *action box* on the right. This action box has three fundamental options for the disposition of this arriving packet:

- *A*: Forward the packet out a local port, possibly modifying certain header fields first.
- *B*: Drop the packet.
- *C*: Pass the packet to the controller.

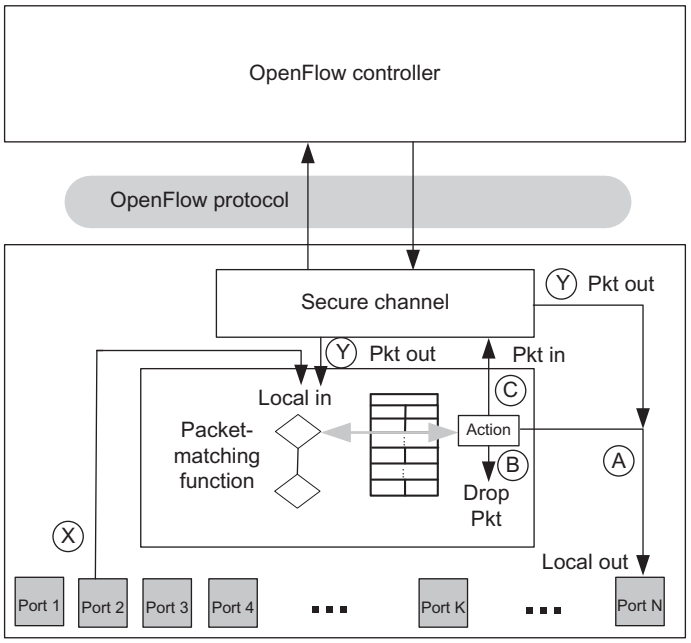

FIG. 5.1

OpenFlow V.1.0 switch.

These three fundamental packet paths are illustrated in Fig. 5.1. In the case of path *C*, the packet is passed to the controller over the *secure channel* shown in the figure. If the controller has either a control message or a data packet to give to the switch, the controller uses this same secure channel in the reverse direction. When the controller has a data packet to forward out through the switch, it uses the OpenFlow PACKET_OUT message. We see in Fig. 5.1 that such a data packet coming from the controller may take two different paths through the OpenFlow logic, both denoted *Y*. In the rightmost case, the controller directly specifies the output port and the packet is passed to that port *N* in the example. In the leftmost path *Y* case, the controller indicates that it wishes to defer the forwarding decision to the packet matching logic. We will see in Section 5.3.4 that the controller dictates this by stipulating the virtual port TABLE as the output port.

A given OpenFlow switch implementation is either *OpenFlow-only* or *OpenFlow-hybrid*. An OpenFlow-only switch is one that forwards packets *only* according to the OpenFlow logic described above. An OpenFlow hybrid is a switch that can also switch packets in its legacy mode as an Ethernet Switch or IP router. One can view the hybrid case as an OpenFlow switch residing next to a completely independent traditional switch. Such a hybrid switch requires a preprocessing classification mechanism that directs packets to either OpenFlow processing or the traditional packet processing. It is probable that hybrid switches will be the norm during the migration to pure OpenFlow implementations.

Note that we use the term OpenFlow *switch* in this chapter instead of the term OpenFlow *device* we customarily use. This is because switch is the term used in the OpenFlow specification. In general, though, we opt to use the term device since there are already nonswitch devices being controlled by OpenFlow controllers, such as wireless access points.

DISCUSSION QUESTION:

Why are there two PKT_OUT options emerging from the Secure Channel box in Fig. 5.1?

5.2.2 THE OpenFlow CONTROLLER

Modern Internet switches make millions of decisions per second about whether or not to forward an incoming packet, to what set of output ports it should be forwarded, and what header fields in the packet may need to be modified, added, or removed. This is a very complex task. The fact that this can be carried out at line rates on multigigabit media is a technological wonder. The switching industry has long understood that not all functions on the switching datapath can be carried out at line rates, so there has long been the notion of splitting the pure data plane from the control plane. The data plane matches headers, modifies packets and forwards them based on a set of forwarding tables and associated logic, and it does this very, very fast. The rate of decisions being made as packets stream into a switch through a 100Gbs interface is astoundingly high. The control plane runs routing and switching protocols and other logic to determine what the forwarding tables and logic in the data plane should be. This process is very complex and cannot be done at line rates as the packets are being processed, and it is for this reason we have seen the control plane separated from the data plane even in legacy network switches.

The OpenFlow control plane differs from the legacy control plane in three key ways. First, it can program different data plane elements with a common, standard language, OpenFlow. Second, it exists

on a separate hardware device than the forwarding plane, unlike traditional switches where the control plane and data plane are instantiated in the same physical box. This separation is made possible because the controller can program the data plane elements remotely over the Internet. Third, the controller can program multiple data plane elements from a single control plane instance.

The OpenFlow controller is responsible for programming all of the packet matching and forwarding *rules* in the switch. Whereas a traditional router would run routing algorithms to determine how to program its forwarding table, that function or *an equivalent replacement* to it is now performed by the controller. Any changes that result in recomputing routes will be programmed onto the switch by the controller.

5.2.3 THE OpenFlow PROTOCOL

As shown in Fig. 5.1 the OpenFlow protocol defines the communication between an OpenFlow controller and an OpenFlow switch. This protocol is what most uniquely identifies OpenFlow technology. At its essence, the protocol consists of a set of messages that are sent from the controller to the switch and a corresponding set of messages that are sent in the opposite direction. The messages, collectively, allow the controller to program the switch so as to allow fine-grained control over the switching of user traffic. The most basic programming defines, modifies and deletes flows. Recall that in Chapter 4 we defined a flow as *a set of packets transferred from one network endpoint (or set of endpoints) to another endpoint (or set of endpoints). The endpoints may be defined as IP address-TCP/UDP port pairs, VLAN endpoints, layer three tunnel endpoints, and input port among other things.* One set of rules describes the forwarding actions that the device should take for all packets belonging to that flow. When the controller defines a flow, it is providing the switch with the information it needs to know how to treat incoming packets that match that flow. The possibilities for treatment have grown more complex as the OpenFlow protocol has evolved, but the most basic prescriptions for treatment of an incoming packet are denoted by paths *A*, *B*, and *C* in Fig. 5.1. These three options are to forward the packet out one or more output ports, drop the packet, or pass the packet to the controller for exception handling.

The OpenFlow protocol has evolved significantly with each version of OpenFlow, so we will cover the detailed messages of the protocol in the version-specific sections that follow. The specification has evolved from development point release 0.2.0 on March 28, 2008 through release V.1.5.0, released in 2014. Numerous point releases over the intervening years have addressed problems with earlier releases and added incremental functionality. OpenFlow was viewed primarily as an experimental platform in its early years. As such, there was little concern on the part of the development community advancing this standard to provide for interoperability between releases. As OpenFlow began to see more widespread commercial deployment, backwards compatibility has become an increasingly important issue. There are many features, however, that were introduced in earlier versions of OpenFlow that are no longer present in the current version. Since the goal of this chapter is to provide a roadmap to understanding OpenFlow as it exists today, we will take a hybrid approach of covering the major releases that have occurred since V.1.0. We focus on those key components of each release that became the basis for the advances in subsequent releases and do not focus on functionality in earlier releases that has been subsumed by new features in subsequent releases.

5.2.4 THE CONTROLLER-SWITCH SECURE CHANNEL

The secure channel is the path used for communications between the OpenFlow controller and the OpenFlow device. Generally, this communication is secured by TLS-based asymmetrical encryption, though unencrypted TCP connections are allowed. These connections may be *in-band* or *out-of-band*. Fig. 5.2 depicts these two variants of the secure channel. In the out-of-band example, we see in the figure that the secure channel connection enters the switch via port *Z*, which is *not* switched by the OpenFlow data plane. Some legacy network stack will deliver the OpenFlow messages via the secure channel to the secure channel process in the switch, where all OpenFlow messages are parsed and handled. Thus, the out-of-band secure channel is only relevant in the case of a OpenFlow-hybrid switch.

In the in-band example, we see the OpenFlow messages from the controller arriving via port *K*, which is part of the OpenFlow data plane. In this case these packets will be handled by the OpenFlow packet matching logic shown in the figure. The flow tables will have been constructed so that this OpenFlow traffic is forwarded to the LOCAL virtual port, which results in the messages being passed to the secure channel process. We discuss the LOCAL virtual port in Section 5.3.4.

Note that when the controller and all the switches it controls are located entirely within a tightly controlled environment such as a data center, it may be wise to consider not using TLS-based encryption to secure the channel. This is because there is a performance overhead incurred by using this type of security and if it is not necessary it is preferable to not pay this performance penalty. Another argument against using TLS-based encryption is that it may be used incorrectly. To put TLS in practice one must

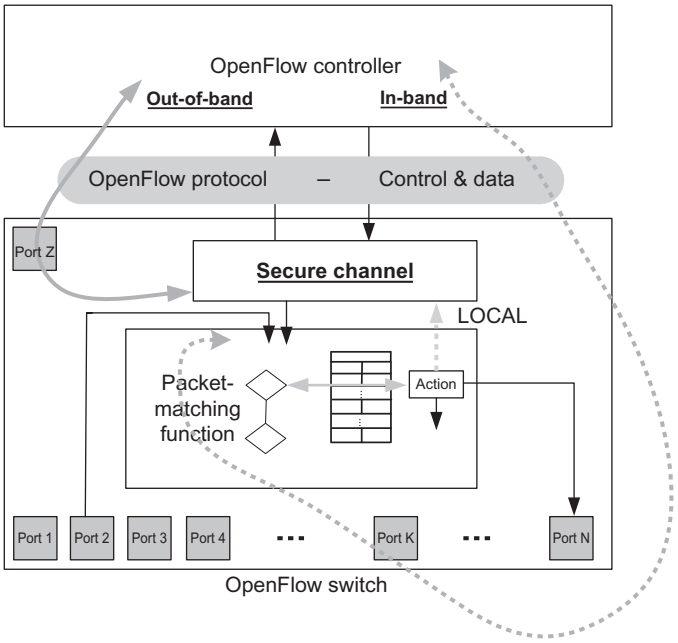

FIG. 5.2

OpenFlow controller-switch secure channel.

obtain and configure security certificates for each device, which can be time-consuming and error-prone for someone not familiar with these methods.

5.3 OpenFlow 1.0 AND OpenFlow BASICS

OpenFlow 1.0 [1] was released on December 31, 2009. For the purposes of this work, we will treat OpenFlow 1.0 as the initial release of OpenFlow. Indeed, years of work and multiple point releases preceded the OpenFlow 1.0 release, but we will subsume all of this incremental progress into the single initial release of 1.0 as if it had occurred atomically. In this section we will describe in detail the basic components of this initial OpenFlow implementation.

5.3.1 PORTS AND PORT QUEUES

The OpenFlow specification defines the concept of an *OpenFlow Port*. An OpenFlow V.1.0 port corresponds to a physical port. This concept is expanded in subsequent releases of OpenFlow. For many years, sophisticated switches have supported multiple *queues* per physical port. These queues are generally served by scheduling algorithms that allow the provisioning of different *Quality of Service* (QoS) levels for different types of packets. OpenFlow embraces this concept and permits a flow to be mapped to an already-defined queue at an output port. Thus, if we look back to Fig. 5.1 the output of a packet on port *N* may include specifying onto which queue on port *N* the packet should be placed. Hence, if we zoom in on option *A* in Fig. 5.1 we reveal what we now see in Fig. 5.3. In our zoomed-in figure we can see that the *actions* box specifically enqueued the packet being processed to queue 1 in port *N*.

Note that the support for QoS is very basic in V.1.0. QoS support in OpenFlow was expanded considerably in later versions (see Section 5.6.3).

5.3.2 FLOW TABLE

The *flow table* lies at the core of the definition of an OpenFlow switch. We depict a generic flow table in Fig. 5.4. A flow table consists of *flow entries*, one of which is shown in Fig. 5.5. A flow entry consists of *header fields*, *counters*, and *actions* associated with that entry. The header fields are used as match criteria to determine if an incoming packet matches this entry. If a match exists, then the packet belongs to this flow. The counters are used to track statistics relative to this flow, such as how many packets have been forwarded or dropped for this flow. The actions fields prescribe what the switch should do with a packet matching this entry. We describe this process of packet matching and actions in Section 5.3.3.

5.3.3 PACKET MATCHING

When a packet arrives at the OpenFlow switch from an input port (or, in some cases, from the controller), it is matched against the flow table to determine if there is a matching flow entry. The following match fields associated with the incoming packet may be used for matching against flow entries:

- switch input port
- VLAN ID

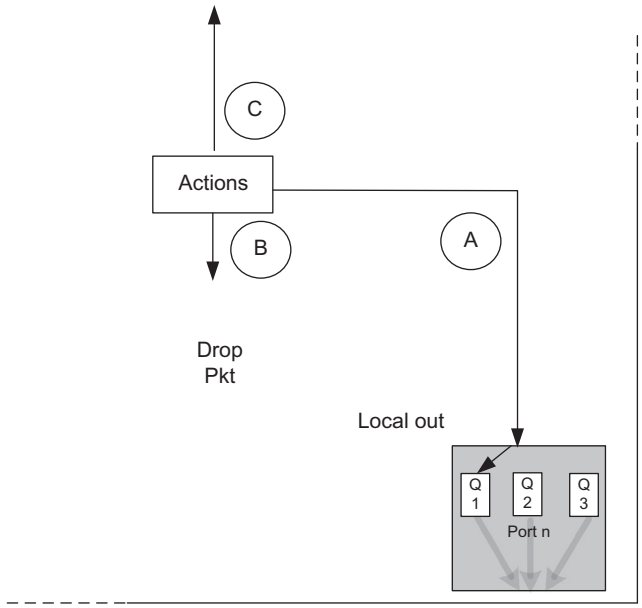

FIG. 5.3

OpenFlow support for multiple queues per port.

Flow entry 0		**Flow entry 1**		...	**Flow entry F**		...	**Flow entry M**	
Header fields	Inport 12 192.32.10.1, Port 1012	Header fields	Inport * 209.*.*.*, Port *		Header fields	Inport 2 192.32.20.1, Port 995		Header fields	Inport 2 192.32.30.1, Port 995
Counters	val	Counters	val		Counters	val		Counters	val
Actions	val	Actions	val		Actions	val		Actions	val

FIG. 5.4

OpenFlow V.1.0 flow table.

- VLAN priority
- Ethernet source address
- Ethernet destination address
- Ethernet frame type
- IP source address
- IP destination address
- IP protocol

Header fields	Field value
Counters	Field value
Actions	Field value

FIG. 5.5

Basic flow entry.

- IP *Type of Service* (ToS) bits
- TCP/UDP source port
- TCP/UDP destination port

These twelve match fields are collectively referred as the basic twelve-tuple of match fields. The flow entry's match fields may be wild-carded using a bit mask, meaning that any value that matches on the unmasked bits in the incoming packet's match fields will be a match. Flow entries are processed in order, and, once a match is found, no further match attempts are made against that flow table. (We will see in subsequent versions of OpenFlow that there may be *additional* flow tables against which packet matching may continue.) Because of this, it is possible for there to be multiple matching flow entries for a packet to be present in a flow table. Only the first flow entry to match is meaningful—the others will not be found as packet matching stops upon the first match.

The V.1.0 specification is silent about which of these twelve match fields are required versus optional. The ONF has clarified this confusion by defining three different types of conformance in their V.1.0 conformance testing program. The three levels are: *full conformance*, meaning all twelve match fields are supported, *layer two conformance*, when only the layer two header field matching is supported, and, finally, *layer three conformance*, when only layer three header field matching is supported.

If the end of the flow table is reached without finding a match, this is called a *table-miss*. In the event of a table-miss in V.1.0, the packet is forwarded to the controller. (Note that this is no longer strictly true in later versions.) If a matching flow entry is found, the actions associated with that flow entry determine how the packet is handled. The most basic action prescribed by an OpenFlow switch entry is how to forward this packet. We discuss this in the following section.

It is important to note that this V.1.0 packet matching function was designed as an abstraction of the way that real-life switching hardware works today. Early versions of OpenFlow were designed to specify the forwarding behavior of existing commercial switches via this abstraction. A good abstraction hides the details of the thing being abstracted while still permitting sufficiently fine-grained control to accomplish the needed tasks. As richer functionality is added in later versions of the OpenFlow protocol, we will see that the specification outpaces the reality of today's hardware. At that point, it is no longer providing a clean abstraction for current implementations but specifying behavior for switching hardware that has yet to be built.

5.3.4 ACTIONS AND PACKET FORWARDING

The required actions that must be supported by a flow entry are to either *output (forward)* or *drop* the matched packet. The most common case is that the output action specifies a physical port on which the packet should be forwarded. There are, however, five special virtual ports defined in V.1.0 that

have special significance for the output action. They are *LOCAL*, *ALL*, *CONTROLLER*, *IN_PORT*, and *TABLE*. We depict the packet transfers resulting from these various virtual port designations in Fig. 5.6. In the figure, the notations in brackets next to the wide shaded arrows represent one of the virtual port types discussed in this section.

LOCAL dictates that the packet should be forwarded to the switch's local OpenFlow control software, circumventing further OpenFlow pipeline processing. LOCAL is used when OpenFlow messages from the controller are received on a port that is receiving packets switched by the OpenFlow data plane. LOCAL indicates that the packet needs to be processed by the local OpenFlow control software.

ALL is used to flood a packet out all ports on the switch except the input port. This provides rudimentary broadcast capability to the OpenFlow switch.

CONTROLLER indicates that the switch should forward this packet to the OpenFlow controller.

IN_PORT instructs the switch to forward the packet back out of the port on which it arrived. Effectively, IN_PORT normally creates a loopback situation, which could be useful for certain scenarios. One scenario where this is useful is the case of an 802.11 wireless port. In this case, it is quite normal to receive a packet from that port from one host and to forward it to the receiving host via the same port. This needs to be done very carefully to not create unintended loopback situations. Thus the protocol requires explicit stipulation of this intent via this special virtual port.

Another instance when IN_PORT is required is *Edge Virtual Bridging* (EVB) [2]. Edge Virtual Bridging defines a reflective relay service between a physical switch in a data center and a lightweight virtual switch within the server known as a *Virtual Edge Port Aggregator* (VEPA). The standard IEEE 802.1Q bridge at the edge of the network will reflect packets back out the port on which they arrive to

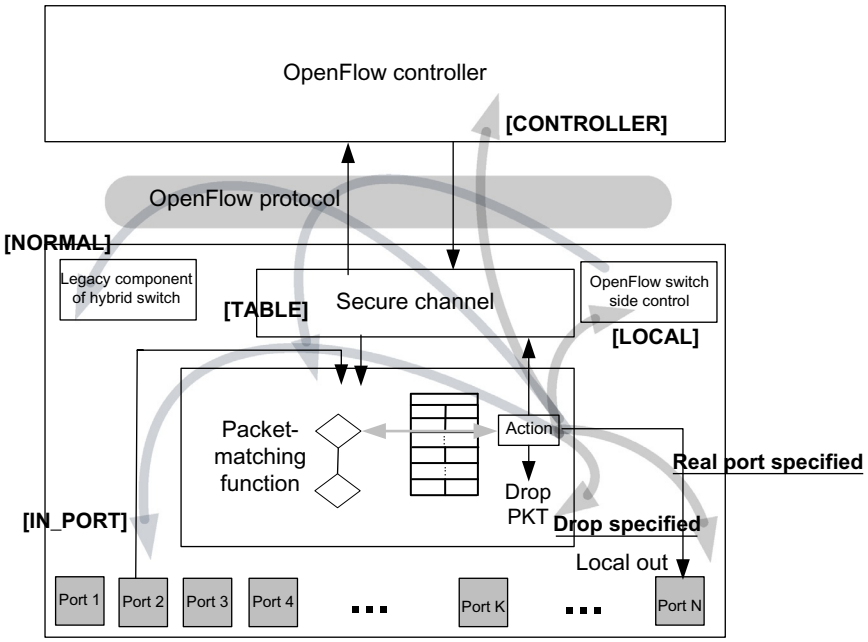

FIG. 5.6

Packet paths corresponding to virtual ports.

allow two virtual machines on the same server to talk to one another. This reflective relay service can be supported by the IN_PORT destination in OpenFlow rules.

Finally, there is the TABLE virtual port, which only applies to packets that the controller sends to the switch. Such packets arrive as part of the PACKET_OUT message from the controller, which includes an *action-list*. This action-list will generally contain an output action, which will specify a port number. The controller may wish to directly specify the output port for this data packet, or, if it wishes the output port to be determined by the normal OpenFlow packet processing pipeline, it may do so by stipulating TABLE as the output port. These two options are depicted in the two *Y* paths shown in Fig. 5.1.

There are two additional virtual ports, but support for these is optional in V.1.0. The first is the NORMAL virtual port. When the output action forwards a packet to the NORMAL virtual port, it sends the packet to the legacy forwarding logic of the switch. We contrast this with the LOCAL virtual port, which designates that the packet be passed to the local OpenFlow control processing. Conversely, packets whose matching rule indicates NORMAL as the output port will remain in the ASIC to be looked up in other forwarding tables which are populated by the local (nonOpenFlow) control plane. Use of NORMAL only makes sense in the case of a hybrid switch.

The remaining virtual port is FLOOD. In this case, the switch sends a copy of the packet out all ports except the ingress port.

Note that we have excluded the ALL and FLOOD representations from Fig. 5.6 as the number of arrows would have overly congested the portrayal. The reader should understand that ALL and FLOOD would show arrows to all ports on the OpenFlow switch except the ingress port. The reader should also note that we depict in the figure the more normal *nonvirtual* cases of *real port specified* and *drop specified* underlined without brackets. When the V.1.0 switch passes a packet to the controller as a result of finding no table matches, the path taken in Fig. 5.6 is the same as that denoted for the virtual port CONTROLLER.

There are two optional actions in V.1.0, *enqueue* and *modify-field*. The enqueue action selects a specific queue belonging to a particular port. This would be used in conjunction with the output action and is used to achieve desired QoS levels via the use of multiple priority queues on a port. Lastly, the modify-field action informs the switch how to modify certain header fields. The specification contains a lengthy list of fields that may be modified via this action. In particular, VLAN headers, Ethernet source and destination address, IPv4 source and destination address and TTL field may be modified. Modify-field is essential for the most basic routing functions. In order to route layer three packets, a router must decrement the TTL field before forwarding the packet out the designated output port.

When there are multiple actions associated with a flow entry, they appear in an action-list, just like the PACKET_OUT message described above. The switch must execute actions in the order in which they appear on the action-list. We will walk through a detailed example of V.1.0 packet forwarding in Section 5.3.7.

DISCUSSION QUESTION:

Looking back there are seven packet paths emerging from the Action box in Fig. 5.6. Five of these are for virtual ports. Which of these seven paths corresponds to the typical case of a packet arriving on one port of the switch and being forwarded out another port with minimal transformation? Which path will be followed in the case of an arriving control plane packet such as an OSPF packet destined for this switch?

5.3.5 MESSAGING BETWEEN CONTROLLER AND SWITCH

The messaging between the controller and switch is transmitted over a secure channel. This secure channel is implemented via an initial TLS connection over TCP. (Subsequent versions of OpenFlow allow for multiple connections within one secure channel.) If the switch knows the IP address of the controller, then the switch will initiate this connection. Each message between controller and switch starts with the OpenFlow header. This header specifies the OpenFlow version number, the message type, the length of the message and the transaction ID of the message. The various message types in V.1.0 are listed in Table 5.1. The messages fall into three general categories: *Symmetric*, *Controller-Switch*, and *Async*. We will explain the categories of messages shown in Table 5.1 in the following paragraphs. We suggest that the reader refer to Fig. 5.7 as we explain each of these messages below. Fig. 5.7 shows the most important of these messages in a normal context and illustrates whether it normally is used during the *initialization*, *operational*, or *monitoring* phases of the controller-switch dialogue. The operational and monitoring phases generally overlap, but for the sake of clarity we show them as disjoint in the figure. In the interest of brevity, we have truncated the OFPT_ prefix from the message names shown in Table 5.1. We will follow this convention whenever referring to these message names in the balance of the book, however they appear in the index with their full name, including their OFPT_ prefix.

Table 5.1 OFPT Message Types in OpenFlow 1.0

Message Type	Category	Subcategory
HELLO	Symmetric	Immutable
ECHO_REQUEST	Symmetric	Immutable
ECHO_REPLY	Symmetric	Immutable
VENDOR	Symmetric	Immutable
FEATURES_REQUEST	Controller-Switch	Switch Configuration
FEATURES_REPLY	Controller-Switch	Switch Configuration
GET_CONFIG_REQUEST	Controller-Switch	Switch Configuration
GET_CONFIG_REPLY	Controller-Switch	Switch Configuration
SET_CONFIG	Controller-Switch	Switch Configuration
PACKET_IN	Async	NA
FLOW_REMOVED	Async	NA
PORT_STATUS	Async	NA
ERROR	Async	NA
PACKET_OUT	Controller-Switch	Cmd from controller
FLOW_MOD	Controller-Switch	Cmd from controller
PORT_MOD	Controller-Switch	Cmd from controller
STATS_REQUEST	Controller-Switch	Statistics
STATS_REPLY	Controller-Switch	Statistics
BARRIER_REQUEST	Controller-Switch	Barrier
BARRIER_REPLY	Controller-Switch	Barrier
QUEUE_GET_CONFIG_REQUEST	Controller-Switch	Queue configuration
QUEUE_GET_CONFIG_REPLY	Controller-Switch	Queue configuration

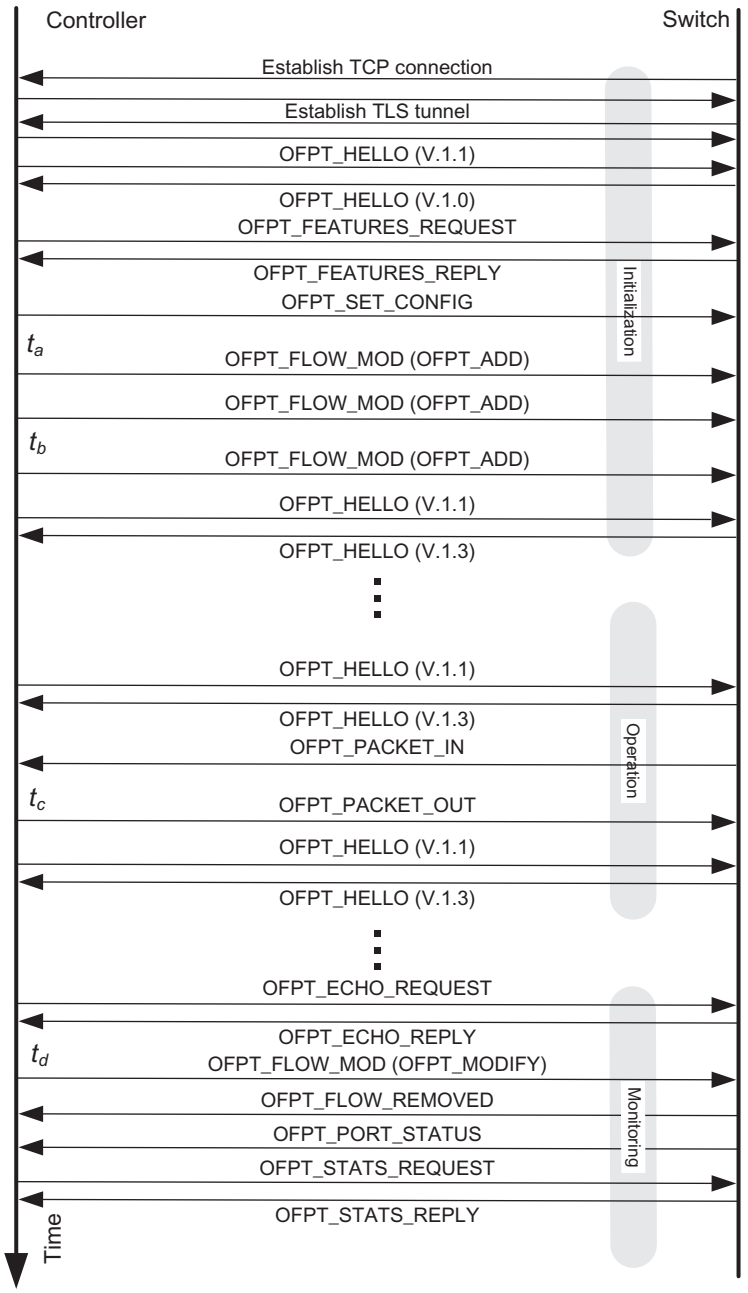

FIG. 5.7

Controller-Switch Protocol Session.

Symmetric messages may be sent by either the controller or the switch, without having been solicited by the other. The HELLO messages are exchanged after the secure channel has been established to determine the highest OpenFlow version number supported by the peers. The protocol specifies that the lower of the two versions is to be used for Controller-Switch communication over this secure channel instance. ECHO messages are used by either side during the life of the channel to ascertain that the connection is still alive and to measure the current latency or bandwidth of the connection. The VENDOR messages are available for vendor-specific experimentation or enhancements.

Async messages are sent from the switch to the controller without having been solicited by the controller. The PACKET_IN message is how the switch passes data packets back to the controller for exception handling. Control plane traffic will usually be sent back to the controller via this message. The switch can inform the controller that a flow entry is removed from the flow table via the FLOW_REMOVED message. PORT_STATUS is used to communicate changes in port status whether by direct user intervention or by a physical change in the communications medium itself. Finally, the switch uses the ERROR message to notify the controller of problems.

Controller-Switch is the broadest category of OpenFlow messages. In fact, as shown in Table 5.1, they can be divided into five subcategories: *Switch Configuration*, *Command From Controller*, *Statistics*, *Queue Configuration*, and *Barrier*. The Switch Configuration messages consist of a unidirectional configuration message and two request-reply message pairs. The unidirectional message, SET_CONFIG, is used by the controller to set configuration parameters in the switch. In Fig. 5.7 we see the SET_CONFIG message sent during the initialization phase of the controller-switch dialogue. The FEATURES message pair is used by the controller to interrogate the switch about which features it supports. Similarly, the GET_CONFIG message pair is used to retrieve a switch's configuration settings.

There are three messages comprising the *Command From Controller* category. PACKET_OUT is the analog of the PACKET_IN mentioned above. It is used by the controller to send data packets to the switch for forwarding out through the data plane. The controller modifies existing flow entries in the switch via the FLOW_MOD message. PORT_MOD is used to modify the status of an OpenFlow port.

Statistics are obtained from the switch by the controller via the STATS message pair. The BARRIER message pair is used by the controller to ensure that a particular OpenFlow command from the controller has finished executing on the switch. The switch must complete execution of all commands received prior to the BARRIER_REQUEST before executing any commands received after it, and notifies the controller of having completed such preceding commands via the BARRIER_REPLY message sent back to the controller.

The Queue Configuration message pair is somewhat of a misnomer in that actual queue configuration is beyond the scope of the OpenFlow specification and is expected to be done by an unspecified out-of-band mechanism. The QUEUE_GET_CONFIG_REQUEST and QUEUE_GET_CONFIG_REPLY message pair is the mechanism by which the controller learns from the switch how a given queue is configured. With this information, the controller can intelligently map certain flows to specific queues to achieve desired QoS levels.

Note that *immutable* in this context means that the message types will not be changed in future releases of OpenFlow. *Author's note: It is somewhat remarkable that the immutable characteristic is cited as distinct in OpenFlow. In many more mature communications protocols, maintaining backwards compatibility is of paramount importance. This is not the case with OpenFlow where*

support for some message formats in earlier versions is dropped in later versions. This situation is handled within the OpenFlow environment by the fact that two OpenFlow implementations coordinate their version numbers via the HELLO protocol, and presumably the higher-versioned implementation reverts to the older version of the protocol for the sake of interoperability with the other device.

In the event that the HELLO protocol detects a loss of the connection between controller and switch, the V.1.0 specification prescribes that the switch should enter *emergency mode* and reset the TCP connection. All flows are to be deleted at this time except special flows that are marked as being part of the *emergency flow cache*. The only packet matching that is allowed in this mode is against those flows in that emergency flow cache. Which flows should be in this cache was not prescribed in the specification, and subsequent versions of OpenFlow have addressed this area differently and more thoroughly.

DISCUSSION QUESTION:

Fig. 5.7 depicts three groupings of controller-switch communications. OFPT_FLOW_MOD messages appear in both the initialization and monitoring groupings. What is the difference between the messages? Do you think that the type of OFPT_FLOW_MOD messages shown in the initialization phase could be interspersed with messages in operation and monitoring?

5.3.6 EXAMPLE: CONTROLLER PROGRAMMING FLOW TABLE

In Fig. 5.8 two simple OpenFlow V.1.0 flow table modifications are performed by the controller. We see in the figure that the initial flow table has three flows. In the quiescent state before time t_a, we see an exploded version of flow entry zero. It shows that the flow entry specifies that all Ethernet frames entering the switch on input port *K* with a destination Ethernet address of 0x000CF15698AD should be output on output port *N*. All other match fields have been wildcarded, indicated by the asterisks in their respective match fields in Fig. 5.8. At time t_a, the controller sends a FLOW_MOD(ADD) command to the switch adding a flow for packets entering the switch on any port, with source IP addresses 192.168.1.1 and destination IP address 209.1.2.1, source TCP port 20 and destination port 20. All other match fields have been wildcarded. The outport port is specified as *P*. We see that after this controller command is received and processed by the switch, the flow table contains a new flow entry *F* corresponding to that ADD message. At time t_b, the controller sends a FLOW_MOD(MODIFY) command for flow entry zero. The controller seeks to modify the corresponding flow entry such that there is a one hour (3600 second) idle time on that flow. The figure shows that after the switch has processed this command, the original flow entry has been modified to reflect that new idle time. Note that idle time for a flow entry means that after that number of seconds of inactivity on that flow, the flow should be deleted by the switch. Looking back at Fig. 5.7, we see an example of such a flow expiration just after time t_d. The FLOW_REMOVED message we see there indicates that the flow programmed at time t_b in our examples in Figs. 5.7 and 5.8 has expired. This controller had requested to be notified of such expiration when the flow was configured and the FLOW_REMOVED message serves this purpose.

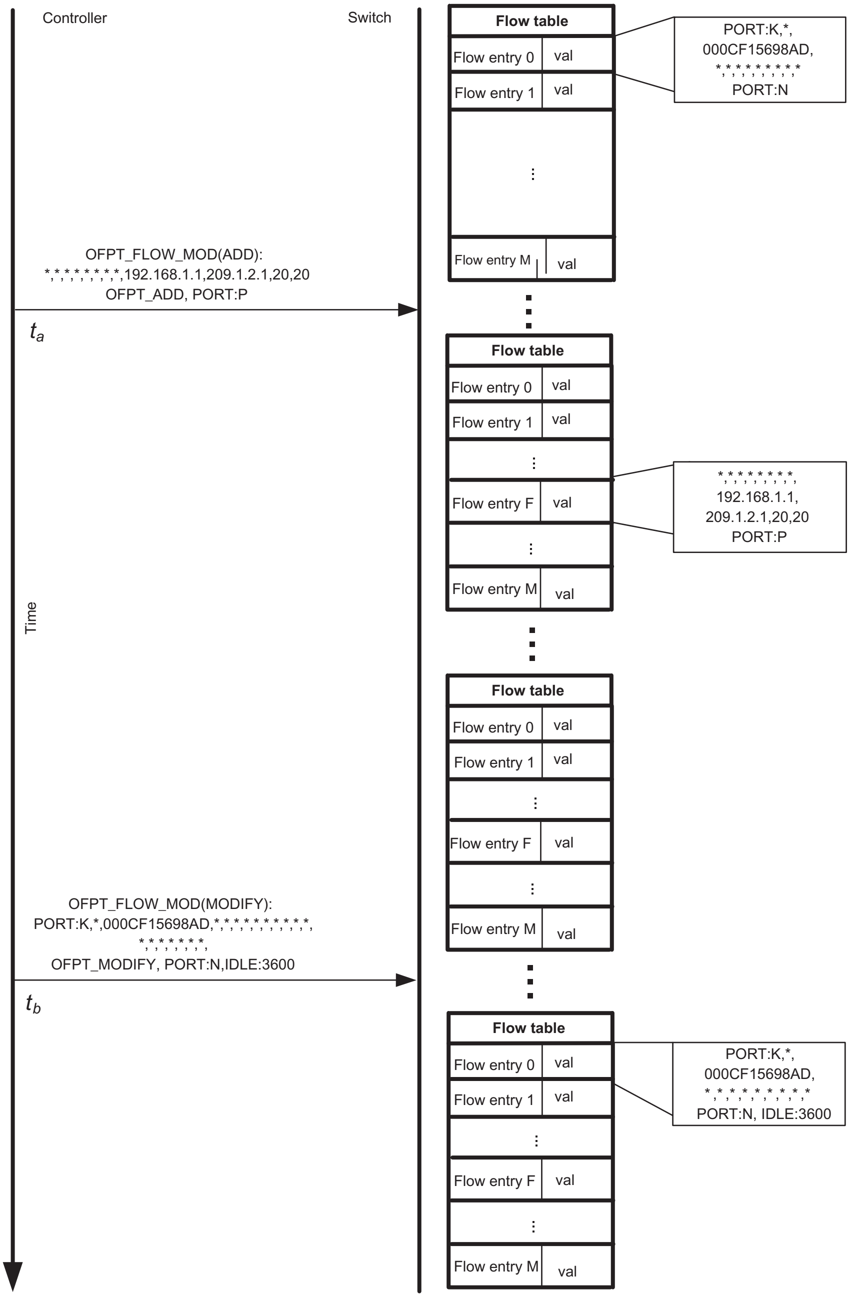

FIG. 5.8

Controller programming flow entries in V.1.0.

5.3.7 EXAMPLE: BASIC PACKET FORWARDING

We illustrate the most basic case of OpenFlow V.1.0 packet forwarding in Fig. 5.9. The figure depicts a packet arriving at the switch through port 2 with source IPv4 address of 192.168.1.1 and destination IPv4 address of 209.1.2.1. The packet matching function scans the flow table starting at flow entry 0 and finds a match in flow entry *F*. Flow entry *F* stipulates that a matching packet should be forwarded out port *P*. The switch does this, completing this simple forwarding example.

An OpenFlow switch forwards packets based on the header fields it matches. The network programmer designates layer three switch behavior by programming the flow entries to try to match layer three headers such as IPv4. If it is a layer two switch, the flow entries will dictate matching on layer two headers. The semantics of the flow entries allow matching for a wide variety of protocol headers but a given switch will be programmed only for those that correspond to the role it plays in packet forwarding. Whenever there is overlap in potential matches of flows, the priority assigned the flow entry by the controller will determine which match takes precedence. For example, if a switch was both a layer two and a layer three switch, placing layer three header matching flow entries at a higher priority would ensure that if possible, layer three switching is done on that packet.

5.3.8 EXAMPLE: SWITCH FORWARDING PACKET TO CONTROLLER

We showed in the last section an example of an OpenFlow switch forwarding an incoming data packet to a specified destination port. Another fundamental action of the OpenFlow V.1.0 switch is to forward packets to the controller for exception handling. The two reasons for which the switch may forward a packet to the controller are OFPR_NO_MATCH and OFPR_ACTION. Obviously OFPR_NO_MATCH is used when no matching flow entry is found. OpenFlow retains the ability to specify that a particular matching flow entry should always be forwarded to the controller. In this case OFPR_ACTION is specified as the reason. An example of this would be a control packet such as a routing protocol packet that always needs to be processed by the controller. In Fig. 5.10 we show an example of an incoming data packet that is an OSPF routing packet. There is a matching table entry for this packet which specifies that the packet should be forwarded to the controller. We see in the figure that a PACKET_IN

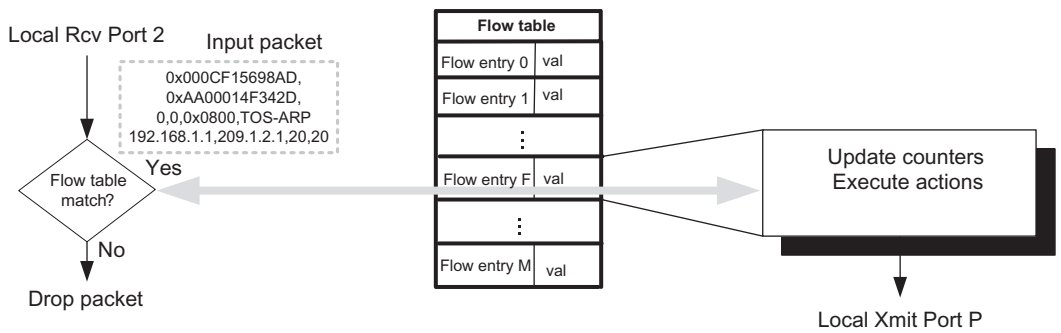

FIG. 5.9

Packet matching function—basic packet forwarding V.1.0.

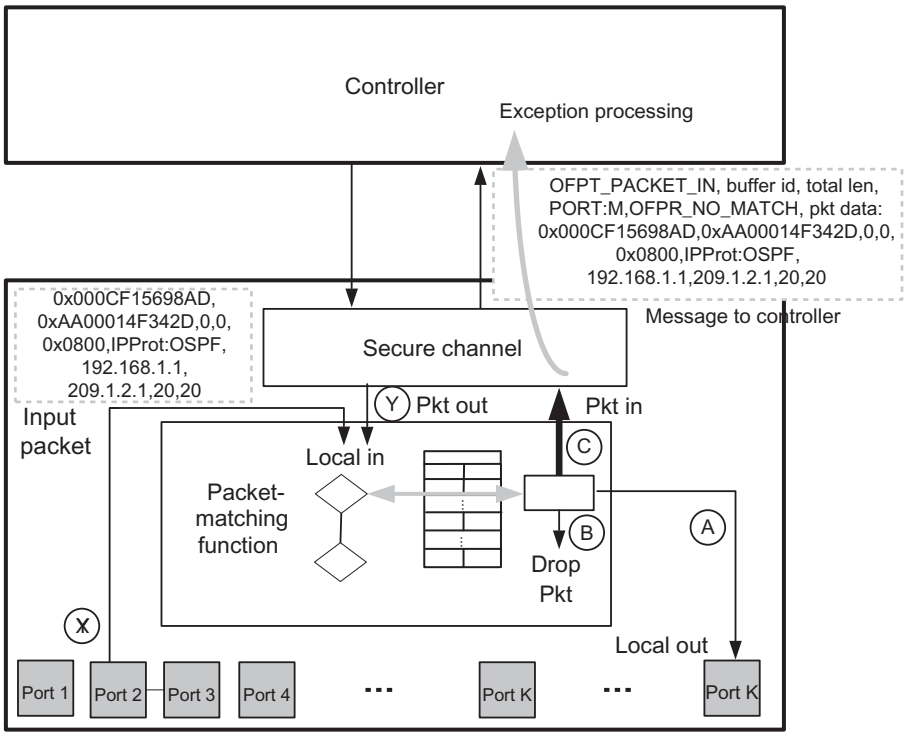

FIG. 5.10

Switch forwarding incoming packet to controller.

message is sent via the secure channel to the controller, handing off this routing protocol update to the controller for exception processing. The processing that would likely take place on the controller is that the OSPF routing protocol would be run, potentially resulting in a change to the forwarding tables in the switch. The controller could then modify the forwarding tables via the brute force approach of sending FLOW_MOD commands to the switch modifying the output port for each flow in the switch affected by this routing table change. (We will see in Section 5.4.2 that there is a more efficient way for the controller to program the output port for flows in a layer three switch where multiple flows share the same next hop IP address.)

At a minimum, the controller needs access to the packet header fields to determine its disposition of the packet. In many cases, though not all, it may need access to the entire packet. This would in fact be true in the case of the OSPF routing packet in this example. In the interest of efficiency, OpenFlow allows the optional buffering of the full packet by the switch. In the event of a large number of packets being forwarded from the switch to the controller for which the controller only needs to examine the packet header, significant bandwidth efficiency gains are achieved by buffering the full packet in the switch and only forwarding the header fields. Since the controller will sometimes need to see the balance of the packet, a buffer ID is communicated with the PACKET_IN message. This buffer

ID may be used by the controller to subsequently retrieve the full packet from the switch. The switch has the ability to age out old buffers that have not been retrieved by the controller.

There are other fundamental actions that the switch may take on an incoming packet: (1) to flood the packet out all ports except the port on which it arrived or, (2) to drop the packet. We trust that the reader's understanding of the drop and flood functions will follow naturally from the two examples just provided, and we'll now move on to discuss the extensions to the basic OpenFlow functions provided in V.1.1.

DISCUSSION QUESTION:

Explain how the controller would use the BUFFER ID field in the message sent to it shown in Fig. 5.10.

5.4 OpenFlow 1.1 ADDITIONS

OpenFlow 1.1 [3] was released on February 28, 2011. Table 5.2 lists the major new features added in this release. Multiple flow tables and group table support are the two most prominent features added to OpenFlow in release V.1.1. We highlight these new features in Fig. 5.11. From a practical standpoint, V.1.1 had little impact other than as a stepping stone to V.1.2. This was because it was released just before the ONF was created and the SDN community waited for the first version following the ONF transition (i.e., V.1.2) before creating implementations. It is important that we cover it here, though, as the following versions of OpenFlow are built upon some of the V.1.1 feature set.

5.4.1 MULTIPLE FLOW TABLES

V.1.1 significantly augments the sophistication of packet processing in OpenFlow. The most salient shift is due to the addition of *multiple flow tables*. The concept of a single flow table remains much like it was in V.1.0, however, in V.1.1 it is now possible to defer further packet processing to subsequent matching in other flow tables. For this reason, it was necessary to break the execution of actions from its direct association with a flow entry. With V.1.1 the new *instruction* protocol object is associated with a flow entry. The processing pipeline of V.1.1 offered much greater flexibility than what was available in V.1.0. This improvement derives from the fact that flow entries can be *chained* by an instruction

Table 5.2 Major New Features Added in OpenFlow 1.1

Feature Name	Description
Multiple Flow Tables	See Section 5.4.1
Groups	See Section 5.4.2
MPLS and VLAN Tag Support	See Section 5.4.3
Virtual Ports	See Section 5.4.4
Controller Connection Failure	See Section 5.4.5

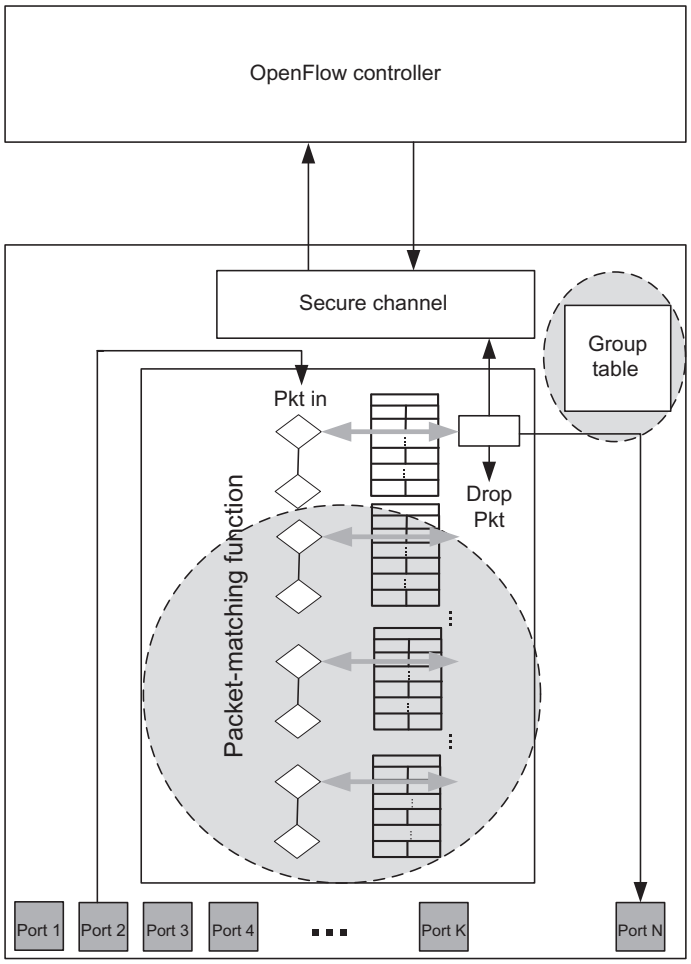

FIG. 5.11

OpenFlow V.1.1 switch with expanded packet processing.

in one flow entry pointing to another flow table. This is called a *GOTO* instruction. When such an instruction is executed, the packet matching function depicted earlier in Figs. 5.9 and 5.10 is invoked again, this time starting the match process with the first flow entry of the new flow table. This new pipeline is reflected in the expanded packet matching function shown in Fig. 5.11. The pipeline allows all the power of an OpenFlow V.1.1 instruction to test and *modify* the contents of a packet to be applied multiple times, with different conditions used in the matching process in each flow table. This allows dramatic increases in both the complexity of the matching logic as well as the degree and nature of the packet modifications that may take place as the packet transits the OpenFlow V.1.1 switch. V.1.1

packet processing, with different instructions chained through complex logic as different flow entries are matched in a sequence of flow tables, constitutes a robust packet processing *pipeline*.[1]

In Section 5.3.4 we described the actions supported in V.1.0 and the order in which they were executed. The V.1.1 instructions form the conduit that controls which actions are taken and in what order. They can do this in two ways. In the first, they add actions to an *action-set*. The action set is initialized and modified by all the instructions executed during a given pass through the pipeline. Instructions delete actions from the action-set or merge new actions into the action-set. When the pipeline ends, the actions in the action-set are executed in the following order:

1. copy TTL inwards
2. pop
3. push
4. copy TTL outwards
5. decrement TTL
6. set: apply all set_field actions to the packet
7. qos: apply all QoS actions to the packet, such as set_queue
8. group: if a group action is specified, apply the actions to the relevant action buckets (which can result in the packet being forwarded out the ports corresponding to those buckets)
9. output: if no group action was specified, forward the packet out the specified port

The output action, if such an instruction exists (and it normally does), is executed last. This is logical because other actions often manipulate the contents of the packet and this clearly must occur before the packet is transmitted. If there is neither a group nor output action, the packet is dropped. We will explain the most important of the actions listed above in discussion and examples in the following sections.

The second way that instructions can invoke actions is via the *Apply-Actions* instruction. This is used to execute certain actions immediately between flow tables, while the pipeline is still active, rather than waiting for the pipeline to reach its end, which is the normal case. Note that the action list that is part of the Apply-Actions has exactly the same semantics as the action list in the PACKET_OUT message, and both are the same as the action list semantics of V.1.0.

In the case of the PACKET_OUT message being processed through the pipeline, rather than the action list being executed upon a flow entry match, its component actions on the action list are merged into the action set being constructed for this incoming packet. If the current flow entry includes a GOTO instruction to a higher-numbered flow table, then further action lists may be merged into the action set before it is ultimately executed.

When a matched flow entry does not specify a GOTO flow table, the pipeline processing completes, and whatever actions have been recorded in the action set are then executed. In the normal case, the final action executed is to forward the packet to an output port, to the controller or to a *group* table, which we describe below.

[1] This increased robustness comes at a price. It has proven challenging to adapt existing hardware switches to support multiple flow tables. We discuss this further in Section 5.6. In Sections 14.9.3 and 15.3.7 we discuss attempts to design hardware specifically designed with this capability in mind.

5.4.2 GROUPS

V.1.1 offers the *group* abstraction as a richer extension to the FLOOD option. In V.1.1 there is the group table, which first appeared in Fig. 5.11 and is shown in detail in Fig. 5.12. The group table consists of *group entries*, each entry consisting of one or more *action buckets*. The buckets in a group have actions associated with them that get applied before the packet is forwarded to the port defined by that bucket. Refinements on flooding, such as multicast, can be achieved in V.1.1 by defining groups as specific sets of ports. Sometimes a group may be used when there is only a single output port, as illustrated in the case of group 2 in Fig. 5.12. A use case for this would be when many flows all should be directed to the same next hop switch and it is desirable to retain the ability to change that next hop with a single configuration change. This can be achieved in V.1.1 by having all the designated flows pointing to a single group entry that forwards to the single port connected to that next hop. If the controller wishes to change that next hop due to a change in IP routing tables in the controller, all the flows can

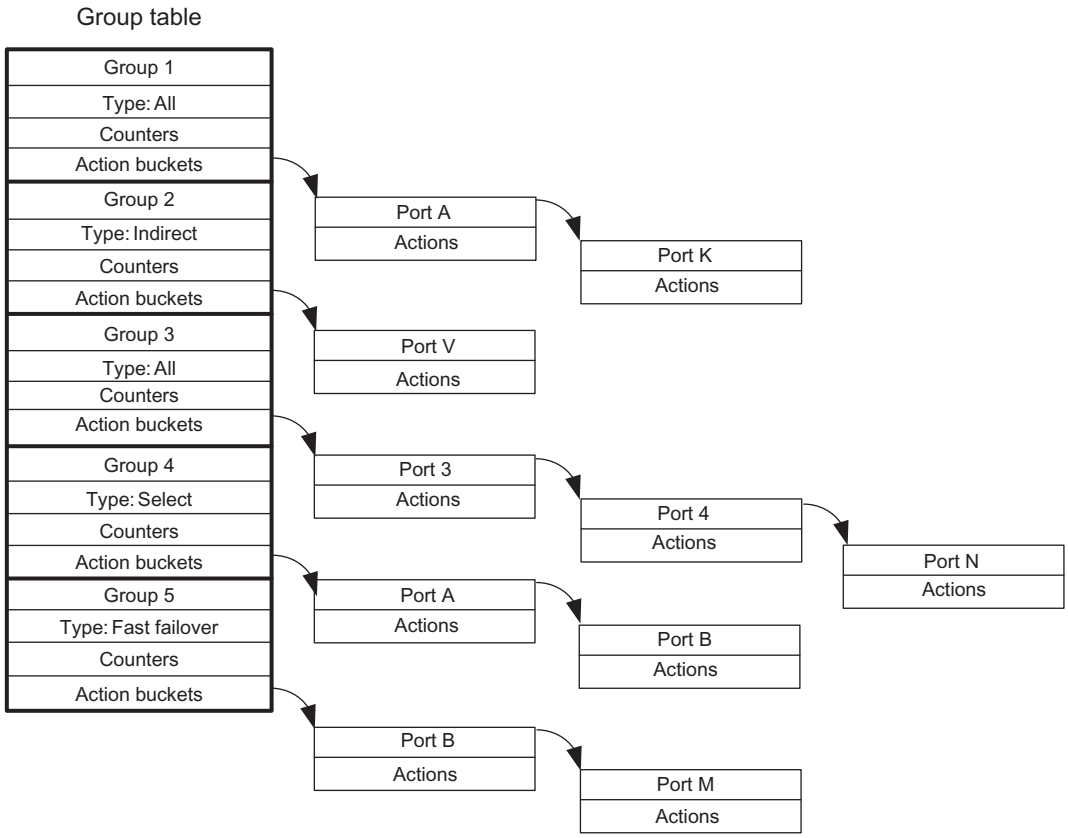

FIG. 5.12

OpenFlow V.1.1 group table.

be rerouted by simply reprogramming the single group entry. This provides a more efficient way of handling the routing change from the OSPF update example presented earlier in Section 5.3.8. Clearly, changing a single group entry's action bucket is faster than updating the potentially large number of flow entries whose next hop changed as a result of a single routing update. Note that this is a fairly common scenario. Whenever a link or neighbor fails or is taken out of operation, *all flows* traversing that failed element need to be rerouted by the switch that detects that failure.

Note that one group's buckets may forward to other groups, providing the capability to chain groups together.

5.4.3 MPLS AND VLAN TAG SUPPORT

V.1.1 is the first OpenFlow version to provide full VLAN support. Since providing complete support for multiple levels of VLAN tags required robust support for the *popping* and *pushing* of multiple levels of tags, support for MPLS tagging followed naturally and is also part of V.1.1. For the interested reader unfamiliar with MPLS technology, we provide a good reference in [4]. Both the new PUSH and POP actions, as well as the chaining of flow tables, were necessary to provide this generic support for VLANs and MPLS. When a PUSH action is executed, a new header of the specified type is inserted in front of the current outermost header. The field contents of this new header are initially copied from the corresponding existing fields in the current outermost header, should one exist. If it does not exist, they are initialized to zero. New header values are then assigned to this new outermost header via subsequent SET actions. While the PUSH is used to add a new tag, the POP is used to remove the current outermost tag. In addition to PUSH and POP actions, V.1.1 also permits modification of the current outermost tag, whether it be a VLAN tag or an MPLS shim header.

While V.1.1 correctly claims full MPLS and VLAN tag support, that support requires specification of very complex matching logic that has to be implemented when such tags are encountered in the incoming packet. The *extensible match support* that will be introduced in Section 5.5.1 provides the more generalized semantics to the controller such that this complex matching logic in the switch can be disposed of. Since this V.1.1 logic is replaced by the more general solution in Section 5.5.1, we will not confuse the reader by providing details about it here.

5.4.4 VIRTUAL PORTS

In V.1.0, the concept of an output port mapped directly to a physical port, with some limited use of *virtual ports*. While the TABLE and other virtual ports did exist in earlier versions of OpenFlow, the concept of virtual ports has been augmented in V.1.1. A V.1.1 switch groups ports into the categories of *standard ports* and *reserved virtual ports*. Standard ports consist of:

- Physical Ports
- Switch-Defined Virtual Ports: In V.1.1 it is now possible for the controller to forward packets to an abstraction called Switch-Defined Virtual Ports. Such ports are used when more complex processing will be required on the packet than simple header field manipulation. One example of this is when the packet should be forwarded via a tunnel. Another use case for a virtual port is *Link Aggregation* (LAG). For more information about LAGs, we refer the reader to [5].

Reserved virtual ports consist of:

- ALL: This is the straightforward mechanism to flood packets out all standard ports except the port on which the packet arrived. This is similar to the effect provided by the optional FLOOD reserved virtual port below.
- CONTROLLER: Forwards the packet to the controller in an OpenFlow message.
- TABLE: Processes the packet through the normal OpenFlow pipeline processing. This only applies to a packet that is being sent from the controller (via a PACKET_OUT message). We explained the use of the TABLE virtual port in Section 5.3.4.
- IN_PORT: This provides a loopback function. The packet is sent back out on the port on which it arrived.
- LOCAL (optional): This optional port provides a mechanism where the packet is forwarded to the switch's local OpenFlow control software. As a LOCAL port may be used both as an output port and an ingress port, this can be used to implement an in-band controller connection, obviating the need for a separate control network for the controller-switch connection. We refer the reader to our earlier discussion on LOCAL in Section 5.3.3 for an example of how LOCAL may be used.
- NORMAL (optional): This directs the packet to the normal nonOpenFlow pipeline of the switch. This differs from the LOCAL reserved virtual port in that it may only be used as an output port.
- FLOOD (optional): The general use of this port is to send the packet out all standard ports except the port on which it arrived. The reader should refer to the OpenFlow V.1.1 specification for the specific nuances of this reserved virtual port.

5.4.5 CONTROLLER CONNECTION FAILURE

Loss of connectivity between the switch and controller is a serious, and real possibility, and the OpenFlow specification needs to specify how it should be handled. The *emergency flow cache* was included in V.1.0 in order to handle such a situation, but support for this was dropped in V.1.1 and replaced with two new mechanisms, *fail secure mode* and *fail stand-alone mode*.[2] The V.1.1 switch immediately enters one of these two modes upon loss of connection to the controller. Which of these two modes is entered will depend upon which is supported by the switch or, if both are supported, by user configuration. In the case of fail secure mode, the switch continues to operate as a normal V.1.1 switch, except that all messages destined for the controller are dropped. In the case of fail stand-alone mode, the switch additionally ceases its OpenFlow pipeline processing and continues to operate in its native, underlying switch or router mode. When the connection to the controller is restored, the switch resumes its normal operation mode. The controller, having detected the loss and restoration of the connection, may choose to delete existing flow entries and begin to configure the switch anew. (*Author's note: This is another example of the aforementioned lack of backwards compatibility in the OpenFlow specifications.*)

[2] A common synonym for fail secure mode is *fail open mode*. Similarly, a synonym for fail stand-alone mode is *fail closed mode*.

5.4.6 EXAMPLE: FORWARDING WITH MULTIPLE FLOW TABLES

Fig. 5.13 expands on our earlier example of V.1.0 packet forwarding that was presented in Fig. 5.9. Assuming that the incoming packet is the same as in Fig. 5.9, we see in Fig. 5.13 that there is a match in the second flow entry in flow table 0. Unlike V.1.0, the pipeline does not immediately execute actions associated with that flow entry. Its counters are updated, and the newly initialized action-set is updated with those actions programmed in that flow entry's instructions. One of those instructions is to continue processing at table *K*. We see this via the jump to processing at the letter *A*, where we resume the pipeline process matching the packet against table *K*. In this case, a matching flow entry is found in flow entry *F*. Once again, this match results in the flow entry's counters being updated and its instructions being executed. These instructions may apply further modifications to the action set that has been carried forward from table 0. In our example, an additional action *A*3 is merged into the action set. Since there is no GOTO instruction present this time, this represents the end of the pipeline processing and the actions in the current action set are performed in the order specified by OpenFlow V.1.1. (The difference between action set and action-list was explained earlier in Section 5.4.1.) The OUTPUT action, if it is present, is the last action performed before the pipeline ends. We see that the OUTPUT action was present in our example as we see the packet forwarded out port *P* in Fig. 5.13.

5.4.7 EXAMPLE: MULTICAST USING V.1.1 GROUPS

We direct the reader to Fig. 5.12 discussed previously. In this figure we see that group 3 has group type ALL and has three action buckets. This group configuration prescribes that a packet sent for processing by this group entry should have a copy sent out each of the ports in the three action buckets. Fig. 5.14 shows that the packet matching function directs the incoming packet to group 3 in the group table for processing. Assuming that the group 3 in Fig. 5.14 is as is shown in Fig. 5.12, the packet is multicast out ports 3, 4, and *N* as indicated by the dashed arrows in the figure.

DISCUSSION QUESTION:

The group table depicted in Fig. 5.14 is the one shown earlier in Fig. 5.12. Explain why, in Fig. 5.14, there appear three copies of the incoming packet being forwarded out ports 3, 4, and N, respectively.

5.5 OpenFlow 1.2 ADDITIONS

OpenFlow 1.2 [6] was released on December 5, 2011. Table 5.3 lists the major new features added in this release. We discuss the most important of these in the following sections.

5.5.1 EXTENSIBLE MATCH SUPPORT

Due to the relatively narrow semantics in the packet matching prior to V.1.2, it was necessary to define complex flow diagrams that described the logic of how to perform packet parsing. The packet matching capability provided in V.1.2 provides sufficient richness in the packet matching descriptors that the

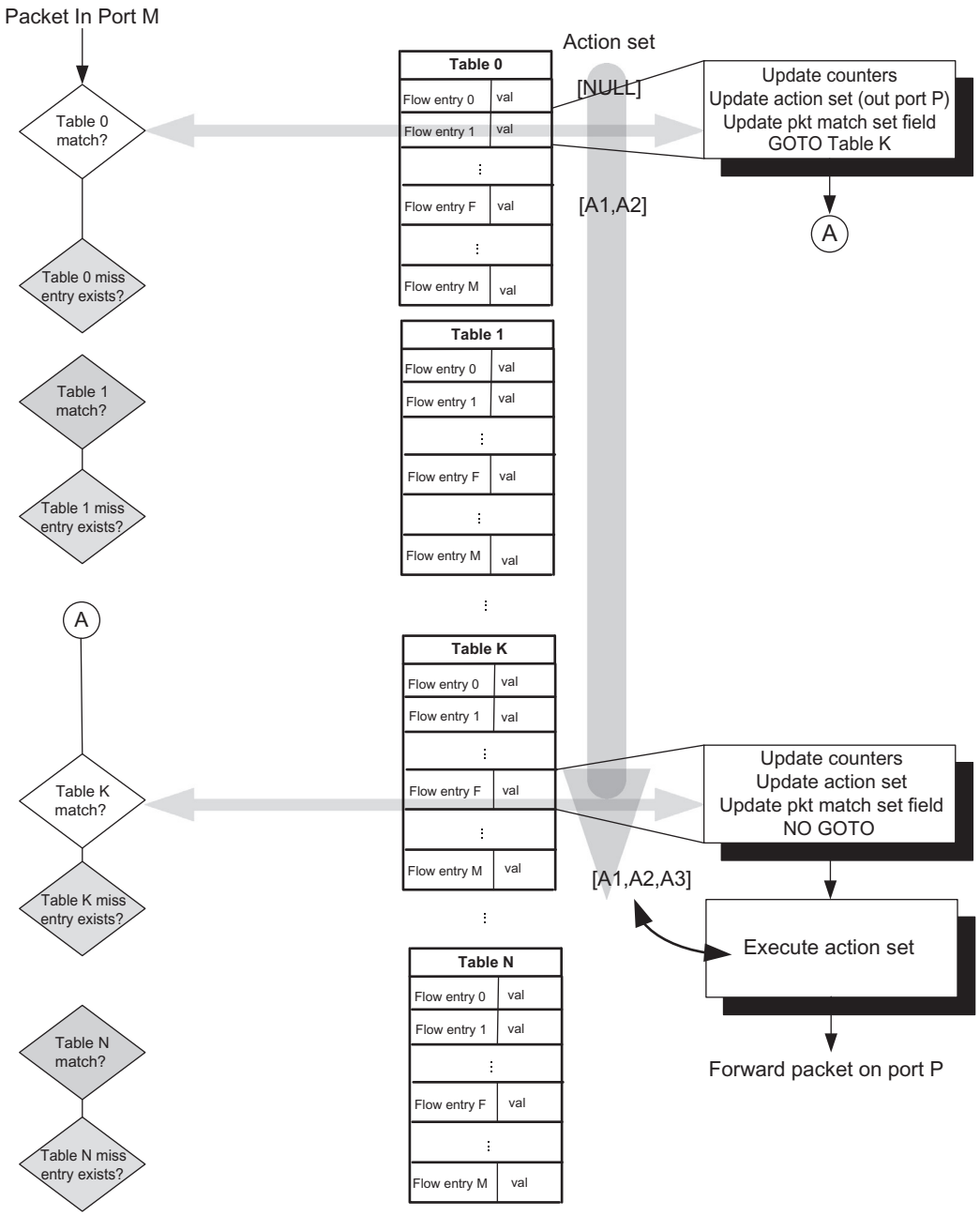

FIG. 5.13

Packet matching function—basic packet forwarding V.1.1.

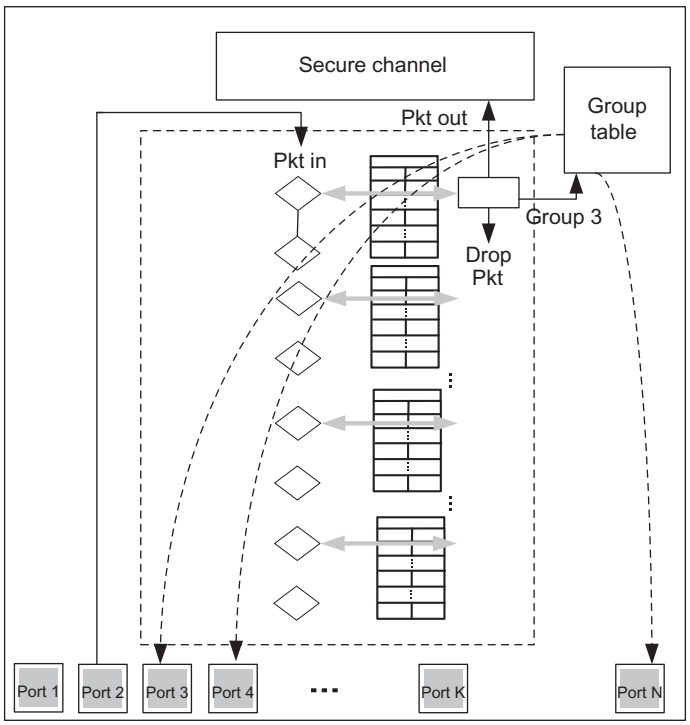

FIG. 5.14

Multicast using group table in V.1.1.

Table 5.3 Major New Features Added in OpenFlow 1.2

Feature Name	Description
Extensible Match Support	See Section 5.5.1
Extensible set_field Packet Rewriting Support	See Section 5.5.2
Extensible Context Expression in "packet-in"	See Section 5.5.3
Extensible Error Messages via Experimenter Error Type	See [6]
IPv6 Support	See Section 5.5.2
Simplified Behavior of Flow-Mod Request	See [6]
Removed Packet Parsing Specification	See Section 5.5.1
Multiple Controller Enhancements	See Section 5.5.4

controller can encode the desired logic in the rules themselves. This obviates the earlier versions' requirement that the behavior be hard-coded into the switch logic.

A generic and extensible packet matching capability has been added in V.1.2 via the *OpenFlow Extensible Match* (OXM) descriptors. OXM defines a set of *type-length-value* (TLV) pairs that can

describe or define virtually any of the header fields an OpenFlow switch would need to use for matching. This list is too long to enumerate here, but in general any of the header fields that are used in matching for Ethernet, VLAN, MPLS, IPv4, and IPv6 switching and routing may be selected and a value (with bitmask wildcard capability) provided. Prior versions of OpenFlow had a much more static match descriptor which limited flexibility in matches and made adding future extensions more difficult. An example of this is that in order to accommodate the earlier fixed structure, the single match field TCP Port was overloaded to also mean UDP Port or ICMP code depending on the context. This confusion has been eliminated in V.1.2 via OXM. Because the V.1.2 controller can encode more descriptive parsing into the pipeline of flow tables, complex packet parsing does not need to be specified inside the V.1.2 switch.

This ability to match on any combination of header fields is provided within the OPENFLOW_BASIC *match class*. V.1.2 expands the possibilities for match fields by allowing for multiple match classes. Specifically, the EXPERIMENTER match class is defined, opening up the opportunity for matching on fields in the packet payload, providing a near limitless horizon for new definitions of flows. The syntax and semantics of EXPERIMENTER are left open so that the number and nature of different fields is subject to the experimental implementation.

5.5.2 EXTENSIBLE SET_FIELD PACKET REWRITING SUPPORT

The V.1.2 switch continues to support the action types we saw in previous versions. A major enhancement provides for the ability to set the value of *any* field in the packet header that may be used for matching. This is due to the fact that the same OXM encoding described above for enhanced packet matching is made available in V.1.2 for generalized setting of fields in the packet header. For example, IPv6 support falls out naturally from the fact that any of the fields that may be described by an OXM TLV may also be set using the set_field action. The ability to match an incoming IPv6 header against a flow entry and, when necessary, to set an IPv6 address to a new value, together provide support for this major feature of V.1.2. CRC recalculation is performed automatically when a set-field action changes the packet's contents.

Since the EXPERIMENTER action type can be a modification or extension to any of the basic action types, or even something totally novel, this allows for packet fields not part of the standard OXM header fields to be modified. In general, if a packet field may be used for matching in V.1.2, then it may also be modified.

5.5.3 EXTENSIBLE CONTEXT EXPRESSION IN PACKET_IN

The OXM encoding is also used to extend the PACKET_IN message sent from the switch to the controller. In previous versions this message included the packet headers used in matching that resulted in the switch deciding to forward the packet to the controller. In addition to the packet contents, the packet matching decision is influenced by *context* information. Formerly, this consisted of the input port identifier. In V.1.2 this context information is expanded to include the input virtual port, the input physical port, and *metadata* that has been built up during packet matching pipeline processing. Metadata semantics is not prescribed by the OpenFlow specification, other than that the OXM metadata TLV may be initialized, modified or tested at any stage of the pipeline processing. Since the OXM encoding described above contains TLV definitions for all the context fields, as well as all matchable

packet headers, the OXM format provides a convenient vehicle to communicate the packet matching state when the switch decides to forward the packet to the controller. The switch may forward a packet to the controller because:

- there was no matching flow or
- an instruction was executed in the pipeline prescribing that a matching packet be forwarded to the controller or
- the packet had an invalid TTL.

5.5.4 MULTIPLE CONTROLLERS

In previous versions of OpenFlow, there was very limited support for backup controllers. In the event that communication with the current controller was lost, the switch entered either fail secure mode or fail stand-alone mode. The notion that the switch would attempt to contact previously configured backup controllers was encouraged by the specification, but not explicitly described. In V.1.2 the switch may be configured to maintain simultaneous connections to multiple controllers. The switch must ensure that it only sends messages to a controller pertaining to a command sent by that controller. In the event that a switch message pertains to multiple controllers, it is duplicated and a copy sent to each controller. A controller may assume one of three different roles relative to a switch:

- Equal
- Slave
- Master

These terms relate to the extent to which the controller has the ability to change the switch configuration. The least powerful is obviously slave mode, wherein the controller may only request data from the switch, such as statistics, but may make no modifications. Both equal and master modes allow the controller the full ability to program the switch, but in the case of master mode the switch enforces that only one controller be in master mode and all others are in slave mode. The multiple controllers feature is part of how OpenFlow addresses the *high availability* (HA) requirement.

5.5.5 EXAMPLE: BRIDGING VLANS THROUGH SP NETWORKS

V.1.2 is the first release to provide convenient support for bridging customer VLANs through service provider (SP) networks. This feature is known by a number of different names, including *provider bridging*, *Q-in-Q* and *Stacked VLANs*. These all refer to the tunneling of an edge VLAN through a provider-based VLAN. There are a number of variants of this basic technology, but they are most often based on the nesting of multiple IEEE 802.1Q tags [7]. In this example we will explain how this can be achieved by stacking one VLAN tag inside another, which involves the PUSHing and POPping of VLAN tags that was introduced in Section 5.4.3. We deferred this example until our discussion of V.1.2 since the V.1.1 packet matching logic to support this was very complex and not easily extended. With the extensible match support in V.1.2, the logic required to provide VLAN stacking in the OpenFlow pipeline is much more straightforward.

This feature is implemented in OpenFlow V.1.2 by encoding a flow entry to match a VLAN tag from a particular customer's layer two network. In our example, this match occurs at an OpenFlow switch

serving as a service provider access switch. This access switch has been programmed to match VLAN 10 from site A of customer X and to tunnel this across the service provider core network to the same VLAN 10 at site B of this same customer X. The OpenFlow pipeline will use the extensible match support to identify frames from this VLAN. The associated action will PUSH service provider VLAN tag 40 onto the frame, and the frame will be forwarded out a port connected to the provider backbone network. When this doubly tagged frame emerges from the backbone, the OpenFlow-enabled service provider access switch matches VLAN tag 40 from that backbone interface, and the action associated with the matching flow entry POPs off the outer tag and the singly tagged frame is forwarded out a port connected the site B of customer X.

Note that while our example uses VLAN tags, similar OpenFlow processing can be used to tunnel customer MPLS connections through MPLS backbone connections. In that case, an outer MPLS label is PUSHed onto the packet and then POPped off upon exiting the backbone.

5.6 OpenFlow 1.3 ADDITIONS

OpenFlow V.1.3 [8] was released on April 13, 2012. This release was a major milestone. Many implementations are being based on V.1.3 in order to stabilize controllers around a single version. This is also true about ASICs. Since V.1.3 represents a major new leap in functionality and has not been followed quickly by another release, this provides an opportunity for ASIC designers to develop hardware support for many of the V.1.3 features with the hope of a more stable market into which to sell their new chips. This ASIC opportunity notwithstanding, there are post V.1.0 features that are very challenging to implement in hardware. For example, the iterative matching actions we described in Section 5.4.1 are difficult to implement in hardware at line rates. It is likely that the real-life chips that support V.1.3 will have to limit the number of flow tables to a manageable number.[3] Such limitations are not imposed on software-only implementations of OpenFlow, and, indeed, such implementations will provide full V.1.3 support. Table 5.4 lists the major new features added in this release. A discussion of the most prominent of these new features is found in the following sections.

5.6.1 REFACTOR CAPABILITIES NEGOTIATION

There is a new MULTIPART_REQUEST/MULTIPART_REPLY message pair in V.1.3. This replaces the READ_STATE messaging in prior versions that used STATS_REQUEST/STATS_REPLY to get this information. Rather than having this capability information embedded as if it were a table statistic, the new message request-reply pair is utilized and the information is conveyed using a standard *type-length-value* (TLV) format. Some of the capabilities that can be reported in this new manner include *next-table*, *table-miss flow entry*, and *experimenter*. This new MULTIPART_REQUEST/REPLY message pair subsumes the older STATS_REQUEST/REPLY pair and is now used for both reporting of statistics as well as capability information. The data formats for the capabilities are the TLV format and this capability information has been removed from the table statistics structure.

[3]We discuss the possibility of ASICs that would allow more general support of these features in Section 14.9.3 and Section 15.3.7.

Table 5.4 Major New Features Added in OpenFlow 1.3

Feature Name	Description
Refactor Capabilities Negotiation	See Section 5.6.1
More Flexible Table Miss Support	See Section 5.6.2
IPv6 Extension Header Handling Support	See [8]
Per Flow Meters	See Section 5.6.3
Per Connection Event Filtering	See Section 5.6.4
Auxiliary Connections	See Section 5.6.5
MPLS BoS Matching	Bottom of Stack bit (BoS) from the MPLS header may now be used as part of match criteria
Provider Backbone Bridging Tagging	See Section 5.6.7
Rework Tag Order	Instruction execution order now determines tag order rather than it being statically specified
Tunnel-ID Metadata	Allows for support of multiple tunnel encapsulations
Cookies in PACKET_IN	See Section 5.6.6
Duration for Stats	The new *duration* field allows more accurate computation of packet and byte rate from the counters included in those statistics.
On Demand Flow Counters	Disable/enable packet and byte counters on a per-flow basis

5.6.2 MORE FLEXIBLE TABLE-MISS SUPPORT

We have explained that a table-miss is when a packet does not match any flow entries in the current flow table in the pipeline. Formerly there were three configurable options for handling such a table-miss. This included dropping the packet, forwarding it to the controller or continuing packet matching at the next flow table. V.1.3 expands upon this limited handling capability via the introduction of the *table-miss flow entry*. The controller programs a table-miss flow entry into a switch in much the same way it would program a normal flow entry. The table-miss flow entry is distinct in that it is by definition of lowest priority (zero) and all match fields are wildcards. The zero-priority characteristic guarantees that it is the last flow entry that can be matched in the table. The fact that all match fields are wild cards means that *any* packet being matched against the table-miss will be a match. The upshot of these characteristics is that the table-miss entry serves as a kind of backstop for any packets that would otherwise have found no matches. The advantage of this approach is that the full semantics of the V.1.3 flow entry, including instructions and actions, may be applied to the case of a table-miss. It should be obvious to the reader that the table-miss base cases of dropping the packet, forwarding it to the controller, or continuing at next flow table are easily implemented using the flow entry instructions and actions. Interestingly, though, by using the generic flow entry semantics, it is now conceivable to treat table-misses in more

sophisticated ways, including passing the packet that misses to a higher-numbered flow table for further processing. This capability adds more freedom to the matching "language" embodied by the flow tables, entries and associated instructions and actions.

5.6.3 PER FLOW METERS

V.1.3 introduces a flexible meter framework. Meters are defined on a per-flow basis and reside in a *meter table*. Fig. 5.15 shows the basic structure of a meter and how it is related to a specific flow. A given meter is identified by its *meter ID*. V.1.3 instructions may direct packets to a meter identified by its meter ID. The framework is designed to be extensible to support the definition of complex meters in future OpenFlow versions. Such future support may include color-aware meters that will provide DiffServ QoS capabilities. V.1.3 meters are only rate-limiting meters. As we see in Fig. 5.15 there may be multiple *meter bands* attached to a given meter. Meter 3 in the example in the figure has three meter bands. Each meter band has a configured bandwidth rate and a type. Note that the units of bandwidth are not specified by OpenFlow. The type determines the action to take when that meter band is processed. When a packet is processed by a meter, at most one band is used. This band is selected based on the highest bandwidth rate band that is lower than the current measured bandwidth for that flow. If the current measured rate is lower than all bands, no band is selected and no action is taken. If a band is selected, the action taken is that prescribed by the band's type field. There are no required types in V.1.3. The *optional* types described in V.1.3 consist of DROP and *DSCP remark*. DSCP remark indicates that the DiffServ drop precedence field of the *Differentiated Services Code Point* (DSCP)

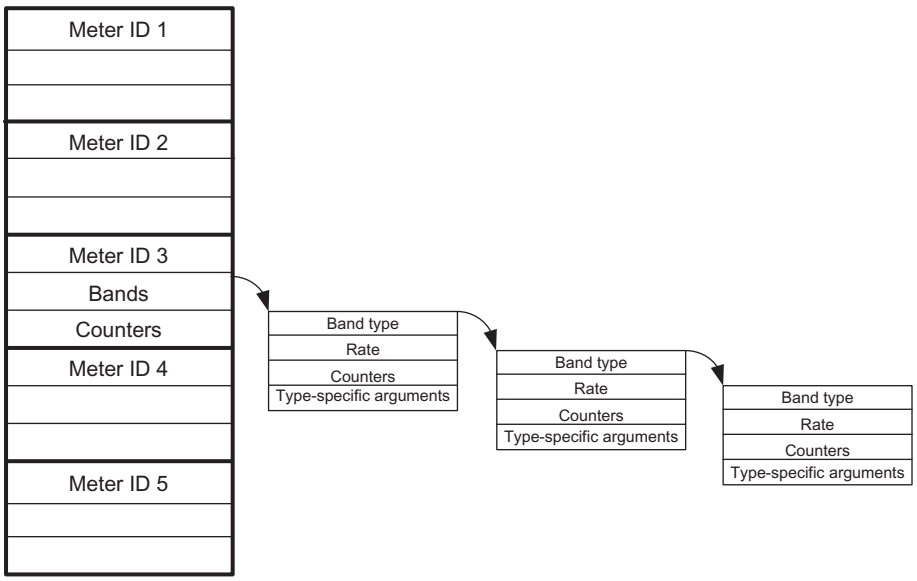

FIG. 5.15

OpenFlow V.1.3 meter table.

field should be decremented, increasing the likelihood that this packet will be dropped in the event of queue congestion. Well known and simple rate-limiting algorithms, such as the *leaky bucket* [9], may be implemented in a straightforward manner under this framework. This feature provides direct QoS control at the flow level to the OpenFlow controller by careful programming of the meters.

As seen in Fig. 5.15 there are meter-level counters and meter-band-level counters. The meter counters are updated for all packets processed by the meter and the per-meter-band counters are updated only when that particular band is selected. The dual level of counters is key in that presumably the majority of the packets processed by the meter suffer no enforcement, but their passage through the meter must be recorded in order to track the current measured rate. The reader should understand now that when a packet is processed by a band, it has exceeded a bandwidth threshold, which results in some kind of enforcement (e.g., dropping of the packet or marking it as *drop eligible* via DSCP remark).

5.6.4 PER CONNECTION EVENT FILTERING

We saw in V.1.2 the introduction of the notion of multiple controllers. Multiple controllers, able to take on different roles, achieve an improved level of fault tolerance and load balancing. The earlier discussion explained that the switches would only provide replies to the controller that initiated the request that elicits that reply. This, however, did not prescribe how to damp down the communication of asynchronous messages from the switch to its family of controllers. The idea that the different controllers have asymmetrical roles was not as effective as it could have been if all controllers must receive the same kind and quantity of asynchronous notifications from the switches. For example, a slave controller may not want to receive all types of asynchronous notifications from the switch. V.1.3 introduces a SET_ASYNC message that allows the controller to specify which sorts of async messages it is willing to receive from a switch. It additionally allows the controller to filter out certain reason codes that it does not wish to receive. A controller may use two different filters, one for the master/equal role and another for the slave role. Note that this filter capability exists in addition to the ability to enable or disable asynchronous messages on a per-flow basis. This new capability is controller-oriented rather than flow-oriented.

5.6.5 AUXILIARY CONNECTIONS

We have already seen that earlier releases of OpenFlow allowed some parallelism in the switch-controller channel via the use of multiple controllers. In this way, multiple parallel communication *channels* existed from a single switch to multiple controllers. V.1.3 introduces an additional layer of parallelism by allowing multiple *connections* per communications channel. That is, between a single controller and switch, multiple connections may exist. We depict these multiple connections between the switch and the MASTER controller in Fig. 5.16. The figure shows that there are also multiple channels connecting the switch to different controllers. The advantage provided by the additional connections on a channel lies in achieving greater overall throughput between the switch and the controller. Because of the flow-control characteristics of a TCP connection, it is possible that the connection be forced to quiesce (due to TCP window closing or packet loss) when there is actually bandwidth available on the physical path(s) between the switch and controller. Allowing multiple parallel connections allows the switch to take advantage of that. The first connection in the channel is specified to be a TCP connection. The specification indicates that other, nonreliable connection types,

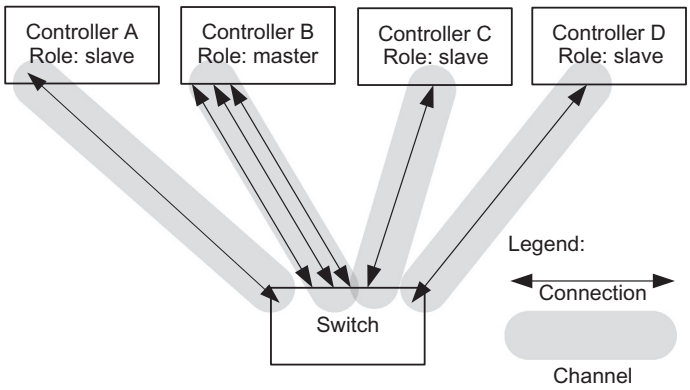

FIG. 5.16

Multiple connections and multiple channels from a switch.

such as UDP, may be used for the secondary connections, but the specification is fairly clear that the problems of potential data loss and sequencing errors that can arise from the use of such alternative transport layers is beyond the scope of the specification.

If bandwidth between the controller and switch becomes a constraint, it is most likely due to a preponderance of data packets. Some loss of data packets can be tolerated. Thus, the primary intended use of the auxiliary connections is to transmit and receive data packets between the switch and controller. This presumes that any control messages would be sent over the reliable, primary connection.

One example of the utility of auxiliary connections is that when a switch has many PACKET_IN messages to send to the controller, this can create a bottleneck due to congestion. The delay due to this bottleneck could prevent important OpenFlow control messages, such as a BARRIER_REPLY message, from reaching the controller. Sending PACKET_IN data messages on the UDP auxiliary connection will obviate this situation. Another possible extension to this idea is that the OpenFlow pipeline could send UDP-based PACKET_IN messages directly from the ASIC to the controller without burdening the control code on the switch CPU.

5.6.6 COOKIES IN PACKET-IN

The straightforward handling of a PACKET_IN message by the controller entails performing a complete packet matching function to determine what existing flow this packet relates to, if any. As the size of commercial deployments of OpenFlow grows, performance considerations have become increasingly important. Accordingly, the evolving specification includes some features that are merely designed to increase performance in high bandwidth situations. Multiple connections per channel, discussed above, is such an example. In the case of PACKET_IN messages, it is somewhat wasteful to require the controller to perform a complete packet match for every PACKET_IN message, considering that this look-up has already just occurred in the switch and, indeed, is the very reason the PACKET_IN message is being sent to the controller (that is, either a specific instruction to forward this packet to the controller was encountered or a table-miss resulted in the packet being handed off to the controller).

This is particularly true if this is likely to happen over and over for the same flow. In order to render this situation more efficient, V.1.3 allows the switch to pass a *cookie* with the PACKET_IN message. This cookie allows the switch to cache the flow entry pointed to by this cookie and circumvent the full packet matching logic. Such a cookie would not provide any efficiency gain the first time it is sent by the switch for this flow, but once the controller cached the cookie and pointer to the related flow entry, considerable performance boost can be achieved. The actual savings accrued would be measured by comparing the computational cost of full packet header matching versus performing a hash on the cookie. As of this writing, we are not aware of any study that has quantified the potential performance gain.

The switch maintains the cookie in a new field in the flow entry. Indeed, the flow entries in V.1.3 have expanded considerably as compared to the basic flow entry we presented in Fig. 5.5. The V.1.3.0 flow entry is depicted in Fig. 5.17. The header, counter and actions fields were present in Fig. 5.5 and were covered earlier in Section 5.3.2. The priority field determines where this particular flow entry is placed in the table. A higher priority places the entry lower in the table such that it will be matched before a lower priority entry. The timeouts field can be used to maintain two timeout clocks for this flow. Both an idle timer and a hard timeout may be set. If the controller wishes for the flow to be removed after a specified amount of time without matching any packets, then the idle timer is used. If the controller wishes for the flow to exist only for a certain amount of time regardless of the amount of traffic, the hard timeout is used and the switch will delete this flow automatically when the timeout expires. We provide an example of such a timeout condition in Section 5.3.6. We covered the application of the cookie field earlier in this section.

5.6.7 PROVIDER BACKBONE BRIDGING TAGGING

Earlier versions of OpenFlow supported both VLAN and MPLS tagging. Another WAN/LAN technology, known as *Provider Backbone Bridging* (PBB) is also based on tags similar to VLAN and MPLS. PBB allows for user LANs to be layer two bridged across provider domains, allowing complete domain separation between the user layer two domain and the provider domain via the use of *MAC-in-MAC* encapsulation. For the reader interested in learning more about PBB, we provide a reference

Header fields	Field value
Priority	Field value
Counters	Field value
Actions/instructions	Field value
Timeouts	Field value
Cookie	Field value

FIG. 5.17

V.1.3 flow entry.

to the related standard in [10]. This is a useful technology already supported by high-end commercial switches, and support for this in OpenFlow falls out easily as an extension to the support already provided for VLAN tags and MPLS shim headers.

5.6.8 EXAMPLE: ENFORCING QOS VIA METER BANDS

Fig. 5.18 depicts the new V.1.3 meter table as well as the group table that first appeared in V.1.1. We see in the figure that the incoming packet is matched against the second flow entry in the first flow table, and that the associated instructions direct the packet to Meter 3. The two dashed lines emanating from the meter table show that, based on the current measured bandwidth of the port selected as the output port (port *N* in this case), the packet may be dropped as exceeding the bandwidth limits, or it may be forwarded to port *N* if the meter does not indicate that bandwidth enforcement is necessary. As we have explained earlier, if the packet passes the meter control without being dropped or marked as dropped-eligible, it is *not* automatically forwarded to the output port, but may undergo any further processing indicated by the instructions in its packet processing pipeline. This could entail being matched against a higher-numbered flow table with its own instructions and related actions.

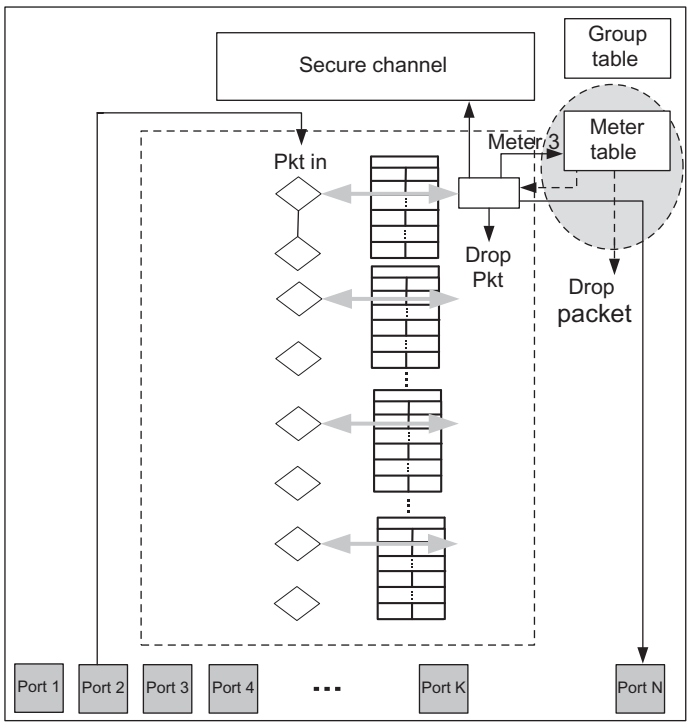

FIG. 5.18

Use of meters for QoS control in V.1.3.

DISCUSSION QUESTION:

Explain why, in Fig. 5.18, if the meter table does not drop the packet, the alternative dashed line indicates that the packet is not sent directly to the output port but rather forwarded back through the packet processing logic.

5.7 OpenFlow 1.4 ADDITIONS

OpenFlow V.1.4.0 [11] was released on October 14, 2013. Table 5.5 lists the major new features added in this release. In the case of the more significant features, the table description references one of the following sections where we provide more explanation of the feature. For other, more minor features, such as new error codes or clarification of the protocol, the table refers the interested reader to [11]. As of this writing, one switch manufacturer, Pica8, has already announced support for OpenFlow 1.4 [12]. The primary drivers behind this decision were the OpenFlow 1.4 support for bundles, eviction and vacancy events, and improved support for multiple controllers. Another notable feature of OpenFlow 1.4 was the new port descriptor which contained fields to support optical ports. We will provide further details on these more important features of OpenFlow 1.4 in the following sections.

5.7.1 BUNDLES

The bundle feature described in [11] provides enhanced transactional capabilities to the OpenFlow controller. In earlier versions, related, but more primitive control came from using a BARRIER_REQUEST

Table 5.5 New Features Added in OpenFlow 1.4

Feature Name	Description
More Extensible Wire Protocol	Standardize on TLV format: See [11]
More Descriptive Reasons for PACKET-IN	Clarification: See [11]
Optical Port Properties	See Section 5.7.4
Flow-removed reasons for meter delete	Clarification: See [11]
Flow monitoring	See Section 5.7.3
Role status events	See Section 5.7.3
Eviction	See Section 5.7.2
Vacancy Events	See Section 5.7.2
Bundles	See Section 5.7.1
Synchronized Tables	See Section 5.7.5
Group and Meter change notifications	See Section 5.7.3
Error code for bad priority	New Error Codes: See [11]
Error code for Set-async-config	New Error Codes: See [11]
PBB UCA header field	See [11]
Error code for duplicate instruction	New Error Codes: See [11]
Error code for multipart timeout	New Error Codes: See [11]
Change default TCP port to 6653	Official IANA port number assigned. See [11]

message whereby the switch must complete execution of all commands received prior to the BARRIER_REQUEST before executing any commands received after it. The OpenFlow 1.4 concept of a bundle is a set of OpenFlow messages that are to be executed as a single transaction. No component message should be acted upon by the switch until the entire set of commands in the bundle has been received and *committed*. The controller uses the new BUNDLE_CONTROL message to create, destroy or commit bundles. OpenFlow 1.4 also defines the two new commands BUNDLE_ADD_MESSAGE and BUNDLE_FAILED to add messages to a bundle and to report the failure of a bundle operation, respectively.

5.7.2 EVICTION AND VACANCY EVENTS

On physical switches, flow tables have very finite capacity and running out of room in a flow table on a switch can cause significant problems for the controller managing that switch. OpenFlow 1.4 contemplates a notion of *vacancy* whereby the switch can inform the controller that a critical level of table capacity has been reached. The motivation behind this is that by being given advance notice that the table is reaching capacity, the controller may be able to take logical steps to prune less important table entries and thus allow continued smooth operation of the switch. OpenFlow 1.4 allows the controller to act on this in a pro-active fashion by denoting certain flows as candidates for automatic eviction by the switch in the event that a vacancy threshold has been reached. Passing a vacancy threshold is communicated to the controller via a TABLE_STATUS message that includes reason codes supporting the vacancy feature. Parameters are provided to set the vacancy threshold itself as a means to quiesce flapping vacancy messages due to too-frequent passing of the threshold. Similarly, the eviction mechanism described in [11] allows configuration of various levels of importance to a flow entry to provide an orderly hierarchy of auto-eviction decisions taken by the switch.

DISCUSSION QUESTION:

Why are the eviction and vacancy events features more relevant to a ASIC-based switch than to a purely software-based switch?

5.7.3 ENHANCED SUPPORT FOR MULTIPLE CONTROLLERS

While OpenFlow 1.2 already included the concept of master and slave controllers, when one controller set itself to master, the previous master controller was not informed of this. In [11], a role status message is defined that allows the switch to inform a controller of a change to its master-slave role.

In addition, OpenFlow 1.4 permits definition of a set of monitors each of which may specify a subset of the flow tables on that switch. When a flow entry is added, deleted or changed within that subset, a notification is sent to the particular controller associated with that monitor. This capability is generically described as *flow monitoring* in [11].

Similarly, V.1.4 supports *group and meter change notifications* whereby the other controllers managing a switch are notified of group and meter changes effected by one controller.

5.7.4 OPTICAL PORT SUPPORT

The port descriptor is augmented in [11] to permit configuration and reporting on a number of optical-specific properties. These include fields for the transmit power of a laser, its transmit and receive frequencies or wavelengths, whether the transmitter and receiver are tunable, and whether the transmit power is configurable. These fields permit the configuration and monitoring of both Ethernet optical ports or the ports on optical circuit switches. Note that while these OpenFlow 1.4 extensions are a start for providing full OpenFlow support for optical ports, the ONF's *Optical Transport Working Group* (OTWG) has continued work in this area. The first specification emanating from this working group is discussed below in Section 5.10.

5.7.5 FLOW TABLE SYNCHRONIZATION

When two flow tables are synchronized, this signifies that when one table is changed the other table will be automatically updated. This is more complicated than it may first appear since the modification to one table may differ from that made to the second table. For example, if the first table matches on the source Ethernet address, the synchronization with the second table may transpose this to the destination Ethernet address. This is precisely the case for the example provided in [11] for flow table synchronization, which is a MAC learning bridge. In this example, when a new Ethernet address is learned on a port, an entry needs to be created in the first table (the learning table) with an idle timer so that inactive addresses can be later pruned. A *forwarding* table could be synchronized with the learning table and the newly learned (timed-out) addresses would be added (deleted) in that second table. The exact details of how the synchronization would occur are not specified in [11].

5.8 OpenFlow 1.5 ADDITIONS

OpenFlow V.1.5.0 [13] was released on December 19, 2014. Table 5.6 lists the major new features added in this release. In the case of the more significant features, the table description references one of the following sections where we provide more explanation of the feature. For other, more minor features related to consistency, flexibility or clarification of the protocol, the table refers the interested reader to [13]. A cursory inspection of the table of contents of the V.1.5 specification leads to the conclusion that there are far more additions to V.1.4 than is in fact the case. This is due to the fact that the document has been reorganized and section labels have been introduced to improved readability. There are, however several important new features introduced which we cover in the sections that follow.

5.8.1 ENHANCED L4–L7 SUPPORT

Even prior to OpenFlow 1.5, directing a packet to a logical port could be used to insert a network service into the flow. With the addition of *port recirculation* in V.1.5, it is now possible for the packet to be directed back to the OpenFlow packet processor after completing the logical port processing. This can enable *service chaining*, which is the sequencing of L4–L7 services such as firewalls or load balancers. This would occur by effectively chaining multiple logical ports such that a packet would be processed by each in sequence in the same switch.

Table 5.6 New Features Added in OpenFlow 1.5

Feature Name	Description
Egress Tables	See Section 5.8.3
Packet Type Aware Pipeline	See Section 5.8.4
Extensible Flow Entry Statistics	See Section 5.8.8
Flow Entry Statistics Trigger	See Section 5.8.8
Copy-Field action to copy between two OXM fields	See Section 5.8.2
Packet Register pipeline fields	See Section 5.8.2
TCP flags matching	See Section 5.8.1
Group Command for selective bucket operation	Flexibility: See [13]
Allow set-field action to set metadata field	See Section 5.8.1
Allow wildcard to be used in set-field action	See Section 5.8.1
Scheduled Bundles	See Section 5.8.5
Controller Connection Status	See Section 5.8.6
Meter Action	Flexibility: See [13]
Enable setting of all pipeline fields in PACKET_OUT	See Section 5.8.7
Port properties of all pipeline fields	See Section 5.8.7
Port Property for recirculation	See Section 5.8.1
Clarify and improve BARRIER	Consistency and clarification: See [13]
Always generate port status on port config change	See Section 5.8.6
Make all Experimenter OXM-IDs 64 bits	Consistency and clarification: See [13]
Unified requests for group, port and queue multiparts	Consistency and clarification: See [13]
Rename some types for consistency	Consistency and clarification: See [13]
Specification Reorganization	Consistency and clarification: See [13]

V.1.5 also offers the capability to maintain flow state via the storing and accessing flow metadata that persists for the lifetime of the flow, not just the current packet. Another new feature of V.1.5 that enhances L4 support is the ability to detect the start and end of TCP connections. This capability comes from the newly added capability of matching on the flag bits in the TCP header.

5.8.2 PIPELINE PROCESSING ENHANCEMENTS

The new *copy-field* action permits the copying of one header or pipeline field into another header or pipeline field. Prior to V.1.5, such fields could only be set to statically programmed values via the *set-field* action. *Packet registers*, new in V.1.5, provide scratchpad space that may be used to pass nonstandard information through the pipeline steps. These V.1.5 features, along with other, more minor ones appearing in Table 5.6, provide new robustness to the transformations that may occur to a packet as it passes through an OpenFlow pipeline.

5.8.3 EGRESS TABLES

V.1.5 introduces the notion of *egress* tables. In earlier versions, a flow table was consulted as part of the matching process on an incoming packet. This traditional role of the flow table is now referred to as an *ingress* table. Ingress table processing is still a mandatory part of the OpenFlow pipeline, whereas egress tables may or may not exist. In a switch where such two-stage pipeline processing is configured,

matching occurs against the egress tables once the output port(s) have been identified in the ingress processing. An example would be where the ingress processing consults a group table which results in the packet being forwarded out three output ports. By including egress tables, three distinct packet transformations may thus be performed in the separate contexts of each of the three output ports. This allows much greater flexibility in encapsulation than was formerly possible.

5.8.4 FITNESS FOR CARRIER USE

While a number of the V.1.5 features mentioned above improve OpenFlow's suitability in a carrier environment, most salient is the support of other data plane technologies beyond Ethernet. V.1.5 includes a new OXM pipeline field that identifies the packet type. The additional packet types supported in V.1.5 are IPV4 (no header in front), IPV6 (no header in front), no packet (e.g., circuit switch), and EXPERIMENTER.

5.8.5 BUNDLE ENHANCEMENTS

We introduced the concept of bundles in Section 5.7.1. V.1.5 now allows for an execution time to be specified with the bundle so that the bundle may be delivered to the switch in advance of the time that the switch should begin acting on the messages contained therein. This new version of OpenFlow also allows a controller to ask the switch to report its bundle capabilities, so that the controller will know what set of bundle features it may invoke on that switch.

5.8.6 ENHANCED SUPPORT FOR MULTIPLE CONTROLLERS

Some new features in V.1.5 are specifically targeted at facilitating more than one controller managing the same switch. One of these is *controller connection status*. This allows one controller to ask the switch the status of its connections to *any* controller, thereby affording one controller knowledge of other controllers managing the same switch. Another weakness in earlier versions was that when a port status was changed via OpenFlow, no port status change message was sent to the controller. In a single controller environment this is perfectly reasonable since the one controller would be aware that it itself effected the port status change. In a multicontroller environment, however, the remaining controllers would not be aware of the port status change which is potentially problematic. In V.1.5 a port status change message will always be sent to all controllers connected to the switch where the port was altered.

5.8.7 ENHANCED SUPPORT FOR TUNNELS

Starting with V.1.5, a logical port's properties allow specification of which OXM pipeline fields should be provided for a packet being sent to that port as well as which OXM fields should be included when a packet is transmitted from that logical port. This can provide enhanced tunnel support since the pipeline field *tunnel_id* can be used to hold the metadata associated for the encapsulation used by a tunnel. In support of this, V.1.5 also includes the ability to set *all* pipeline fields in the PACKET_OUT message, whereas earlier versions only supported the setting of the *in_port* field. In particular, should a logical port's V.1.5 properties denote that it *consumes* (i.e., requires) the *tunnel_id* field, the packet forwarding logic sending to that logical port would be able set the *tunnel_id* field thus passing information relevant to the aforementioned tunnel encapsulation.

5.8.8 ENHANCEMENTS TO FLOW ENTRY STATISTICS

Flow entry statistics are represented in a more extensible format starting in V.1.5. Statistic fields supported in earlier versions such as flow duration, and counters for flows, packets and bytes are now expressed in standard TLV format. Flow idle time is now captured as a standard statistic. In addition, support for arbitrary experimenter-based statistics is included. Another important performance enhancement pertaining to flow entry statistics is the addition of a trigger capability. This eliminates the overhead of the controller having to poll the switch for statistics and enables the switch to proactively send statistics to the controller on the basis of a trigger. This trigger may be based on either a time period or on the basis of a statistic reaching a certain threshold value.

5.8.9 SUMMARY

While OpenFlow V.1.3 provided the flexibility to handle most challenges posed by the data center, it did not meet the much greater flexibility required by network operators in the WAN [14]. The changes encompassed by V.1.4 and V.1.5 discussed above have provided many features needed for such WAN deployments. Commercial adoption of V.1.4 and V.1.5 has so far, however, been weak. Part of this stems from persistent problems achieving interoperability between mixes of controllers and switches from different manufacturers that all claim compliance with the standards. We discuss some of the ONF's approaches to address this issue in the next section.

5.9 IMPROVING OpenFlow INTEROPERABILITY

The move away from the single flow table of OpenFlow 1.0 to multiple flow tables, specified in OpenFlow 1.1, and implemented by most vendors in OpenFlow 1.3 was instigated in order to open up the rich potential of SDN to an ever increasing spectrum of applications. The single, very flexible flow table did not map well to the hardware pipelines available in advanced switching silicon [15]. While the chaining of multiple flow tables into a pipeline does indeed more closely approximate real hardware implementations, differences between these implementations required that any application wishing to use these detailed features needed to be developed with intimate knowledge of those hardware details. One reason for the difference in these implementations is that support for many OpenFlow 1.3 features is considered optional, further confusing the task for the application developer. The situation was such that applications intended for use on hardware switches needed to be developed specifically for those switches. While it was possible to implement uniform OpenFlow support on software-based switches, limiting an application to such switches usually reduces its usefulness.

In order resolve this interoperability problem, the ONF has espoused two new abstractions that are intended to facilitate interoperation between OpenFlow 1.3 switches. These two abstractions are *Table Type Patterns* (TTPs) and *Flow Objectives.* We discuss these in the following sections.

5.9.1 TABLE TYPE PATTERNS

In 2014, the ONF published the first TTP specification [16]. A TTP is a formal and detailed description of a complete set of logical switch behavior [15]. Many *OpenFlow Logical Switches* (OFLSs) may coexist inside a given physical switch. From an application developer's perspective, an OFLS possesses

the switching capabilities required by a particular application. From the switch vendor's perspective, the behavior of an OFLS is described succinctly in the OpenFlow specification terminology of the TTP. The fact that a particular TTP is succinct and limited to one application or class of applications makes it feasible for the switch vendor to confirm support or nonsupport for that particular OFLS. As the OFLS is described by a TTP, the TTP becomes the language that the controller and switch can use to confirm or deny the existence of a needed set of capabilities.

The formal language to describe a TTP is described in [15] and is based on *JavaScript Object Notation* (JSON). A very useful sample TTP describing an OFLS implementing a simple L2 and L3 forwarding switch is provided in [15].

5.9.2 FLOW OBJECTIVES

While TTPs provide a uniform way of describing a particular pipeline needed in a multi-table OpenFlow-enabled switch, the language of TTPs still requires detailed knowledge of the OpenFlow details that implement that pipeline. Flow objectives are intended to abstract multitable OpenFlow-level actions into generic application *objectives* [17]. These services-oriented objectives describe relatively high-level switching functions needed by an application but implemented via different pipelines on different physical switches. In Fig. 5.19 we show where flow objectives are defined within the *Open Network Operating System* (ONOS) embedded within the *Atrium* open source solution provided by the ONF [18]. Examples of these services-oriented objectives would include a filtering objective, a forwarding objective, and a next-hop objective.

The goal of exposing the flow objective abstraction is to permit application developers to build applications that benefit from scalable, multitable OpenFlow-based switching platforms without being aware of the details of the OpenFlow pipelines that would implement the needed services. As shown in Fig. 5.19, the pipeline-specific drivers architecturally below the flow objectives mask the differences between the different pipelines allowing the application developer to specify services at a more abstract level than using OpenFlow terminology.

DISCUSSION QUESTION:

Both TTPs and flow objectives are intended to enhance interoperability between OpenFlow implementations and as a result increase overall adoption of OpenFlow. In what ways do these two concepts differ in their approach to accomplishing these goals? In what way are they similar?

5.10 OPTICAL TRANSPORT PROTOCOL EXTENSIONS

The OpenFlow *Optical Transport Protocol Extensions* [19] was released on March 15, 2015. This specification builds upon the *port attribute extensibility* defined in OpenFlow 1.4. While that V.1.4 extension does facilitate the configuration and reporting of optical-specific port parameters, there are a number of areas that remain to be addressed to provide for robust OpenFlow support of practical optical networking. We discuss these below.

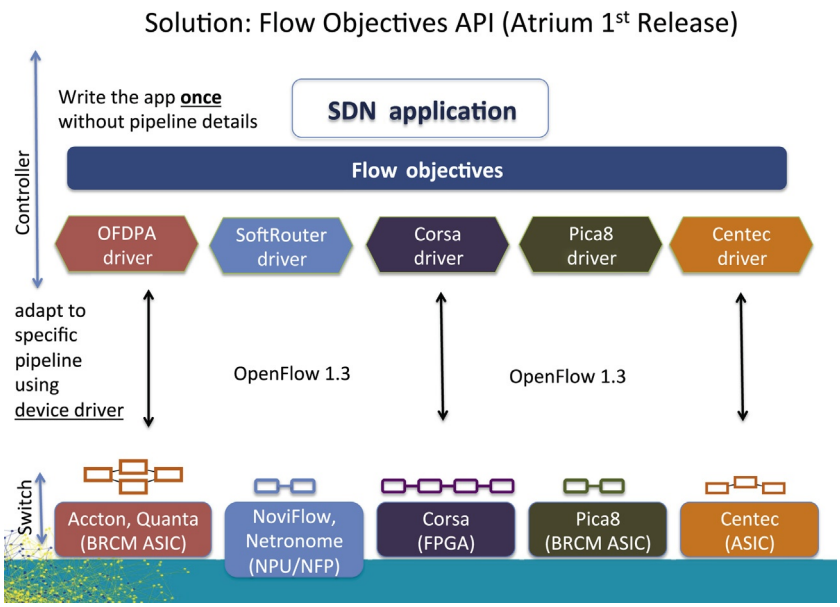

FIG. 5.19

Flow objectives abstraction.

Reproduced with permission from Saurav D. ATRIUM Open SDN Distribution, SDN Solutions Showcase, ONS 2015, Dusseldorf. Retrieved from: https://www.opennetworking.org/images/stories/news-and-events/sdn-solutions-showase/Atrium-ONS-Live.pdf.

5.10.1 MATCH/ACTION SUPPORT

In packet switching, much of the control functionality can be achieved by the control plane injecting control packets into the data plane. For example, knowledge about the adjacencies that together form the topology of a network can be collected by the exchange of such control packets between neighboring control planes. This method does not work with optical switching where the switches cannot simply inject packets in-line.

Similarly, optical switches are not packet switches, they are circuit switches. Implicit in this distinction, an OpenFlow-enabled circuit switch will not typically inspect packet headers to use as match criteria for flow tables, and thus not use that information to take actions such as forward or drop. Instead, the circuit switch will base these match and action decisions for a signal on the basis of an *Optical Channel* (OCh) characteristic such as wavelength for L0 switching or on the basis of an *Optical Data Unit* (ODU) characteristic like ODU tributary slot for L1 switching. The ODU tributary slot would identify the circuit by its position in time whereas individual physical optical channels are distinguished by transmission on different wavelengths. In either case, identification of the circuit permits a flow table-based set of rules to prescribe what action the switch should take for this circuit. Note that in this *Optical Transport Network* (OTN) context, we use the terms signal, channel, and circuit interchangeably.

5.10.2 OPTICAL PORT ATTRIBUTE SUPPORT

In [19] the OTWG defines *OTN port description extensions*. These extensions are used to identify and define the port types for L0 and L1 circuit switching control. More specifically, the extensions are used to specify and report on the characteristics of *Optical Transmission Section* (OTS), *Optical Physical Section* (OPS) and *Optical Channel Transport Unit* (OTU) ports. Examples of a characteristic that would be specified would be the specific layer class of an OTN signal. Examples of these layer classes are:

- OCH (OCh layer signal type)
- ODU (ODU layer signal type)
- ODUCLT (ODU Client layer signal type).

Within each of these layer classes, there are many possibilities for the specific signal type. This specific signal type is also specified in the OTN extensions. There are numerous other OTN-specific fields defined in these extensions and it is beyond the scope of this book to delve into the details of optical networking. We refer the reader to [19] for further details.

In [19] two methods are provided by which these OTN port description extensions may be passed between the controller and the switch, one based on OpenFlow 1.3 and one based on OpenFlow 1.4. One of the changes in OpenFlow 1.4 was extension of the basic port descriptor to provide support for a broader range of port types. While this expanded port descriptor provided a richer set of parameters for describing a port, it still lacks the full richness required to fully describe an OTN port. Thus, in OpenFlow 1.4 and beyond, the OTN extensions are captured by a combination of the extended port descriptor plus a set of OTN extensions defined in [19]. For OpenFlow 1.3, Ref. [19] specifies an experimenter multipart message whose body will contain the OpenFlow 1.4 port descriptor followed by those OTN extensions. (Note that these OTN extensions are sometimes referred to as the OTWG OpenFlow 1.5 OTN extensions.)

5.10.3 ADJACENCY DISCOVERY

Since optical switches cannot inject control packets in-line into the data plane to pass control and topology information through the network as packet switches currently do, other mechanisms are necessary to provide important basic features such as adjacency discovery. One proposed mechanism is based on using the control information known as the *Trail Trace Identifier* (TTI). Current ITU-sponsored standards provide that TTI be passed in-line within the data plane. In the optical world, this information is available to the control plane. The extensions contemplated in [19] allow for this TTI information to be made available via OpenFlow to the controller. The controller, in turn, can use this TTI information reported from different switches to synthesize adjacencies and thus network topology in much the way it does for packet switching. This represents a feasible mechanism for adjacency discovery for OpenFlow enabled optical circuit switches.

5.10.4 FUTURE WORK NEEDED

A number of areas that were identified as important for OpenFlow support of OTN require further specification than that provided in [19]. These include:

- Support for carrier reliability mechanisms for the data plane, in particular support for OAM monitoring of OTN network links as well as support for rapid protection switching functions that guarantee recovery from link failure.

- Support for network elements handling multiple technology layers according to transport layering models.
- Support for a hierarchy of controllers. In such a hierarchy, OpenFlow messages are exchanged between parent and child controllers in a *Control Virtual Network Interface* (CVNI) context. This is important as the scale required to support carrier-grade optical networks will require large numbers of OpenFlow controllers arranged in such a hierarchy.

As well as the future work identified in [19], the Calient corporation has identified areas that are urgently needed to foster the use of OpenFlow in optical circuit switches. These additional proposed extensions are described in [20].

DISCUSSION QUESTION:

Could OpenFlow-enabled switches support optical ports without the extensions described in Section 5.10? What major new class of OpenFlow-enabled switches is contemplated via these extensions?

5.11 OpenFlow LIMITATIONS

As OpenFlow remains in a state of rapid evolution, it is difficult to pin down precise limitations as these may be addressed in subsequent releases. One limitation is that the currently defined match fields are limited to the packet header. Thus, *Deep Packet Inspection*, (DPI) where fields in the packet's payload may be used to distinguish flows, is not supported in standard OpenFlow. Nonetheless, the EXPERIMENTER modes that are permitted within OpenFlow do open the way for such application-layer flow definition in the future. Secondly, some OpenFlow abstractions may be too complex to implement directly in today's silicon. This is unlikely to remain an insurmountable obstacle for long, however, as the tremendous momentum behind SDN is likely to give birth to switching chips that are designed explicitly to implement even OpenFlow's most complicated features. Another limitation is the possible processing delay if there is no matching entry in the OpenFlow switch's flow table, in which case processing the packet mandates that it be sent to the controller. While this is just a default behavior and the switch can be easily programmed to handle the packet explicitly, this potential delay is inherent to the OpenFlow paradigm. There are also a number of possible security vulnerabilities introduced by OpenFlow, such as (1) *Distributed Denial of Service* (DDoS) risks that can make the controller unavailable, and (2) failure to implement switch authentication in the controller. A brief discussion on these and other potential OpenFlow security vulnerabilities can be found in [21].

If we cast a wider net and consider limitations of all of Open SDN, there are a number of other areas to consider. We discuss these in Section 6.1.

5.12 CONCLUSION

In this chapter we have attempted to provide the reader with a high-level understanding of the general OpenFlow framework. This covered the protocol that an OpenFlow controller uses to configure and control an OpenFlow switch. We have also presented the switch-based OpenFlow abstractions that must be implemented on a physical switch as an interface to the actual hardware tables and forwarding engine of the switch in order for it to behave as an OpenFlow switch. For the reader interested in a

Table 5.7 OpenFlow Protocol Constant Classes

Prefix	Description
OFPT	OpenFlow Message Type
OFPPC	Port Configuration Flags
OFPPS	Port State
OFPP	Port Numbers
OFPPF	Port Features
OFPQT	Queue Properties
OFPMT	Match Type
OFPXMC	OXM Class Identifiers
OFPXMT	OXM Flow Match Field Types for Basic Class
OFPIT	Instruction Types
OFPAT	Action Types
OFPC	Datapath Capabilities and Configuration
OFPTT	Flow Table Numbering
OFPFC	Flow Modification Command Type
OFPFF	Flow Modification Flags
OFPGC	Group Modification Command Type
OFPGT	Group Type Identifier
OFPM	Meter Numbering
OFPMC	Meter Commands
OFPMF	Meter Configuration Flags
OFPMBT	Meter Band Type
OFPMP	Multipart Message Types
OFPTFPT	Flow Table Feature Property Type
OFPGFC	Group Capabilities Flags

deeper understanding of these issues, whether out of sheer academic interest or a need to implement one of the versions of OpenFlow, there is no substitute for reading the OpenFlow specifications cited in this chapter, especially [13], as it is generally an updated superset of its predecessors. These specifications are, however, very detailed and are not especially easy for the OpenFlow novice to follow. We believe that this chapter will serve as an excellent starting point and guide for that reader who needs to delve into the lengthy specifications that comprise the OpenFlow releases covered here. In particular, these specifications contain a plethora of structure and constant names. To aid in reading the specifications, we draw your attention to Table 5.7 in this chapter as a quick reference and aid to understanding to what part of the protocol a specific reference pertains.

REFERENCES

[1] OpenFlow Switch Specification, Version 1.0.0 (Wire Protocol 0x01). Open Networking Foundation, December 31, 2009. Retrieved from: https://www.opennetworking.org/sdn-resources/onf-specifications.

[2] IEEE Standard for Local and Metropolitan Area Networks—Media Access Control (MAC) Bridges and Virtual Bridged Local Area Networks—Amendment 21: Edge Virtual Bridging. IEEE 802.1Qbg, July 2012. New York: IEEE; 2012.

[3] OpenFlow Switch Specification, Version 1.1.0 (Wire Protocol 0x02). Open Networking Foundation, December 31, 2009. Retrieved from: https://www.opennetworking.org/sdn-resources/onf-specifications.
[4] Davie B, Doolan P, Rekhter Y. Switching in IP networks. San Francisco: Morgan Kaufmann; 1998.
[5] IEEE Standard for Local and Metropolitan Area Networks–Link Aggregation. IEEE 802.1AX, November 3, 2008. USA: IEEE Computer Society; 2008.
[6] OpenFlow Switch Specification, Version 1.2.0 (Wire Protocol 0x03). Open Networking Foundation, December 5, 2011. Retrieved from: https://www.opennetworking.org/sdn-resources/onf-specifications.
[7] IEEE Standard for Local and Metropolitan Area Networks—Media Access Control (MAC) Bridges and Virtual Bridged Local Area Networks. IEEE 802.1Q, August 2011. New York: IEEE; 2011.
[8] OpenFlow Switch Specification, Version 1.3.0 (Wire Protocol 0x04). Open Networking Foundation, June 25, 2012. Retrieved from: https://www.opennetworking.org/sdn-resources/onf-specifications.
[9] Shenker S, Partridge C, Guerin R. Specification of guaranteed quality of service. RFC 2212. Internet Engineering Task Force; 1997.
[10] Draft Standard for Local and Metropolitan Area Networks—Virtual Bridged Local Area Networks—Amendment 6: Provider Backbone Bridges. IEEE P802.1ah/D4.2, March 2008. New York: IEEE; 2008.
[11] OpenFlow Switch Specification, Version 1.4.0 (Wire Protocol 0x05). Open Networking Foundation, October 14, 2013. Retrieved from: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.4.0.noipr.pdf.
[12] Oliver B. Pica8: First to adopt openflow 1.4; why isn't anyone else? Tom's ITPRO, May 2, 2014. Retrieved from: http://www.tomsitpro.com/articles/pica8-openflow-1.4-sdn-switches,1-1927.html.
[13] OpenFlow Switch Specification, Version 1.5.0 (Wire Protocol 0x06). Open Networking Foundation, December 19, 2014. Retrieved from: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.0.noipr.pdf.
[14] Hunt G. The Quest for Dominance: OpenFlow or NETCONF for Networks Outside the Data Center? Network Matter, February 6, 2015. Retrieved from: http://networkmatter.com/2015/02/06/the-quest-for-dominance-openflow-or-netconf-for-networks-outside-the-data-center/.
[15] Dixon C. OpenFlow & Table Type Patterns in OpenDaylight's Next Release. SDX Central, August 11, 2014. Retrieved from: https://www.sdxcentral.com/articles/contributed/openflow-table-type-patterns-opendaylight-next-release-colin-dixon/2014/08/.
[16] OpenFlow Table Type Patterns, Version 1.0 (ONF TS-017). Open Networking Foundation, August 15, 2014. Retrieved from: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/OpenFlowTableTypePatternsv1.0.pdf.
[17] Pitt D. Paving the way For SDN interoperability. InformationWeek Network Computing, October 9, 2015. Retrieved from: http://www.networkcomputing.com/networking/paving-the-way-for-sdn-interoperability-/a/d-id/1322562.
[18] Saurav D. ATRIUM Open SDN Distribution. SDN Solutions Showcase, ONS 2015, Dusseldorf. Retrieved from: https://www.opennetworking.org/images/stories/news-and-events/sdn-solutions-showcase/Atrium-ONS-Live.pdf.
[19] Optical Transport Protocol Extensions, Version 1.0 (ONF TS-022). Open Networking Foundation, March 15, 2015. Retrieved from: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/Optical_Transport_Protocol_Extensions_V1.0.pdf.
[20] Optical Circuit Switch OpenFlow Protocol Extensions. Calient, June 2, 2015. Retrieved from: https://wiki.opendaylight.org/images/1/1d/OCS_OF_Protocol_Extensions_Rev._0.4.pdf.
[21] Benton K, Camp L, Small C. OpenFlow Vulnerability Assessment. HotSDN, SIGCOMM 2013, August 2013, Hong Kong. Retrieved from: http://conferences.sigcomm.org/sigcomm/2013/papers/hotsdn/p151.pdf.