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
Copyright © 2017 Elsevier Inc. All rights reserved.

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.

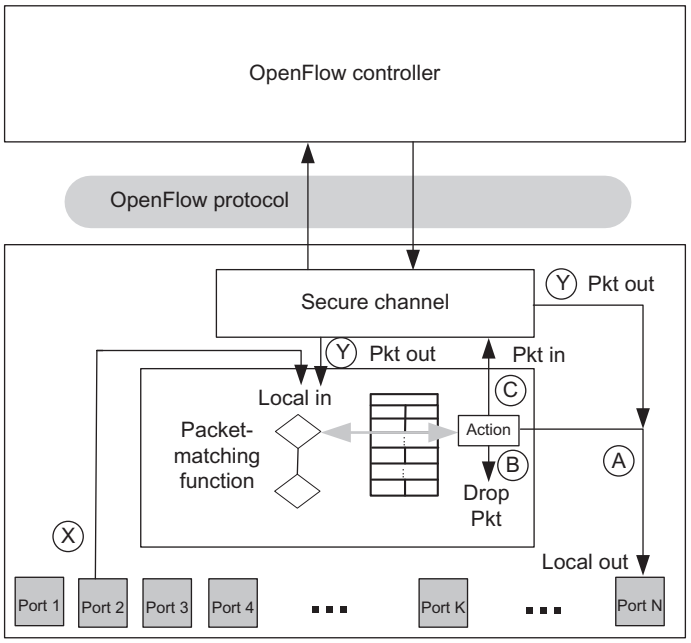


FIG. 5.1

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

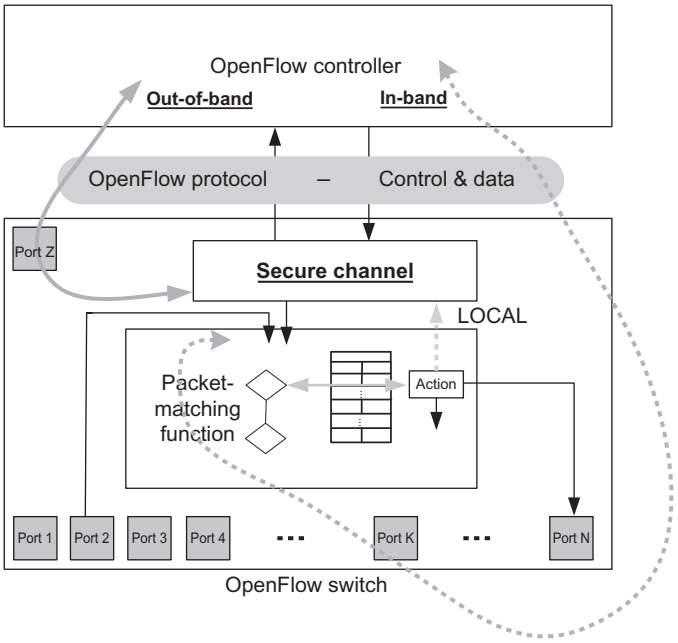


FIG. 5.2

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

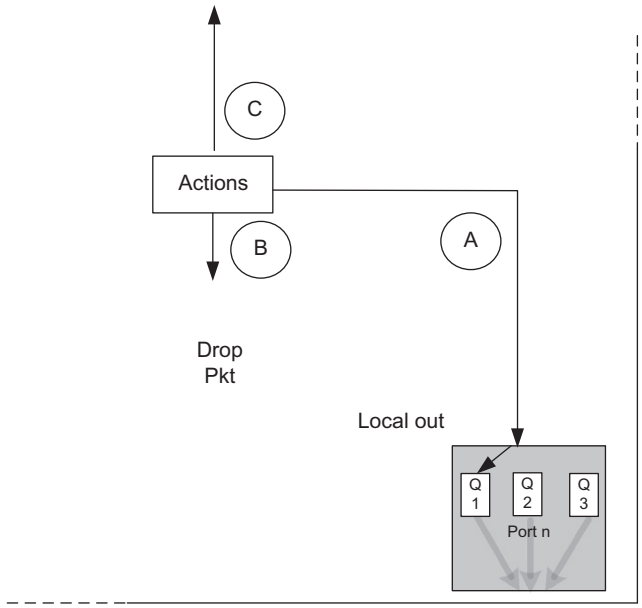


FIG. 5.3

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

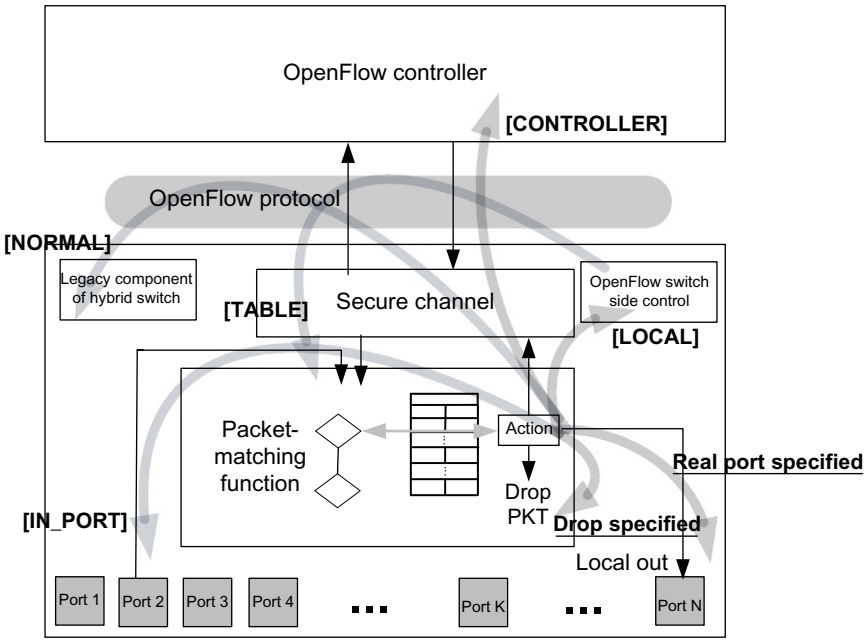


FIG. 5.6

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

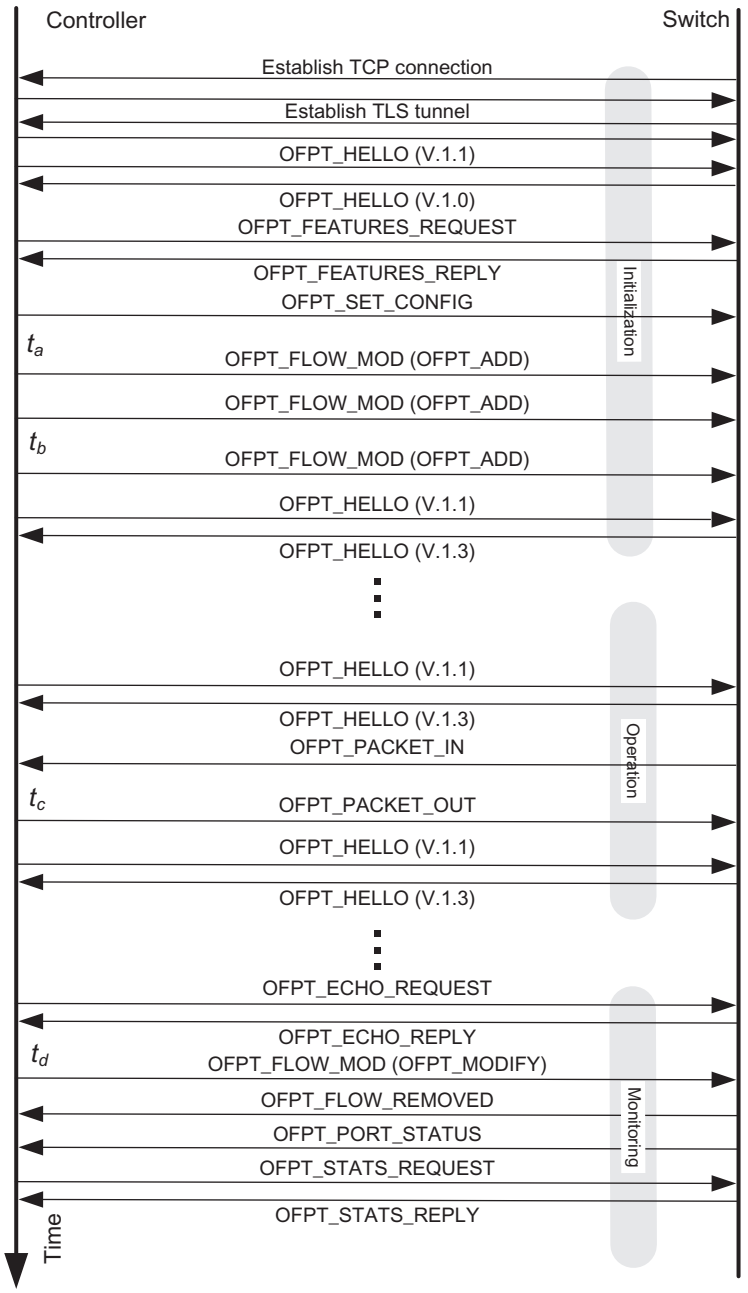


FIG. 5.7

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.

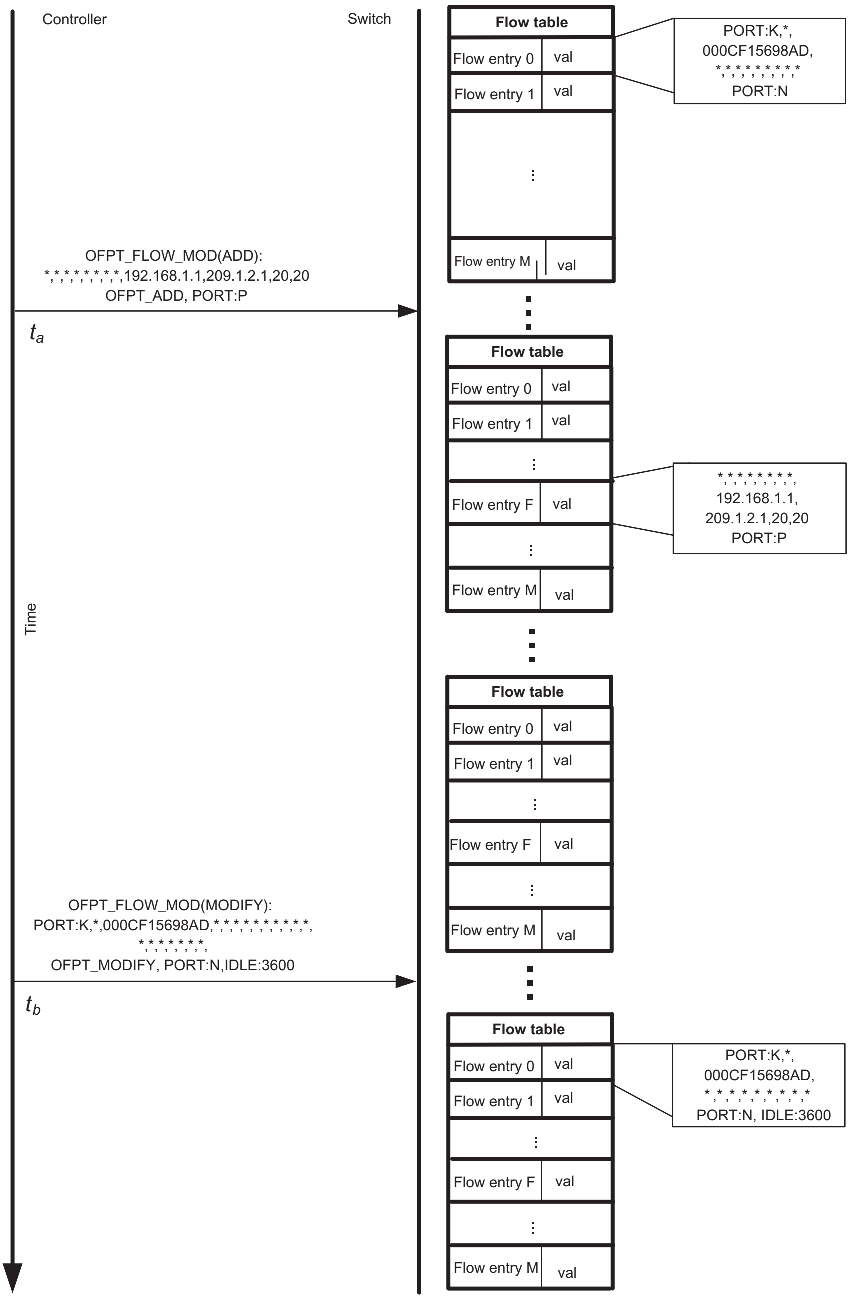


FIG. 5.8

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

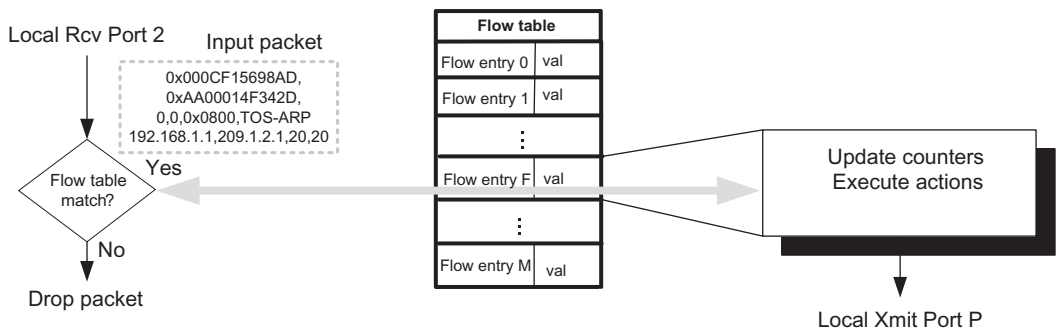


FIG. 5.9

Packet matching function—basic packet forwarding V.1.0.

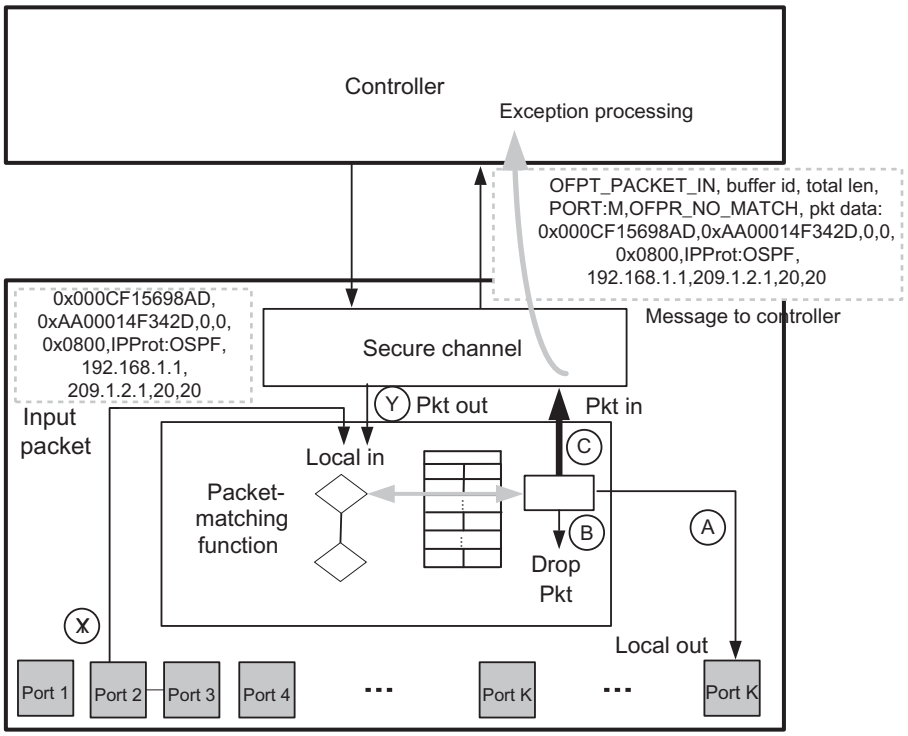


FIG. 5.10

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.