5.4 OpenFlow 1.1 ADDITIONS

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

5.4.1 MULTIPLE FLOW TABLES

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

Table 5.2 Major New Features Added in OpenFlow 1.1

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

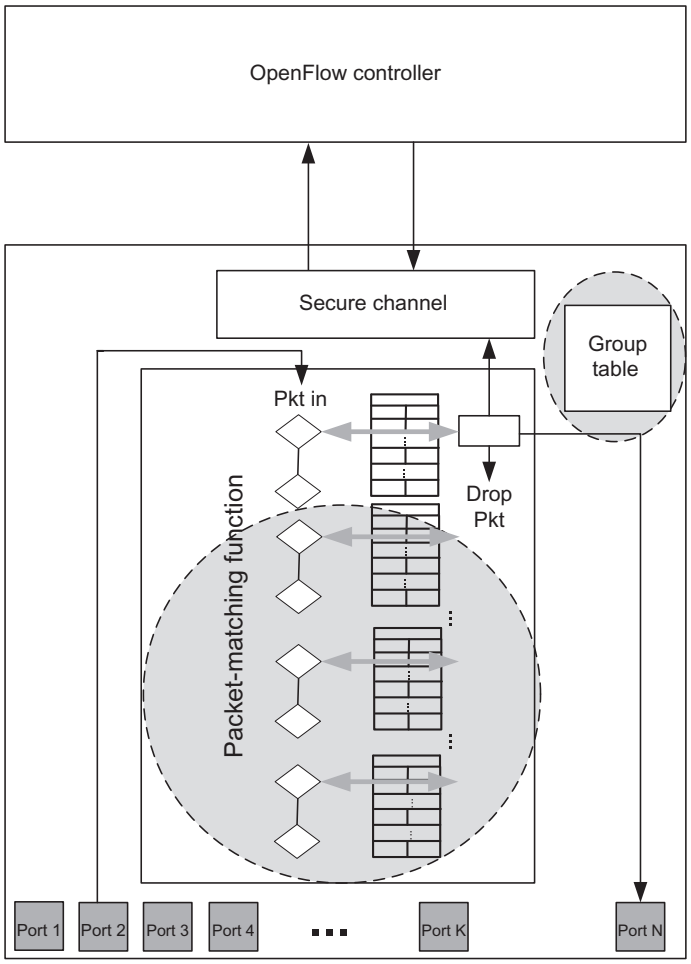


FIG. 5.11

OpenFlow V.1.1 switch with expanded packet processing.

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

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

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

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

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

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

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

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

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

5.4.2 GROUPS

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

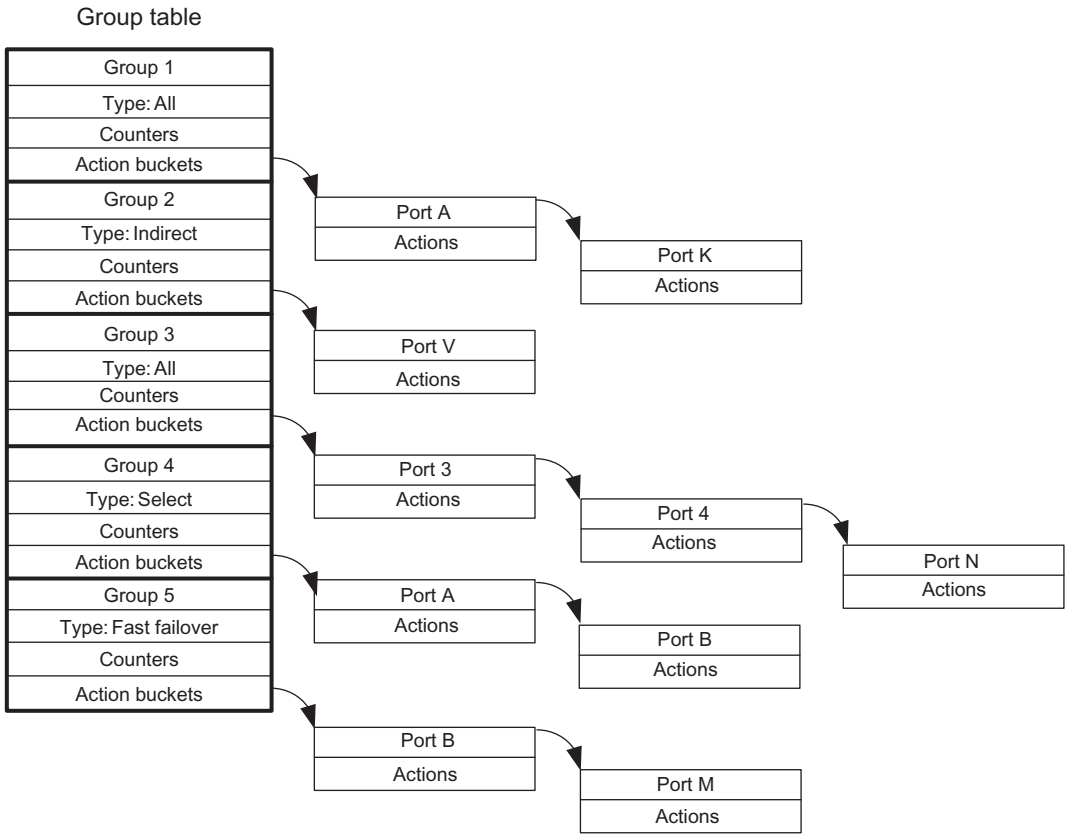


FIG. 5.12

OpenFlow V.1.1 group table.

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

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

5.4.3 MPLS AND VLAN TAG SUPPORT

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

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

5.4.4 VIRTUAL PORTS

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

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

Reserved virtual ports consist of:

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

5.4.5 CONTROLLER CONNECTION FAILURE

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

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

5.4.6 EXAMPLE: FORWARDING WITH MULTIPLE FLOW TABLES

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

5.4.7 EXAMPLE: MULTICAST USING V.1.1 GROUPS

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

DISCUSSION QUESTION:

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

5.5 OpenFlow 1.2 ADDITIONS

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

5.5.1 EXTENSIBLE MATCH SUPPORT

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

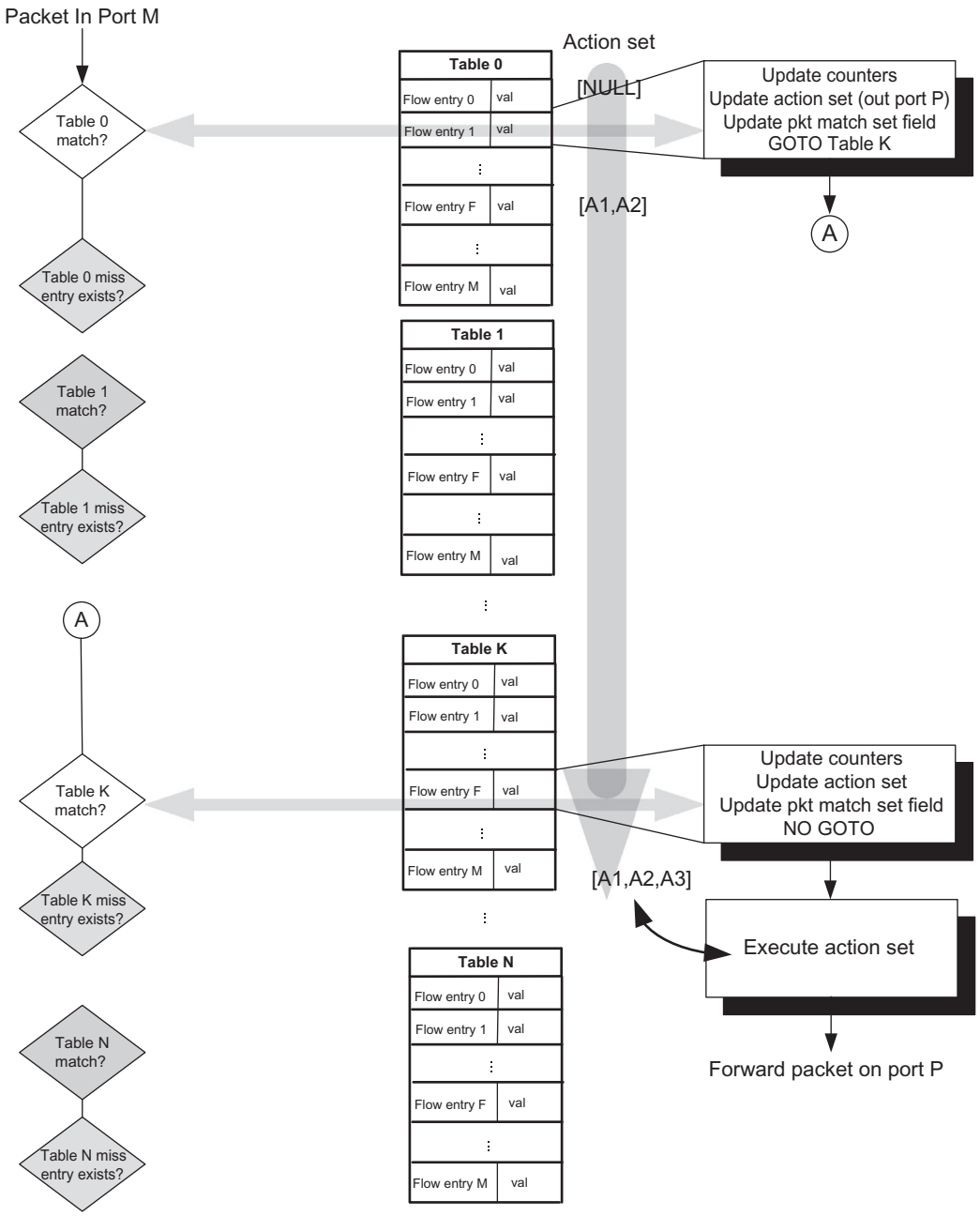


FIG. 5.13

Packet matching function—basic packet forwarding V.1.1.

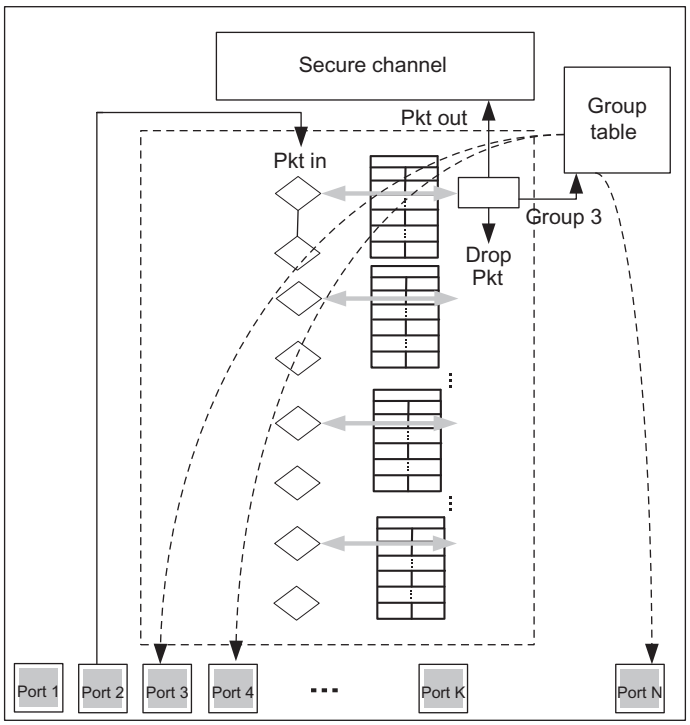


FIG. 5.14

Multicast using group table in V.1.1.

Table 5.3 Major New Features Added in OpenFlow 1.2

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

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

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

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

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

5.5.2 EXTENSIBLE SET_FIELD PACKET REWRITING SUPPORT

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

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

5.5.3 EXTENSIBLE CONTEXT EXPRESSION IN PACKET_IN

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

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

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

5.5.4 MULTIPLE CONTROLLERS

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

- Equal
- Slave
- Master

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

5.5.5 EXAMPLE: BRIDGING VLANS THROUGH SP NETWORKS

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

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

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

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