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

5.4 OpenFlow 1.1 Additions

OpenFlow 1.1 [2] 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. We highlight these new features in Figure 5.12. 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 V.1.1 here, though, because the subsequent versions of OpenFlow are built on 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 as 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 was available in V.1.0. This improvement derives from the fact that flow entries can be *chained* by an instruction 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 Figures 5.10 and 5.11 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 Figure 5.12. 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*.[2]

[2]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 12.9.3 and 13.2.7, we discuss attempts to specifically design hardware with this capability in mind.

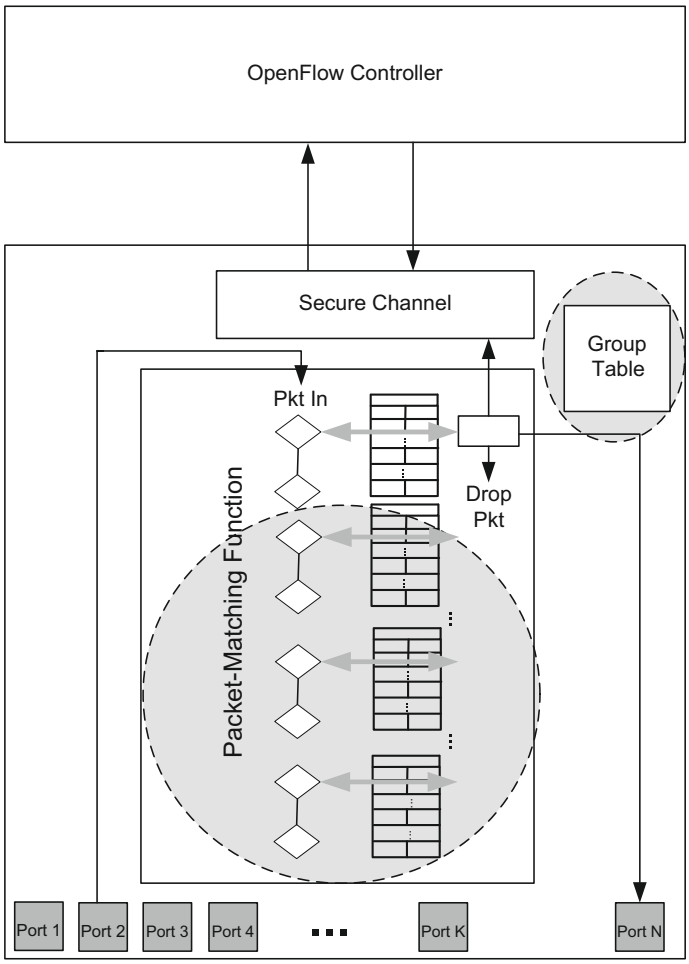


FIGURE 5.12

OpenFlow V.1.1 switch with expanded packet processing.

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 the actions that are taken and in what order. They can do this in two ways. First, they can 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 inward
2. Pop

3. Push
4. Copy TTL outward
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 is 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 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 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, 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 next.

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 we first included in Figure 5.12 and is shown in detail in Figure 5.13. 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 Figure 5.13. 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 wants to change that next hop due to a change in IP routing tables in the controller, all the flows can be rerouted simply by 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

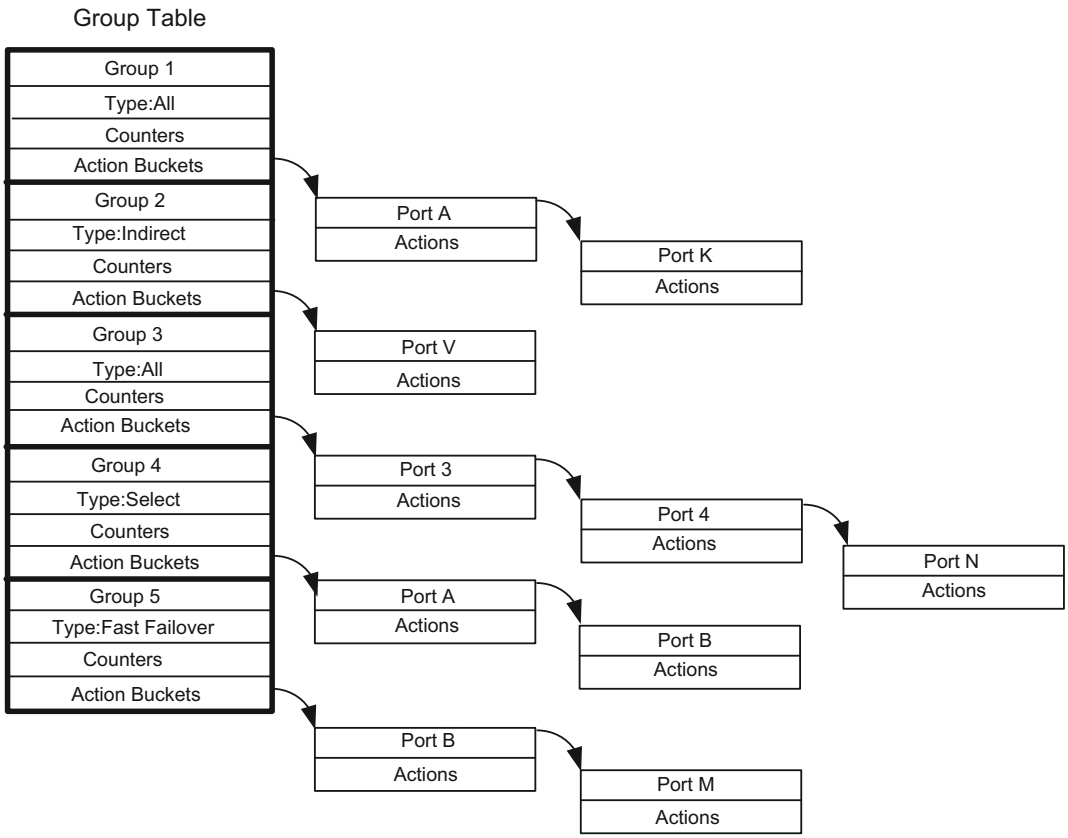


FIGURE 5.13

OpenFlow V.1.1 group table.

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 [5]. 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. Whereas the PUSH is used to add a new tag, the POP is used to remove the current outermost tag. In addition to PUSHing and POPping, V.1.1 also permits modification of the current outermost tag, whether it be a VLAN tag or an MPLS shim header.

V.1.1 correctly claims full MPLS and VLAN tag support, but 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 we introduce 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*. Though the TABLE and other virtual ports did exist in earlier versions of OpenFlow, the concept of virtual ports was augmented in V.1.1. A V.1.1 switch classifies 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 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, refer to [7].

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.
- **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 whereby the packet is forwarded to the switch's local OpenFlow control software. Because a LOCAL port may be used both as an output port and an ingress port, LOCAL can be used to implement an in-band controller connection, obviating the need for a separate control network for the controller-switch connection. Refer 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 non-OpenFlow pipeline of the switch. NORMAL 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. 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 cache was dropped in V.1.1 and replaced with two new mechanisms, *fail secure mode* and *fail standalone mode*. 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 on 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 standalone 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 backward compatibility in the OpenFlow specifications.)

5.4.6 Example: Forwarding with Multiple Flow Tables

Figure 5.14 expands on our earlier example of V.1.0 packet forwarding that was presented in Figure 5.10. Assuming that the incoming packet is the same as in Figure 5.10, we see in Figure 5.14 that there is a match in the second flow entry in flow table 0. Unlike in 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 $A3$ 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 the packet is forwarded out port P in Figure 5.14.

5.4.7 Example: Multicast Using V.1.1 Groups

We direct the reader to Figure 5.13, 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

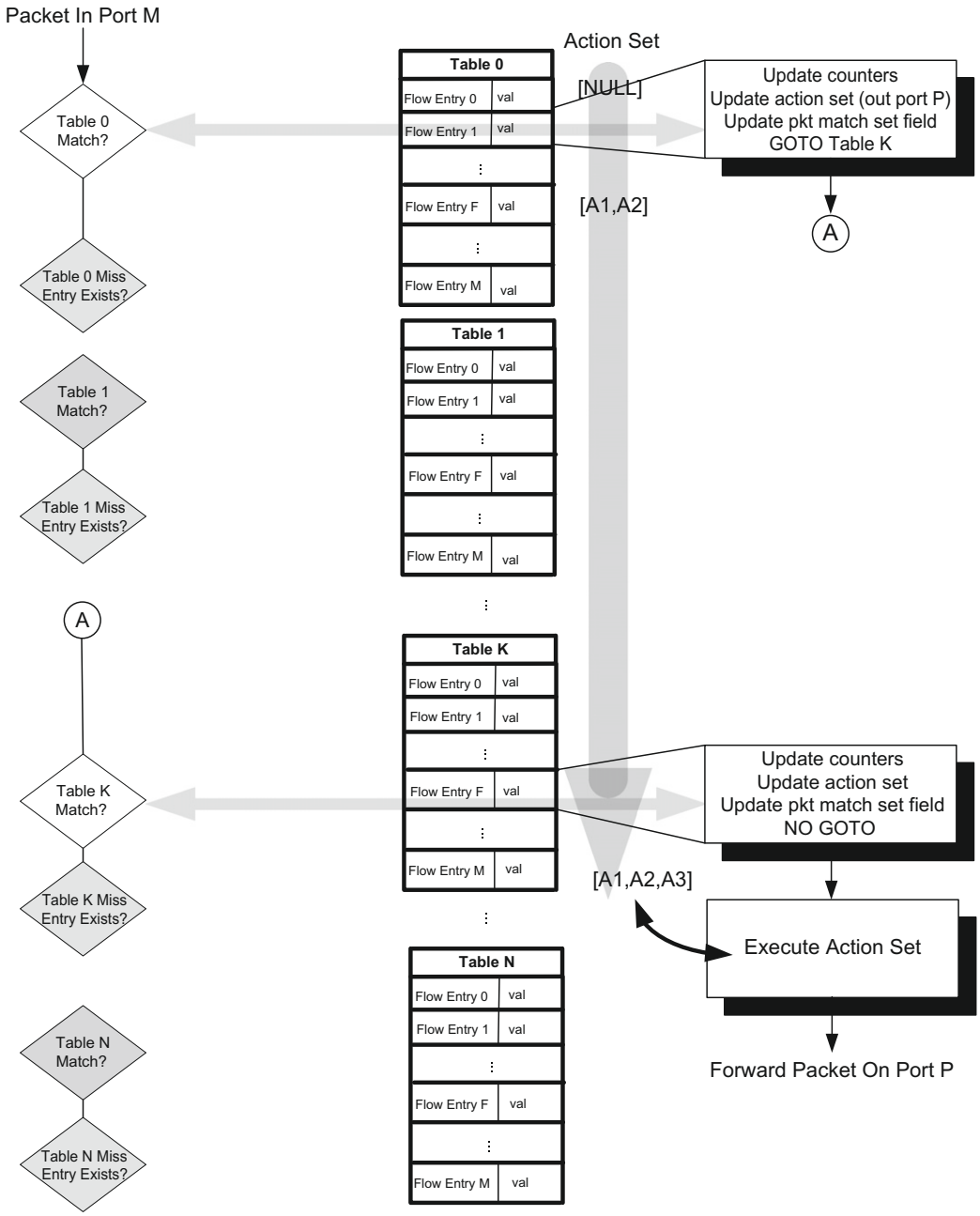


FIGURE 5.14

Packet matching function - basic packet forwarding, V.1.1.

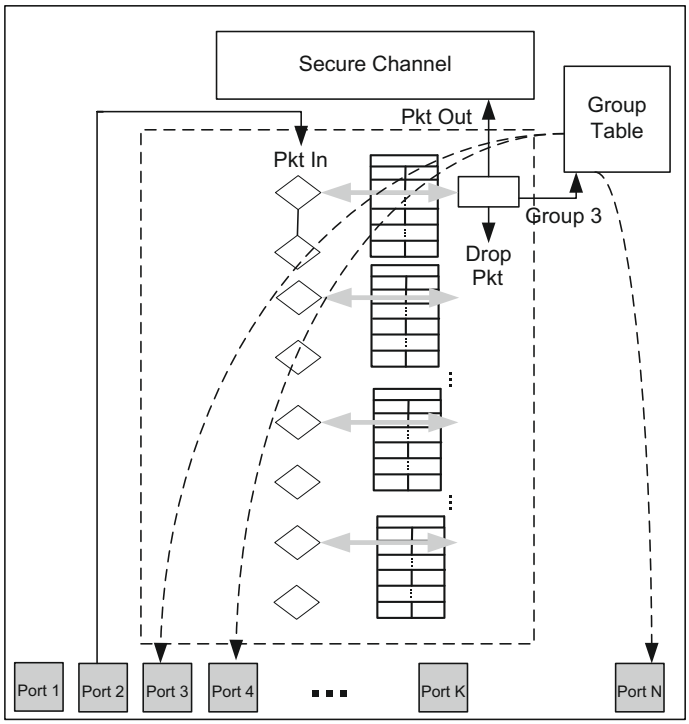


FIGURE 5.15

Multicast using group table in V.1.1.

by this group entry should have a copy sent out each of the ports in the three action buckets. Figure 5.15 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 Figure 5.15 is as shown in Figure 5.13, the packet is multicast out ports 3, 4, and *N*, as indicated by the dashed arrows in the figure.

5.5 OpenFlow 1.2 Additions

OpenFlow 1.2 [3] 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

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	-
IPv6 support	See Section 5.5.2
Simplified behavior of flow-mod request	-
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 hardcoded 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 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 various fields are 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 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 we described 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, 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 we described 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.
- An instruction was executed in the pipeline, prescribing that a matching packet be forwarded to the controller.
- 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 standalone mode. The notion that the switch would attempt to contact previously configured backup controllers was encouraged by the specification, but it was 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 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 switch be in master mode and all others be in slave mode. The multiple-controllers feature is part of the way 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 [10]. In this example we 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 process 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 to site B of customer A.

Note that although 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 [4] 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 of 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, some 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. The most prominent of these new features are discussed in the following sections.

[3] In Sections 12.9.3 and 13.2.7, we discuss the possibility of ASICs that would allow more general support of these features.

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	-
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

5.6.1 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 the STATS_REQUEST/STATS_REPLY to get this information. Rather than having this capability information embedded as though 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 and capability information. The data formats for the capabilities are the TLV format, and this capability information has been removed from the table statistics structure.

5.6.2 More Flexible Table-Miss Support

We have explained that a table miss occurs 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 on 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 wildcards 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*. Figure 5.16 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 Figure 5.16, there may be multiple *meter bands* attached to a given meter. Meter 3 in the example in the figure has three meter

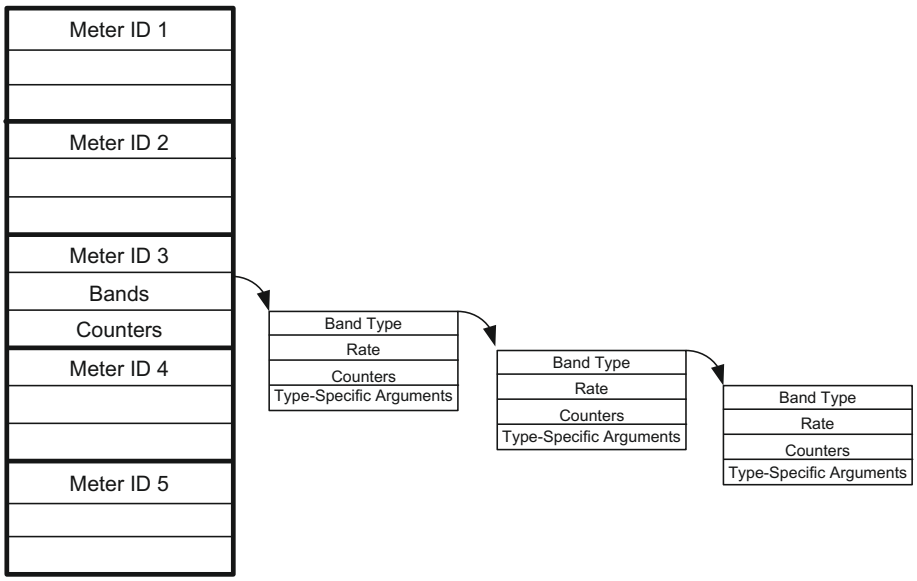


FIGURE 5.16

OpenFlow V.1.3 meter table.

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) 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* [6] 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 shown in Figure 5.16, 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 the 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 want 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, mutiple parallel communication *channels* existed from a single switch to mutiple 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 Figure 5.17. The figure shows that there are also multiple channels

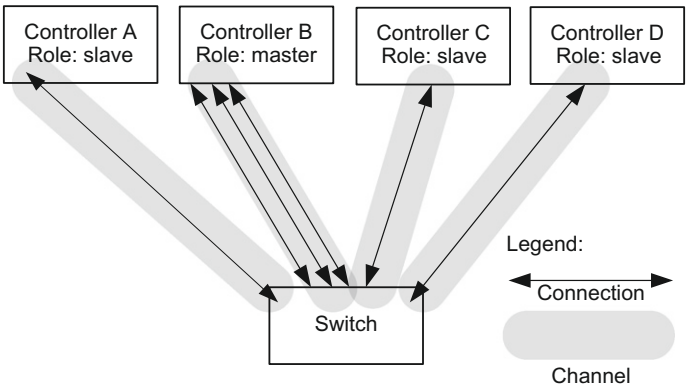


FIGURE 5.17

Multiple connections and multiple channels from a switch.

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 for the connection to 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, unreliable connection types, 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 the 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 the existing flow to which this packet relates, 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 lookup 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. 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 caches 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 compared to the basic flow entry we presented in Figure 5.6. The V.1.3.0 flow entry is depicted in Figure 5.18. The header, counter, and actions fields were present in Figure 5.6 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 wants the flow to be removed after a specified amount of time without matching any packets, then the idle timer is used. If the controller wants 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.

Header Fields	Field value
Priority	Field value
Counters	Field value
Actions/Instructions	Field value
Timeouts	Field value
Cookie	Field value

FIGURE 5.18

V.1.3 flow entry.

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 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 to the related standard in [8]. This is a useful technology that's already supported by high-end commercial switches; 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

Figure 5.19 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

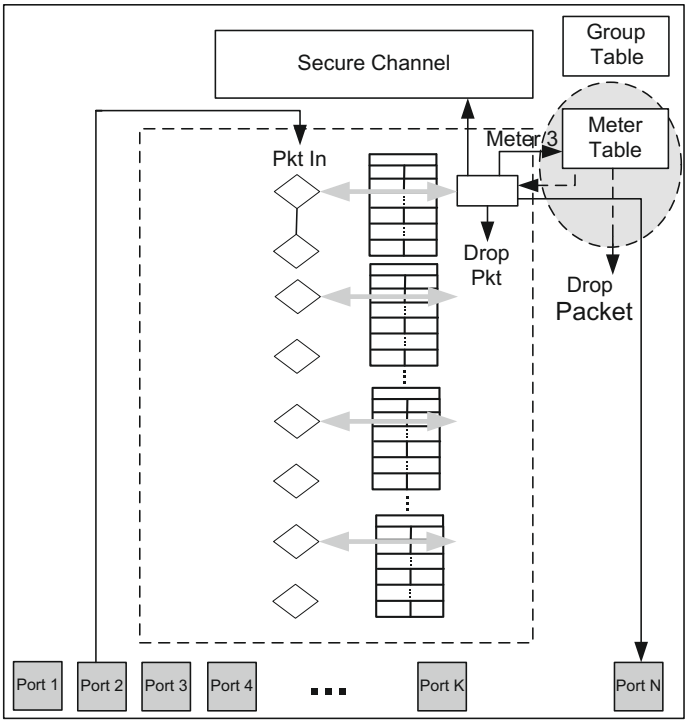


FIGURE 5.19

Use of meters for QoS control in V.1.3.

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 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.

5.7 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), whereby fields in the packet's payload may be used to distinguish flows, is not supported in a standard OpenFlow. Nonetheless, the EXPERIMENTER modes that are permitted within OpenFlow do open the way for such application layer flow definition in the future. Second, 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, since 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.

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.

5.8 Conclusion

In this chapter we attempted to provide the reader with a high-level understanding of the general OpenFlow framework. This discussion covered the protocol that an OpenFlow controller uses to configure and control an OpenFlow switch. We 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 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 specifications cited in [1–4]. These specifications are, however, very detailed and are not especially easy for the OpenFlow novice to follow. We believe that this chapter serves 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 5.5 in this chapter as a quick reference and an aid to understanding to what part of the protocol a specific reference pertains.

Table 5.5 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

References

[1] OpenFlow switch specification, Version 1.0.0 (wire protocol 0x01). Open Networking Foundation; December 31, 2009. Retrieved from <www.opennetworking.org/sdn-resources/onf-specifications>.

[2] OpenFlow switch specification, Version 1.1.0 (wire protocol 0x02). Open Networking Foundation; February 28, 2011. Retrieved from <www.opennetworking.org/sdn-resources/onf-specifications>.

[3] OpenFlow switch specification, Version 1.2.0 (wire protocol 0x03). Open Networking Foundation; December 5, 2011. Retrieved from <www.opennetworking.org/sdn-resources/onf-specifications>.

[4] OpenFlow switch specification, Version 1.3.0 (wire protocol 0x04). Open Networking Foundation; June 25, 2012. Retrieved from <www.opennetworking.org/sdn-resources/onf-specifications>.

[5] Davie B, Doolan P, Rekhter Y. Switching in IP networks. San Francisco,CA, USA: Morgan Kaufmann; 1998.

[6] Shenker S, Partridge C, Guerin R. Specification of guaranteed quality of service, RFC 2212. Internet Engineering Task Force; 1997.
[7] IEEE standard for local and metropolitan area networks: link aggregation, IEEE 802.1AX. USA: IEEE Computer Society; November 3, 2008.
[8] IEEE Draft standard for local and metropolitan area networks–virtual bridged local area networks–amendment 6: provider backbone bridges, IEEE P802.1ah/D4.2. New York, NY, USA: IEEE; March 2008.
[9] 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. IEEE; July 2012.
[10] IEEE standard for local and metropolitan area networks: media access control (MAC) bridges and virtual bridged local area networks, IEEE 802.1Q. New York, NY, USA: IEEE; August 2011.