NAME

ovs-actions − OpenFlow actions and instructions with Open vSwitch extensions

INTRODUCTION

This document aims to comprehensively document all of the OpenFlow actions and instructions, both standard and non−standard, supported by Open vSwitch, regardless of origin. The document includes information of interest to Open vSwitch users, such as the semantics of each supported action and the syntax used by Open vSwitch tools, and to developers seeking to build controllers and switches compatible with Open vSwitch, such as the wire format for each supported message.

Actions

In this document, we define an **action** as an OpenFlow action, which is a kind of command that specifies what to do with a packet. Actions are used in OpenFlow flows to describe what to do when the flow matches a packet, and in a few other places in OpenFlow. Each version of the OpenFlow specification defines standard actions, and beyond that many OpenFlow switches, including Open vSwitch, implement extensions to the standard.

OpenFlow groups actions in two ways: as an **action list** or an **action set**, described below.

Action Lists

An **action list**, a concept present in every version of OpenFlow, is simply an ordered sequence of actions. The OpenFlow specifications require a switch to execute actions within an action list in the order specified, and to refuse to execute an action list entirely if it cannot implement the actions in that order [OpenFlow 1.0, section 3.3], with one exception: when an action list outputs multiple packets, the switch may output the packets in an order different from that specified. Usually, this exception is not important, especially in the common case when the packets are output to different ports.

Action Sets

OpenFlow 1.1 introduced the concept of an **action set**. An action set is also a sequence of actions, but the switch reorders the actions and drops duplicates according to rules specified in the OpenFlow specifications. Because of these semantics, some standard OpenFlow actions cannot usefully be included in an action set. For some, but not all, Open vSwitch extension actions, Open vSwitch defines its own action set semantics and ordering.

The OpenFlow pipeline has an action set associated with it as a packet is processed. After pipeline processing is otherwise complete, the switch executes the actions in the action set.

Open vSwitch applies actions in an action set in the following order: Except as noted otherwise below, the action set only executes at most a single action of each type, and when more than one action of a given type is present, the one added to the set later replaces the earlier action:

1. **strip_vlan**
2. **pop_mpls**
3. **decap**
4. **encap**
5. **push_mpls**
6. **push_vlan**
7. **dec_ttl**
8. **dec_mpls_ttl**
9. **dec_nsh_ttl**
10. All of the following actions are executed in the order added to the action set, with cumulative effect. That is, when multiple actions modify the same part of a field, the later modification takes effect, and when they modify different parts of a field (or different fields), then both modifications are applied:

- **load**
- **move**
- **mod_dl_dst**
- **mod_dl_src**
- **mod_nw_dst**
- **mod_nw_src**
- **mod_nw_tos**
- **mod_nw_ecn**
- **mod_nw_ttl**
- **mod_tp_dst**
- **mod_tp_src**
- **mod_vlan_pcp**
- **mod_vlan_vid**
- **set_field**
- **set_tunnel**
- **set_tunnel64**

11. **set_queue**
12. **group**, **output**, **resubmit**, **ct_clear**, or **ct**. If more than one of these actions is present, then the one listed earliest above is executed and the others are ignored, regardless of the order in which they were added to the action set. (If none of these actions is present, the action set has no real effect, because the modified packet is not sent anywhere and thus the modifications are not visible.)

An action set may only contain the actions listed above.

Error Handling

Packet processing can encounter a variety of errors:

Bridge not found

Open vSwitch supports an extension to the standard OpenFlow **controller** action called a **continuation**, which allows the controller to interrupt and later resume the processing of a packet through the switch pipeline. This error occurs when such a packet's processing cannot be resumed, e.g. because the bridge processing it has been destroyed. Open vSwitch reports this error to the controller as Open vSwitch extension error **NXR_STALE**.

This error prevents packet processing entirely.

Recursion too deep

While processing a given packet, Open vSwitch limits the flow table recursion depth to 64, to ensure that packet processing uses a finite amount of time and space. Actions that count against the recursion limit include **resubmit** from a given OpenFlow table to the same or an earlier table, **group**, and **output** to patch ports.

A **resubmit** from one table to a later one (or, equivalently, a **goto_table** instruction) does not count against the depth limit because resubmits to strictly monotonically increasing tables will eventually terminate. OpenFlow tables are most commonly traversed in numerically increasing order, so this limit has little effect on conventionally designed OpenFlow pipelines.

This error terminates packet processing. Any previous side effects (e.g. output actions) are retained.

Usually this error indicates a loop or other bug in the OpenFlow flow tables. To assist debugging, when this error occurs, Open vSwitch 2.10 and later logs a trace of the packet execution, as if by **ovs−appctl ofproto/trace**, rate−limited to one per minute to reduce the log volume.

Too many resubmits

Open vSwitch limits the total number of **resubmit** actions that a given packet can execute to 4,096. For this purpose, **goto_table** instructions and output to the **table** port are treated like **resubmit**. This limits the amount of time to process a single packet.

Unlike the limit on recursion depth, the limit on resubmits counts all resubmits, regardless of direction.

This error has the same effect, including logging, as exceeding the recursion depth limit.

Stack too deep

Open vSwitch limits the amount of data that the **push** action can put onto the stack at one time to 64 kB of data.

This error terminates packet processing. Any previous side effects (e.g. output actions) are retained.

No recirculation context / Recirculation conflict

These errors indicate internal errors inside Open vSwitch and should generally not occur. If you notice recurring log messages about these errors, please report a bug.

Too many MPLS labels

Open vSwitch can process packets with any number of MPLS labels, but its ability to push and pop MPLS labels is limited, currently to 3 labels. Attempting to push more than the supported number of labels onto a packet, or to pop any number of labels from a packet with more than the supported number, raises this error.

This error terminates packet processing, retaining any previous side effects (e.g. output actions). When this error arises within the execution of a group bucket, it only terminates that bucket's execution, not packet processing overall.

Invalid tunnel metadata

Open vSwitch raises this error when it processes a Geneve packet that has TLV options with an invalid form, e.g. where the length in a TLV would extend past the end of the options.

This error prevents packet processing entirely.

Unsupported packet type

When a **encap** action encapsulates a packet, Open vSwitch raises this error if it does not support the combination of the new encapsulation with the current packet. **encap(ethernet)** raises this error if the current packet is not an L3 packet, and **encap(nsh)** raises this error if the current packet is not Ethernet, IPv4, IPv6, or NSH.

The **decap** action is supported only for packet types ethernet, NSH and MPLS. Openvswitch raises this error for other packet types. When a **decap** action decapsulates a packet, Open vSwitch raises this error if it does not support the type of inner packet. **decap** of an Ethernet header raises this error if a VLAN header is present, **decap** of a NSH packet raises this error if the NSH inner packet is not Ethernet, IPv4, IPv6, or NSH.

This error terminates packet processing, retaining any previous side effects (e.g. output actions). When this error arises within the execution of a group bucket, it only terminates that bucket's execution, not packet processing overall.

Inconsistencies

OpenFlow 1.0 allows any action to be part of any flow, regardless of the flow's match. Some combinations do not make sense, e.g. an **set_nw_tos** action in a flow that matches only ARP packets or **strip_vlan** in a flow that matches packets without VLAN tags. Other combinations have varying results depending on the kind of packet that the flow processes, e.g. a **set_nw_src** action in a flow that does not match on Ethertype will be treated as a no−op when it processes a non−IPv4 packet. Nevertheless OVS allows all of the above in conformance with OpenFlow 1.0, that is, the following will succeed:

```
$ ovs-ofctl -O OpenFlow10 add-flow br0 arp,actions=mod_nw_tos:12
$ ovs-ofctl -O OpenFlow10 add-flow br0 dl_vlan=0xffff,actions=strip_vlan
$ ovs-ofctl -O OpenFlow10 add-flow br0 actions=mod_nw_src:1.2.3.4
```

Open vSwitch calls these kinds of combinations **inconsistencies** between match and actions. OpenFlow 1.1 and later forbid inconsistencies, and disallow the examples described above by preventing such flows from being added. All of the above, for example, will fail with an error message if one replaces **OpenFlow10** by **OpenFlow11**.

OpenFlow 1.1 and later cannot detect and disallow all inconsistencies. For example, the **write_actions** instruction arbitrarily delays execution of the actions inside it, which can even be canceled with **clear_actions**, so that there is no way to ensure that its actions are consistent with the packet at the time they execute. Thus, actions with **write_actions** and some other contexts are exempt from consistency requirements.

When OVS executes an action inconsistent with the packet, it treats it as a no−op.

Inter−Version Compatibility

Open vSwitch supports multiple OpenFlow versions simultaneously on a single switch. When actions are added with one OpenFlow version and then retrieved with another, Open vSwitch does its best to translate between them.

Inter−version compatibility issues can still arise when different connections use different OpenFlow versions. Backward compatibility is the most obvious case. Suppose, for example, that an OpenFlow 1.1 session adds a flow with a **push_vlan** action, for which there is no equivalent in OpenFlow 1.0. If an OpenFlow 1.0 session retrieves this flow, Open vSwitch must somehow represent the action.

Forward compatibility can also be an issue, because later OpenFlow versions sometimes remove functionality. The best example is the **enqueue** action from OpenFlow 1.0, which OpenFlow 1.1 removed.

In practice, Open vSwitch uses a variety of strategies for inter−version compatibility:

- Most standard OpenFlow actions, such as **output** actions, translate without compatibility issues.
- Open vSwitch supports its extension actions in every OpenFlow version, so they do not pose inter−version compatibility problems.
- Open vSwitch sometimes adds extension actions to ensure backward or forward compatibility. For example, for backward compatibility with the **group** action added in OpenFlow 1.1, Open vSwitch includes an OpenFlow 1.0 extension **group** action.

Perfect inter−version compatibility is not possible, so best results require OpenFlow connections to use a consistent version. One may enforce use of a particular version by setting the **protocols** column for a bridge, e.g. to force **br0** to use only OpenFlow 1.3:

```
ovs-vsctl set bridge br0 protocols=OpenFlow13
```

Field Specifications

Many Open vSwitch actions refer to fields. In such cases, fields may usually be referred to by their common names, such as **eth_dst** for the Ethernet destination field, or by their full OXM or NXM names, such

as **NXM_OF_ETH_DST** or **OXM_OF_ETH_DST**. Before Open vSwitch 2.7, only OXM or NXM field names were accepted.

Many actions that act on fields can also act on **subfields**, that is, parts of fields, written as **field[start..end]**, where **start** is the first bit and **end** is the last bit to use in **field**, e.g. **vlan_tci[13..15]** for the VLAN PCP. A single−bit subfield may also be written as **field[offset]**, e.g. **vlan_tci[13]** for the least−significant bit of the VLAN PCP. Empty brackets may be used to explicitly designate an entire field, e.g. **vlan_tci[]** for the entire 16−bit VLAN TCI header. Before Open vSwitch 2.7, brackets were required in field specifications.

See **ovs−fields(7)** for a list of fields and their names.

Port Specifications

Many Open vSwitch actions refer to OpenFlow ports. In such cases, the port may be specified as a numeric port number in the range 0 to 65,535, although Open vSwitch only assigns port numbers in the range 1 through 62,279 to ports. OpenFlow 1.1 and later use 32−bit port numbers, but Open vSwitch never assigns a port number that requires more than 16 bits.

In most contexts, the name of a port may also be used. (The most obvious context where a port name may not be used is in an **ovs−ofctl** command along with the **−−no−names** option.) When a port's name contains punctuation or could be ambiguous with other actions, the name may be enclosed in double quotes, with JSON−like string escapes supported (see [RFC 8259]).

Open vSwitch also supports the following standard OpenFlow port names (even in contexts where port names are not otherwise supported). The corresponding OpenFlow 1.0 and 1.1+ port numbers are listed alongside them but should not be used in flow syntax:

- **in_port** (65528 or 0xfff8; 0xfffffff8)
- **table** (65529 or 0xfff9; 0xfffffff9)
- **normal** (65530 or 0xfffa; 0xfffffffa)
- **flood** (65531 or 0xfffb; 0xfffffffb)
- **all** (65532 or 0xfffc; 0xfffffffc)
- **controller** (65533 or 0xfffd; 0xfffffffd)
- **local** (65534 or 0xfffe; 0xfffffffe)
- **any** or **none** (65535 or 0xffff; 0xffffffff)
- **unset** (not in OpenFlow 1.0; 0xfffffff7)

OUTPUT ACTIONS

These actions send a packet to a physical port or a controller. A packet that never encounters an output action on its trip through the Open vSwitch pipeline is effectively dropped. Because actions are executed in order, a packet modification action that is not eventually followed by an output action will not have an externally visible effect.

The output action

Syntax:

port
output:*port*
output:*field*
output(port=*port***, max_len=***nbytes***)**

Outputs the packet to an OpenFlow port most commonly specified as *port*. Alternatively, the output port may be read from *field*, a field or subfield in the syntax described under *Field Specifications* above. Either way, if the port is the packet's input port, the packet is not output.

The *port* may be one of the following standard OpenFlow ports:

local Outputs the packet on the **local port** that corresponds to the network device that has the same name as the bridge, unless the packet was received on the local port. OpenFlow switch implementations are not required to have a local port, but Open vSwitch bridges always do.

in_port
Outputs the packet on the port on which it was received. This is the only standard way to output the packet to the input port (but see *Output to the Input port*, below).

The *port* may also be one of the following additional OpenFlow ports, unless **max_len** is specified:

normal
Subjects the packet to the device's normal L2/L3 processing. This action is not implemented by all OpenFlow switches, and each switch implements it differently. The section *The OVS Normal Pipeline* below documents the OVS implementation.

flood Outputs the packet on all switch physical ports, except the port on which it was received and any ports on which flooding is disabled. Flooding can be disabled automatically on a port by Open vSwitch when IEEE 802.1D spanning tree (STP) or rapid spanning tree (RSTP) is enabled, or by a controller using an OpenFlow **OFPT_MOD_PORT** request to set the port's **OFPPC_NO_FLOOD** flag (**ovs−ofctl mod−port** provides a command−line interface to set this flag).

all Outputs the packet on all switch physical ports except the port on which it was received.

controller
Sends the packet and its metadata to an OpenFlow controller or controllers encapsulated in an OpenFlow **packet−in** message. The separate **controller** action, described below, provides more options for output to a controller.

Open vSwitch rejects output to other standard OpenFlow ports, including **none**, **unset**, and port numbers reserved for future use as standard ports, with the error **OFPBAC_BAD_OUT_PORT**.

With **max_len**, the packet is truncated to at most *nbytes* bytes before being output. In this case, the output port may not be a patch port. Truncation is just for the single output action, so that later actions in the OpenFlow pipeline work with the complete packet. The truncation feature is meant for use in monitoring applications, e.g. for mirroring packets to a collector.

When an **output** action specifies the number of a port that does not currently exist (and is not in the range for standard ports), the OpenFlow specification allows but does not require OVS to reject the action. All versions of Open vSwitch treat such an action as a no−op. If a port with the number is created later, then the action will be honored at that point. (OpenFlow requires OVS to reject output to a port number that will never be valid, with **OFPBAC_BAD_OUT_PORT**, but this situation does not arise when OVS is a software switch, since the user can add or renumber ports at any time.)

A controller can suppress output to a port by setting its **OFPPC_NO_FORWARD** flag using an OpenFlow **OFPT_MOD_PORT** request (**ovs−ofctl mod−port** provides a command−line interface to set this flag). When output is disabled, **output** actions (and other actions that output to the port) are allowed but have no effect.

Open vSwitch allows output to a port that does not exist, although OpenFlow allows switches to reject such actions.

Conformance
All versions of OpenFlow and Open vSwitch support **output** to a literal **port**. Output to a register is an OpenFlow extension introduced in Open vSwitch 1.3. Output with truncation is an OpenFlow extension introduced in Open vSwitch 2.6.

Output to the Input Port

OpenFlow requires a switch to ignore attempts to send a packet out its ingress port in the most straightforward way. For example, **output:234** has no effect if the packet has ingress port 234. The rationale is that dropping these packets makes it harder to loop the network. Sometimes this behavior can even be convenient, e.g. it is often the desired behavior in a flow that forwards a packet to several ports (**floods** the packet).

Sometimes one really needs to send a packet out its ingress port (**hairpin**). In this case, use **in_port** to explicitly output the packet to its input port, e.g.:

```
$ ovs-ofctl add-flow br0 in_port=2,actions=in_port
```

This also works in some circumstances where the flow doesn't match on the input port. For example, if you know that your switch has five ports numbered 2 through 6, then the following will send every received packet out every port, even its ingress port:

```
$ ovs-ofctl add-flow br0 actions=2,3,4,5,6,in_port
```

or, equivalently:

```
$ ovs-ofctl add-flow br0 actions=all,in_port
```

Sometimes, in complicated flow tables with multiple levels of **resubmit** actions, a flow needs to output to a particular port that may or may not be the ingress port. It's difficult to take advantage of output to **in_port** in this situation. To help, Open vSwitch provides, as an OpenFlow extension, the ability to modify the **in_port** field. Whatever value is currently in the **in_port** field is both the port to which output will be dropped and the destination for **in_port**. This means that the following adds flows that reliably output to port 2 or to ports 2 through 6, respectively:

```
$ ovs-ofctl add-flow br0 "in_port=2,actions=load:0->in_port,2"
$ ovs-ofctl add-flow br0 "actions=load:0->in_port,2,3,4,5,6"
```

If **in_port** is important for matching or other reasons, one may save and restore it on the stack:

```
$ ovs-ofctl add-flow br0 \
      actions="push:in_port,load:0->in_port,2,3,4,5,6,pop:in_port"
```

The OVS Normal Pipeline

This section documents how Open vSwitch implements output to the **normal** port. The OpenFlow specification places no requirements on how this port works, so all of this documentation is specific to Open vSwitch.

Open vSwitch uses the **Open_vSwitch** database, detailed in **ovs−vswitchd.conf.db(5)**, to determine the details of the normal pipeline.

The normal pipeline executes the following ingress stages for each packet. Each stage either accepts the packet, in which case the packet goes on to the next stage, or drops the packet, which terminates the pipeline. The result of the ingress stages is a set of output ports, which is the empty set if some ingress stage drops the packet:

1. **Input port lookup**: Looks up the OpenFlow **in_port** field's value to the corresponding **Port** and **Interface** record in the database.

   The **in_port** is normally the OpenFlow port that the packet was received on. If **set_field** or another actions changes the **in_port**, the updated value is honored. Accept the packet if the lookup succeeds, which it normally will. If the lookup fails, for example because **in_port** was changed to an unknown

value, drop the packet.

2. **Drop malformed packet**: If the packet is malformed enough that it contains only part of an 802.1Q header, then drop the packet with an error.

3. **Drop packets sent to a port reserved for mirroring**: If the packet was received on a port that is configured as the output port for a mirror (that is, it is the **output_port** in some **Mirror** record), then drop the packet.

4. **VLAN input processing**: This stage determines what VLAN the packet is in. It also verifies that this VLAN is valid for the port; if not, drop the packet. How the VLAN is determined and which ones are valid vary based on the **vlan−mode** in the input port's **Port** record:

   trunk The packet is in the VLAN specified in its 802.1Q header, or in VLAN 0 if there is no 802.1Q header. The **trunks** column in the **Port** record lists the valid VLANs; if it is empty, all VLANs are valid.

   access The packet is in the VLAN specified in the **tag** column of its **Port** record. The packet must not have an 802.1Q header with a nonzero VLAN ID; if it does, drop the packet.

   native−tagged / native−untagged
   Same as **trunk** except that the VLAN of a packet without an 802.1Q header is not necessarily zero; instead, it is taken from the **tag** column.

   dot1q−tunnel
   The packet is in the VLAN specified in the **tag** column of its **Port** record, which is a QinQ service VLAN with the Ethertype specified by the **Port**'s **other_config:qinq−ethtype**. If the packet has an 802.1Q header, then it specifies the customer VLAN. The **cvlans** column specifies the valid customer VLANs; if it is empty, all customer VLANs are valid.

5. **Drop reserved multicast addresses**: If the packet is addressed to a reserved Ethernet multicast address and the **Bridge** record does not have **other_config:forward−bpdu** set to **true**, drop the packet.

6. **LACP bond admissibility**: This step applies only if the input port is a member of a bond (a **Port** with more than one **Interface**) and that bond is configured to use LACP. Otherwise, skip to the next step.

   The behavior here depends on the state of LACP negotiation:

   - If LACP has been negotiated with the peer, accept the packet if the bond member is enabled (i.e. carrier is up and it hasn't been administratively disabled). Otherwise, drop the packet.
   - If LACP negotiation is incomplete, then drop the packet. There is one exception: if fallback to active−backup mode is enabled, continue with the next step, pretending that the active−backup balancing mode is in use.

7. **Non−LACP bond admissibility**: This step applies if the input port is a member of a bond without LACP configured, or if a LACP bond falls back to active−backup as described in the previous step. If neither of these applies, skip to the next step.

   If the packet is an Ethernet multicast or broadcast, and not received on the bond's active member, drop the packet.

   The remaining behavior depends on the bond's balancing mode:

   L4 (aka TCP balancing)
   Drop the packet (this balancing mode is only supported with LACP).

   Active−backup
   Accept the packet only if it was received on the active member.

SLB (Source Load Balancing)
Drop the packet if the bridge has not learned the packet's source address (in its VLAN) on the port that received it. Otherwise, accept the packet unless it is a gratuitous ARP. Otherwise, accept the packet if the MAC entry we found is ARP−locked. Otherwise, drop the packet. (See the **SLB Bonding** section in the OVS bonding document for more information and a rationale.)

8. **Learn source MAC**: If the source Ethernet address is not a multicast address, then insert a mapping from packet's source Ethernet address and VLAN to the input port in the bridge's MAC learning table. (This is skipped if the packet's VLAN is listed in the switch's **Bridge** record in the **flood_vlans** column, since there is no use for MAC learning when all packets are flooded.)

   When learning happens on a non−bond port, if the packet is a gratuitous ARP, the entry is marked as ARP−locked. The lock expires after 5 seconds. (See the **SLB Bonding** section in the OVS bonding document for more information and a rationale.)

9. **IP multicast path**: If multicast snooping is enabled on the bridge, and the packet is an Ethernet multicast but not an Ethernet broadcast, and the packet is an IP packet, then the packet takes a special processing path. This path is not yet documented here.

10. **Output port set**: Search the MAC learning table for the port corresponding to the packet's Ethernet destination and VLAN. If the search finds an entry, the output port set is just the learned port. Otherwise (including the case where the packet is an Ethernet multicast or in **flood_vlans**), the output port set is all of the ports in the bridge that belong to the packet's VLAN, except for any ports that were disabled for flooding via OpenFlow or that are configured in a **Mirror** record as a mirror destination port.

The following egress stages execute once for each element in the set of output ports. They execute (conceptually) in parallel, so that a decision or action taken for a given output port has no effect on those for another one:

1. **Drop loopback**: If the output port is the same as the input port, drop the packet.

2. **VLAN output processing**: This stage adjusts the packet to represent the VLAN in the correct way for the output port. Its behavior varies based on the **vlan−mode** in the output port's **Port** record:

   trunk / native−tagged / native−untagged
   If the packet is in VLAN 0 (for **native−untagged**, if the packet is in the native VLAN) drops any 802.1Q header. Otherwise, ensures that there is an 802.1Q header designating the VLAN.

   access Remove any 802.1Q header that was present.

   dot1q−tunnel
   Ensures that the packet has an outer 802.1Q header with the QinQ Ethertype and the specified configured tag, and an inner 802.1Q header with the packet's VLAN.

3. **VLAN priority tag processing**: If VLAN output processing discarded the 802.1Q headers, but priority tags are enabled with **other_config:priority−tags** in the output port's **Port** record, then a priority−only tag is added (perhaps only if the priority would be nonzero, depending on the configuration).

4. **Bond member choice**: If the output port is a bond, the code chooses a particular member. This step is skipped for non−bonded ports.

   If the bond is configured to use LACP, but LACP negotiation is incomplete, then normally the packet is dropped. The exception is that if fallback to active−backup mode is enabled, the egress pipeline continues choosing a bond member as if active−backup mode was in use.

   For active−backup mode, the output member is the active member. Other modes hash appropriate header fields and use the hash value to choose one of the enabled members.

5. **Output**: The pipeline sends the packet to the output port.

The controller action

Syntax:

controller
controller:*max_len*
controller(*key***[=***value***], ...)**

Sends the packet and its metadata to an OpenFlow controller or controllers encapsulated in an OpenFlow **packet−in** message. The supported options are:

max_len=*max_len*
Limit to *max_len* the number of bytes of the packet to send in the **packet−in.** A *max_len* of 0 prevents any of the packet from being sent (thus, only metadata is included). By default, the entire packet is sent, equivalent to a *max_len* of 65535.

reason=*reason*
Specify *reason* as the reason for sending the message in the **packet−in**. The supported reasons are **no_match**, **action**, **invalid_ttl**, **action_set**, **group**, and **packet_out**. The default reason is **action**.

id=*controller_id*
Specify *controller_id*, a 16−bit integer, as the connection ID of the OpenFlow controller or controllers to which the **packet−in** message should be sent. The default is zero. Zero is also the default connection ID for each controller connection, and a given controller connection will only have a nonzero connection ID if its controller uses the **NXT_SET_CONTROLLER_ID** Open vSwitch extension to OpenFlow.

userdata=*hh***...**
Supplies the bytes represented as hex digits *hh* as additional data to the controller in the **packet−in** message. Pairs of hex digits may be separated by periods for readability.

pause Causes the switch to freeze the packet's trip through Open vSwitch flow tables and serializes that state into the packet−in message as a **continuation,** an additional property in the **NXT_PACKET_IN2** message. The controller can later send the continuation back to the switch in an **NXT_RESUME** message, which will restart the packet's traversal from the point where it was interrupted. This permits an OpenFlow controller to interpose on a packet midway through processing in Open vSwitch.

Conformance

All versions of OpenFlow and Open vSwitch support **controller** action and its **max_len** option. The **userdata** and **pause** options require the Open vSwitch **NXAST_CONTROLLER2** extension action added in Open vSwitch 2.6. In the absence of these options, the **reason** (other than **reason=action**) and **controller_id** (option than **controller_id=0**) options require the Open vSwitch **NXAST_CONTROLLER** extension action added in Open vSwitch 1.6.

The enqueue action

Syntax:

enqueue(*port***,***queue***)**
enqueue:*port***:***queue*

Enqueues the packet on the specified *queue* within port *port*.

port must be an OpenFlow port number or name as described under *Port Specifications* above. *port* may be **in_port** or **local** but the other standard OpenFlow ports are not allowed.

queue must be a number between 0 and 4294967294 (0xfffffffe), inclusive. The number of actually

supported queues depends on the switch. Some OpenFlow implementations do not support queuing at all. In Open vSwitch, the supported queues vary depending on the operating system, datapath, and hardware in use. Use the **QoS** and **Queue** tables in the Open vSwitch database to configure queuing on individual OpenFlow ports (see **ovs−vswitchd.conf.db(5)** for more information).

Conformance

Only OpenFlow 1.0 supports **enqueue**. OpenFlow 1.1 added the **set_queue** action to use in its place along with **output**.

Open vSwitch translates **enqueue** to a sequence of three actions in OpenFlow 1.1 or later: **set_queue:***queue*,**output:***port*,**pop_queue**. This is equivalent in behavior as long as the flow table does not otherwise use **set_queue**, but it relies on the **pop_queue** Open vSwitch extension action.

The bundle and bundle_load actions

Syntax:

bundle(*fields*,*basis*,*algorithm*,**ofport,members:***port*...**)**
bundle_load(*fields*,*basis*,*algorithm*,**ofport,***dst*,**members:***port*...**)**

These actions choose a port (a **member**) from a comma−separated OpenFlow *port* list. After selecting the port, **bundle** outputs to it, whereas **bundle_load** writes its port number to *dst*, which must be a 16−bit or wider field or subfield in the syntax described under *Field Specifications* above.

These actions hash a set of *fields* using *basis* as a universal hash parameter, then apply the bundle link selection *algorithm* to choose a *port*.

fields must be one of the following. For the options with **symmetric** in the name, reversing source and destination addresses yields the same hash:

eth_src Ethernet source address.

nw_src IPv4 or IPv6 source address.

nw_dst IPv4 or IPv6 destination address.

symmetric_l4
Ethernet source and destination, Ethernet type, VLAN ID or IDs (if any), IPv4 or IPv6 source and destination, IP protocol, TCP or SCTP (but not UDP) source and destination.

symmetric_l3l4
IPv4 or IPv6 source and destination, IP protocol, TCP or SCTP (but not UDP) source and destination.

symmetric_l3l4+udp
Like **symmetric_l3l4** but include UDP ports.

algorithm must be one of the following:

active_backup
Chooses the first live port listed in **members**.

hrw (Highest Random Weight)
Computes the following, considering only the live ports in **members**:

```
for i in [1, n_members]:
    weights[i] = hash(flow, i)
member = { i such that weights[i] >= weights[j] for all j != i }
```

This algorithm is specified by RFC 2992.

The algorithms take port liveness into account when selecting members. The definition of whether a port is live is subject to change. It currently takes into account carrier status and link monitoring protocols such as BFD and CFM. If none of the members is live, **bundle** does not output the packet and **bundle_load** stores **OFPP_NONE** (65535) in the output field.

Example: **bundle(eth_src,0,hrw,ofport,members:4,8)** uses an Ethernet source hash with basis 0, to select between OpenFlow ports 4 and 8 using the Highest Random Weight algorithm.

Conformance

Open vSwitch 1.2 introduced the **bundle** and **bundle_load** OpenFlow extension actions.

The group action

Syntax:

group:*group*

Outputs the packet to the OpenFlow group *group*, which must be a number in the range 0 to 4294967040 (0xffffff00). The group must exist or Open vSwitch will refuse to add the flow. When a group is deleted, Open vSwitch also deletes all of the flows that output to it.

Groups contain action sets, whose semantics are described above in the section *Action Sets*. The semantics of action sets can be surprising to users who expect action list semantics, since action sets reorder and sometimes ignore actions.

A **group** action usually executes the action set or sets in one or more group buckets. Open vSwitch saves the packet and metadata before it executes each bucket, and then restores it afterward. Thus, when a group executes more than one bucket, this means that each bucket executes on the same packet and metadata. Moreover, regardless of the number of buckets executed, the packet and metadata are the same before and after executing the group.

Sometimes saving and restoring the packet and metadata can be undesirable. In these situations, workarounds are possible. For example, consider a pipeline design in which a **select** group bucket is to communicate to a later stage of processing a value based on which bucket was selected. An obvious design would be for the bucket to communicate the value via **set_field** on a register. This does not work because registers are part of the metadata that **group** saves and restores. The following alternative bucket designs do work:

- Recursively invoke the rest of the pipeline with **resubmit**.
- Use **resubmit** into a table that uses **push** to put the value on the stack for the caller to **pop** off. This works because **group** preserves only packet data and metadata, not the stack.

  (This design requires indirection through **resubmit** because actions sets may not contain **push** or **pop** actions.)

An **exit** action within a group bucket terminates only execution of that bucket, not other buckets or the overall pipeline.

Conformance

OpenFlow 1.1 introduced **group**. Open vSwitch 2.6 and later also supports **group** as an extension to OpenFlow 1.0.

ENCAPSULATION AND DECAPSULATION ACTIONS

The strip_vlan and pop actions

Syntax:

strip_vlan
pop_vlan

Removes the outermost VLAN tag, if any, from the packet.

The two names for this action are synonyms with no semantic difference. The OpenFlow 1.0 specification uses the name **strip_vlan** and later versions use **pop_vlan**, but OVS accepts either name regardless of version.

In OpenFlow 1.1 and later, consistency rules allow **strip_vlan** only in a flow that matches only packets with a VLAN tag (or following an action that pushes a VLAN tag, such as **push_vlan**). See *Inconsistencies*, above, for more information.

Conformance

All versions of OpenFlow and Open vSwitch support this action.

The push_vlan action

Syntax:

push_vlan:*ethertype*

Pushes a new outermost VLAN onto the packet. Uses TPID *ethertype*, which must be **0x8100** for an 802.1Q C−tag or **0x88a8** for a 802.1ad S−tag.

Conformance

OpenFlow 1.1 and later supports this action. Open vSwitch 2.8 added support for multiple VLAN tags (with a limit of 2) and 802.1ad S−tags.

The push_mpls action

Syntax:

push_mpls:*ethertype*

Pushes a new outermost MPLS label stack entry (LSE) onto the packet and changes the packet's Ethertype to *ethertype*, which must be either **B0x8847** or **0x8848**. If the packet did not already contain any MPLS labels, initializes the new LSE as:

Label 2, if the packet contains IPv6, 0 otherwise.

TC The low 3 bits of the packet's DSCP value, or 0 if the packet is not IP.

TTL Copied from the IP TTL, or 64 if the packet is not IP.

If the packet did already contain an MPLS label, initializes the new outermost label as a copy of the existing outermost label.

OVS currently supports at most 3 MPLS labels.

This action applies only to Ethernet packets.

Conformance

Open vSwitch 1.11 introduced support for MPLS. OpenFlow 1.1 and later support **push_mpls**. Open vSwitch implements **push_mpls** as an extension to OpenFlow 1.0.

The pop_mpls action

Syntax:

pop_mpls:*ethertype*

Strips the outermost MPLS label stack entry and changes the packet's Ethertype to *ethertype*. This action applies only to Ethernet packets with at least one MPLS label. If there is more than one MPLS label, then *ethertype* should be an MPLS Ethertype (**B0x8847** or **0x8848**).

Conformance

Open vSwitch 1.11 introduced support for MPLS. OpenFlow 1.1 and later support **pop_mpls**. Open vSwitch implements **pop_mpls** as an extension to OpenFlow 1.0.

The encap action

Syntax:

encap(nsh([md_type=*md_type***], [tlv(***class***,***type***,***value***)]...))**
encap(ethernet)
encap(mpls)
encap(mpls_mc)

The **encap** action encapsulates a packet with a specified header. It has variants for different kinds of encapsulation.

The **encap(nsh(...))** variant encapsulates an Ethernet frame with NSH. The *md_type* may be **1** or **2** for metadata type 1 or 2, defaulting to 1. For metadata type 2, TLVs may be specified with *class* as a 16−bit hexadecimal integer beginning with **0x**, *type* as an 8−bit decimal integer, and *value* a sequence of pairs of hex digits beginning with **0x**. For example:

encap(nsh(md_type=1))

Encapsulates the packet with an NSH header with metadata type 1.

encap(nsh(md_type=2,tlv(0x1000,10,0x12345678)))

Encapsulates the packet with an NSH header, NSH metadata type 2, and an NSH TLV with class 0x1000, type 10, and the 4−byte value 0x12345678.

The **encap(ethernet)** variant encapsulate a bare L3 packet in an Ethernet frame. The Ethernet type is initialized to the L3 packet's type, e.g. 0x0800 if the L3 packet is IPv4. The Ethernet source and destination are initially zeroed.

The **encap(mpls)** variant adds a MPLS header at the start of the packet. When encap(ethernet) is applied after this action, the ethertype of ethernet header will be populated with MPLS unicast ethertype (**0x8847**).

The **encap(mpls_mc)** variant adds a MPLS header at the start of the packet. When encap(ethernet) is applied after this action, the ethertype of ethernet header will be populated with MPLS multicast ethertype (**0x8848**).

Conformance

This action is an Open vSwitch extension to OpenFlow 1.3 and later, introduced in Open vSwitch 2.8.

The MPLS support for this action is added in Open vSwitch 2.17.

The decap action

Syntax:

decap
decap(packet_type(ns=*namespace,***type=***type***))**

Removes an outermost encapsulation from the packet:

- If the packet is an Ethernet packet, removes the Ethernet header, which changes the packet into a bare L3 packet. If the packet has VLAN tags, raises an unsupported packet type error (see *Error Handling*, above).
- Otherwise, if the packet is an NSH packet, removes the NSH header, revealing the inner packet. Open vSwitch supports Ethernet, IPv4, IPv6, and NSH inner packet types. Other types raise unsupported packet type errors.

- Otherwise, if the packet is encapsulated inside a MPLS header, removes the MPLS header and classifies the inner packet as mentioned in the packet type argument of the decap. The *packet_type* field specifies the type of the packet in the format specified in OpenFlow 1.5 chapter *7.2.3.11 Packet Type Match Field*. The inner packet will be incorrectly classified, if the inner packet is different from mentioned in the *packet_type* field.
- Otherwise, raises an unsupported packet type error.

Conformance

This action is an Open vSwitch extension to OpenFlow 1.3 and later, introduced in Open vSwitch 2.8.

The MPLS support for this action is added in Open vSwitch 2.17.

FIELD MODIFICATION ACTIONS

These actions modify packet data and metadata fields.

The set_field and load actions

Syntax:

set_field:*value***[/***mask***]−>***dst*
load:*value***−>***dst*

These actions loads a literal value into a field or part of a field. The **set_field** action takes *value* in the customary syntax for field *dst*, e.g. **00:11:22:33:44:55** for an Ethernet address, and *dst* as the field's name. The optional *mask* allows part of a field to be set.

The **load** action takes *value* as an integer value (in decimal or prefixed by **0x** for hexadecimal) and *dst* as a field or subfield in the syntax described under *Field Specifications* above.

The following all set the Ethernet source address to 00:11:22:33:44:55:

- **set_field:00:11:22:33:44:55−>eth_src**
- **load:0x001122334455−>eth_src**
- **load:0x001122334455−>OXM_OF_ETH_SRC[]**

The following all set the multicast bit in the Ethernet destination address:

- **set_field:01:00:00:00:00:00/01:00:00:00:00:00−>eth_dst**
- **load:1−>eth_dst[40]**

Open vSwitch prohibits a **set_field** or **load** action whose *dst* is not guaranteed to be part of the packet; for example, **set_field** of **nw_dst** is only allowed in a flow that matches on Ethernet type 0x800. In some cases, such as in an action set, Open vSwitch can't statically check that *dst* is part of the packet, and in that case if it is not then Open vSwitch treats the action as a no−op.

Conformance

Open vSwitch 1.1 introduced **NXAST_REG_LOAD** as a extension to OpenFlow 1.0 and used **load** to express it. Later, OpenFlow 1.2 introduced a standard **OFPAT_SET_FIELD** action that was restricted to loading entire fields, so Open vSwitch added the form **set_field** with this restriction. OpenFlow 1.5 extended **OFPAT_SET_FIELD** to the point that it became a superset of **NXAST_REG_LOAD**. Open vSwitch translates either syntax as necessary for the OpenFlow version in use: in OpenFlow 1.0 and 1.1, **NXAST_REG_LOAD**; in OpenFlow 1.2, 1.3, and 1.4, **NXAST_REG_LOAD** for **load** or for loading a subfield, **OFPAT_SET_FIELD** otherwise; and OpenFlow 1.5 and later, **OFPAT_SET_FIELD**.

The move action

Syntax:

move:*src*−>*dst*

Copies the named bits from field or subfield *src* to field or subfield *dst*. *src* and *dst* should fields or subfields in the syntax described under *Field Specifications* above. The two fields or subfields must have the same width.

Examples:

- **move:reg0[0..5]−>reg1[26..31]** copies the six bits numbered 0 through 5 in register 0 into bits 26 through 31 of register 1.
- **move:reg0[0..15]−>vlan_tci** copies the least significant 16 bits of register 0 into the VLAN TCI field.

Conformance

In OpenFlow 1.0 through 1.4, **move** ordinarily uses an Open vSwitch extension to OpenFlow. In OpenFlow 1.5, **move** uses the OpenFlow 1.5 standard **OFPAT_COPY_FIELD** action. The ONF has also made **OFPAT_COPY_FIELD** available as an extension to OpenFlow 1.3. Open vSwitch 2.4 and later understands this extension and uses it if a controller uses it, but for backward compatibility with older versions of Open vSwitch, **ovs−ofctl** does not use it.

The mod_dl_src and mod_dl_dst actions

Syntax:

mod_dl_src:*mac*
mod_dl_dst:*mac*

Sets the Ethernet source or destination address, respectively, to *mac*, which should be expressed in the form **xx:xx:xx:xx:xx:xx**.

For L3−only packets, that is, those that lack an Ethernet header, this action has no effect.

Conformance

OpenFlow 1.0 and 1.1 have specialized actions for these purposes. OpenFlow 1.2 and later do not, so Open vSwitch translates them to appropriate **OFPAT_SET_FIELD** actions for those versions,

The mod_nw_src and mod_nw_dst actions

Syntax:

mod_nw_src:*ip*
mod_nw_dst:*ip*

Sets the IPv4 source or destination address, respectively, to *ip*, which should be expressed in the form **w.x.y.z**.

In OpenFlow 1.1 and later, consistency rules allow these actions only in a flow that matches only packets that contain an IPv4 header (or following an action that adds an IPv4 header, e.g. **pop_mpls:0x0800**). See *Inconsistencies*, above, for more information.

Conformance

OpenFlow 1.0 and 1.1 have specialized actions for these purposes. OpenFlow 1.2 and later do not, so Open vSwitch translates them to appropriate **OFPAT_SET_FIELD** actions for those versions,

The mod_nw_tos and mod_nw_ecn actions

Syntax:

mod_nw_tos:*tos*
mod_nw_ecn:*ecn*

The **mod_nw_tos** action sets the DSCP bits in the IPv4 ToS/DSCP or IPv6 traffic class field to *tos*, which must be a multiple of 4 between 0 and 255. This action does not modify the two least significant bits of the ToS field (the ECN bits).

The **mod_nw_ecn** action sets the ECN bits in the IPv4 ToS or IPv6 traffic class field to *ecn*, which must be a value between 0 and 3, inclusive. This action does not modify the six most significant bits of the field (the DSCP bits).

In OpenFlow 1.1 and later, consistency rules allow these actions only in a flow that matches only packets that contain an IPv4 or IPv6 header (or following an action that adds such a header). See *Inconsistencies*, above, for more information.

Conformance

OpenFlow 1.0 has a **mod_nw_tos** action but not **mod_nw_ecn**. Open vSwitch implements the latter in OpenFlow 1.0 as an extension using **NXAST_REG_LOAD**. OpenFlow 1.1 has specialized actions for these purposes. OpenFlow 1.2 and later do not, so Open vSwitch translates them to appropriate **OFPAT_SET_FIELD** actions for those versions.

The mod_tp_src and mod_tp_dst actions

Syntax:

mod_tp_src:*port*
mod_tp_dst:*port*

Sets the TCP or UDP or SCTP source or destination port, respectively, to *port*. Both IPv4 and IPv6 are supported.

In OpenFlow 1.1 and later, consistency rules allow these actions only in a flow that matches only packets that contain a TCP or UDP or SCTP header. See *Inconsistencies*, above, for more information.

Conformance

OpenFlow 1.0 and 1.1 have specialized actions for these purposes. OpenFlow 1.2 and later do not, so Open vSwitch translates them to appropriate **OFPAT_SET_FIELD** actions for those versions,

The dec_ttl action

Syntax:

dec_ttl
dec_ttl(*id1***[,***id2***[, ...]])**

Decrement TTL of IPv4 packet or hop limit of IPv6 packet. If the TTL or hop limit is initially 0 or 1, no decrement occurs, as packets reaching TTL zero must be rejected. Instead, Open vSwitch sends a **packet−in** message with reason code **OFPR_INVALID_TTL** to each connected controller that has enabled receiving such messages, and stops processing the current set of actions. (However, if the current set of actions was reached through **resubmit**, the remaining actions in outer levels resume processing.)

As an Open vSwitch extension to OpenFlow, this action supports the ability to specify a list of controller IDs. Open vSwitch will only send the message to controllers with the given ID or IDs. Specifying no list is equivalent to specifying a single controller ID of zero.

In OpenFlow 1.1 and later, consistency rules allow these actions only in a flow that matches only packets that contain an IPv4 or IPv6 header. See *Inconsistencies*, above, for more information.

Conformance

All versions of OpenFlow and Open vSwitch support this action.

The set_mpls_label, set_mpls_tc, and set_mpls_ttl actions

Syntax:

set_mpls_label:*label*
set_mpls_tc:*tc*
set_mpls_ttl:*ttl*

The **set_mpls_label** action sets the label of the packet's outer MPLS label stack entry. *label* should be a 20−bit value that is decimal by default; use a **0x** prefix to specify the value in hexadecimal.

The **set_mpls_tc** action sets the traffic class of the packet's outer MPLS label stack entry. *tc* should be in the range 0 to 7, inclusive.

The **set_mpls_ttl** action sets the TTL of the packet's outer MPLS label stack entry. *ttl* should be in the range 0 to 255 inclusive. In OpenFlow 1.1 and later, consistency rules allow these actions only in a flow that matches only packets that contain an MPLS label (or following an action that adds an MPLS label, e.g. **push_mpls:0x8847**). See *Inconsistencies*, above, for more information.

Conformance

OpenFlow 1.0 does not support MPLS, but Open vSwitch implements these actions as extensions. OpenFlow 1.1 has specialized actions for these purposes. OpenFlow 1.2 and later do not, so Open vSwitch translates them to appropriate **OFPAT_SET_FIELD** actions for those versions,

The dec_mpls_ttl and dec_nsh_ttl actions

Syntax:

dec_mpls_ttl
dec_nsh_ttl

These actions decrement the TTL of the packet's outer MPLS label stack entry or its NSH header, respectively. If the TTL is initially 0 or 1, no decrement occurs. Instead, Open vSwitch sends a **packet−in** message with reason code **BOFPR_INVALID_TTL** to OpenFlow controllers with ID 0, if it has enabled receiving them. Processing the current set of actions then stops. (However, if the current set of actions was reached through **resubmit**, remaining actions in outer levels resume processing.)

In OpenFlow 1.1 and later, consistency rules allow this actions only in a flow that matches only packets that contain an MPLS label or an NSH header, respectively. See *Inconsistencies*, above, for more information.

Conformance

Open vSwitch 1.11 introduced support for MPLS. OpenFlow 1.1 and later support **dec_mpls_ttl**. Open vSwitch implements **dec_mpls_ttl** as an extension to OpenFlow 1.0.

Open vSwitch 2.8 introduced support for NSH, although the NSH draft changed after release so that only Open vSwitch 2.9 and later conform to the final protocol specification. The **dec_nsh_ttl** action and NSH support in general is an Open vSwitch extension not supported by any version of OpenFlow.

The check_pkt_larger action

Syntax:

check_pkt_larger(*pkt_len***)−>***dst*

Checks if the packet is larger than the specified length in *pkt_len*. If so, stores 1 in *dst*, which should be a 1−bit field; if not, stores 0.

The packet length to check against the argument *pkt_len* includes the L2 header and L2 payload of the packet, but not the VLAN tag (if present).

Examples:

- **check_pkt_larger(1500)−>reg0[0]**
- **check_pkt_larger(8000)−>reg9[10]**

This action was added in Open vSwitch 2.12.

The delete_field action

Syntax:

delete_field:*field*

The **delete_field** action deletes a *field* in the syntax described under *Field Specifications* above. Currently, only the **tun_metadta** fields are supported.

This action was added in Open vSwitch 2.14.

METADATA ACTIONS

The set_tunnel action

Syntax:

set_tunnel:*id*
set_tunnel64:*id*

Many kinds of tunnels support a tunnel ID, e.g. VXLAN and Geneve have a 24−bit VNI, and GRE has an optional 32−bit key. This action sets the value used for tunnel ID in such tunneled packets, although whether it is used for a particular tunnel depends on the tunnel's configuration. See the tunnel ID documentation in **ovs−fields(7)** for more information.

Conformance

These actions are OpenFlow extensions. **set_tunnel** was introduced in Open vSwitch 1.0. **set_tunnel64**, which is needed if *id* is wider than 32 bits, was added in Open vSwitch 1.1. Both actions always set the entire tunnel ID field. Open vSwitch supports these actions in all versions of OpenFlow, but in OpenFlow 1.2 and later it translates them to an appropriate standardized **OFPAT_SET_FIELD** action.

The set_queue and pop_queue actions

Syntax:

set_queue:*queue*
pop_queue

The **set_queue** action sets the queue ID to be used for subsequent output actions to *queue*, which must be a 32−bit integer. The range of meaningful values of *queue*, and their meanings, varies greatly from one OpenFlow implementation to another. Even within a single implementation, there is no guarantee that all OpenFlow ports have the same queues configured or that all OpenFlow ports in an implementation can be configured the same way queue−wise. For more information, see the documentation for the output queue field in **ovs−fields(7)**.

The **pop_queue** restores the output queue to the default that was set when the packet entered the switch (generally 0).

Four billion queues ought to be enough for anyone: *https://mailman.stanford.edu/pipermail/openflow−spec/2009−August/000394.html*

Conformance

OpenFlow 1.1 introduced the **set_queue** action. Open vSwitch also supports it as an extension in OpenFlow 1.0.

The **pop_queue** action is an Open vSwitch extension.

FIREWALLING ACTIONS

Open vSwitch is often used to implement a firewall. The preferred way to implement a firewall is **connection tracking,** that is, to keep track of the connection state of individual TCP sessions. The **ct** action described in this section, added in Open vSwitch 2.5, implements connection tracking. For new deployments, it is the recommended way to implement firewalling with Open vSwitch.

Before **ct** was added, Open vSwitch did not have built−in support for connection tracking. Instead, Open vSwitch supported the **learn** action, which allows a received packet to add a flow to an OpenFlow flow table. This could be used to implement a primitive form of connection tracking: packets passing through the firewall in one direction could create flows that allowed response packets back through the firewall in the other direction. The additional **fin_timeout** action allowed the learned flows to expire quickly after TCP session termination.

The ct action

Syntax:

ct([*argument***]...)**
ct(commit[,*argument***]...)**

The action has two modes of operation, distinguished by whether **commit** is present. The following arguments may be present in either mode:

zone=*value*

A zone is a 16−bit id that isolates connections into separate domains, allowing overlapping network addresses in different zones. If a zone is not provided, then the default is 0. The *value* may be specified either as a 16−bit integer literal or a field or subfield in the syntax described under *Field Specifications* above.

Without **commit**, this action sends the packet through the connection tracker. The connection tracker keeps track of the state of TCP connections for packets passed through it. For each packet through a connection, it checks that it satisfies TCP invariants and signals the connection state to later actions using the **ct_state** metadata field, which is documented in **ovs−fields(7)**.

In this form, **ct** forks the OpenFlow pipeline:

- In one fork, **ct** passes the packet to the connection tracker. Afterward, it reinjects the packet into the OpenFlow pipeline with the connection tracking fields initialized. The **ct_state** field is initialized with connection state and **ct_zone** to the connection tracking zone specified on the **zone** argument. If the connection is one that is already tracked, **ct_mark** and **ct_label** to its existing mark and label, respectively; otherwise they are zeroed. In addition, **ct_nw_proto**, **ct_nw_src**, **ct_nw_dst**, **ct_ipv6_src**, **ct_ipv6_dst**, **ct_tp_src**, and **ct_tp_dst** are initialized appropriately for the original direction connection. See the **resubmit** action for a way to search the flow table with the connection tracking original direction fields swapped with the packet 5−tuple fields. See **ovs−fields(7)** for details on the connection tracking fields.
- In the other fork, the original instance of the packet continues independent processing following the **ct** action. The **ct_state** field and other connection tracking metadata are cleared.

Without **commit**, the **ct** action accepts the following arguments:

table=*table*

Sets the OpenFlow table where the packet is reinjected. The *table* must be a number between 0 and 254 inclusive, or a table's name. If *table* is not specified, then the packet is not reinjected.

nat

nat(*type***=***addrs***[:***ports***][,***flag***]...)**

Specify address and port translation for the connection being tracked. The *type* must be **src**, for source address/port translation (SNAT), or **dst**, for destination address/port translation (DNAT). Setting up address translation for a new connection takes effect only if the connection is later committed with **ct(commit ...)**.

The **src** and **dst** options take the following arguments:

addrs The IP address **addr** or range **addr1−addr2** from which the translated address should be selected. If only one address is given, then that address will always be selected, otherwise the address selection can be informed by the optional persistent flag as described below. Either IPv4 or IPv6 addresses can be provided, but both addresses must be of the same type, and the datapath behavior is undefined in case of providing IPv4 address range for an IPv6 packet, or IPv6 address range for an IPv4 packet. IPv6 addresses must be bracketed with **[** and **]** if a port range is also given.

ports The L4 **port** or range **port1−port2** from which the translated port should be selected. When a port range is specified, fallback to ephemeral ports does not happen, else, it will. The port number selection can be informed by the optional **random** and **hash** flags described below. The userspace datapath only supports the **hash** behavior.

The optional *flags* are:

random

The selection of the port from the given range should be done using a fresh random number. This flag is mutually exclusive with **hash**.

hash The selection of the port from the given range should be done using a datapath specific hash of the packet's IP addresses and the other, non−mapped port number. This flag is mutually exclusive with **random**.

persistent

The selection of the IP address from the given range should be done so that the same mapping can be provided after the system restarts.

If **alg** is specified for the committing **ct** action that also includes **nat** with a **src** or **dst** attribute, then the datapath tries to set up the helper to be NAT−aware. This functionality is datapath specific and may not be supported by all datapaths.

A **bare nat** argument with no options will only translate the packet being processed in the way the connection has been set up with an earlier, committed **ct** action. A **nat** action with **src** or **dst**, when applied to a packet belonging to an established (rather than new) connection, will behave the same as a bare **nat**.

For SNAT, there is a special case when the **src** IP address is configured as all 0's, i.e., **nat(src=0.0.0.0)**. In this case, when a source port collision is detected during the commit, the source port will be translated to an ephemeral port. If there is no collision, no SNAT is performed.

Open vSwitch 2.6 introduced **nat**. Linux 4.6 was the earliest upstream kernel that implemented **ct** support for **nat**.

With **commit**, the connection tracker commits the connection to the connection tracking module. The **commit** flag should only be used from the pipeline within the first fork of **ct** without **commit**. Information about the connection is stored beyond the lifetime of the packet in the pipeline. Some **ct_state** flags are only available for committed connections.

The following options are available only with **commit**:

force A committed connection always has the directionality of the packet that caused the connection to be committed in the first place. This is the **original direction** of the connection, and the opposite direction is the **reply direction**. If a connection is already committed, but it is in the wrong direction, **force** effectively terminates the existing connection and starts a new one in the current direction. This flag has no effect if the original direction of the connection is already the same as that of the current packet.

exec(*action***...)**
Perform each *action* within the context of connection tracking. Only actions which modify the **ct_mark** or **ct_label** fields are accepted within **exec** action, and these fields may only be modified with this option. For example:

set_field:*value***[/***mask***]->ct_mark**
Store a 32−bit metadata value with the connection. Subsequent lookups for packets in this connection will populate **ct_mark** when the packet is sent to the connection tracker with the table specified.

set_field:*value***[/***mask***]->ct_label**
Store a 128−bit metadata value with the connection. Subsequent lookups for packets in this connection will populate **ct_label** when the packet is sent to the connection tracker with the table specified.

alg=*alg*
Specify application layer gateway *alg* to track specific connection types. If subsequent related connections are sent through the **ct** action, then the **rel** flag in the **ct_state** field will be set. Supported types include:

ftp Look for negotiation of FTP data connections. Specify this option for FTP control connections to detect related data connections and populate the **rel** flag for the data connections.

tftp Look for negotiation of TFTP data connections. Specify this option for TFTP control connections to detect related data connections and populate the **rel** flag for the data connections.

Related connections inherit **ct_mark** from that stored with the original connection (i.e. the connection created by **ct(alg=...)**.

With the Linux datapath, global sysctl options affect **ct** behavior. In particular, if **net.netfilter.nf_conntrack_helper** is enabled, which it is by default until Linux 4.7, then application layer gateway helpers may be executed even if *alg* is not specified. For security reasons, the netfilter team recommends users disable this option. For further details, please see *http://www.netfilter.org/news.html#2012−04−03* .

The **ct** action may be used as a primitive to construct stateful firewalls by selectively committing some traffic, then matching **ct_state** to allow established connections while denying new connections. The following flows provide an example of how to implement a simple firewall that allows new connections from port 1 to port 2, and only allows established connections to send traffic from port 2 to port 1:

```
table=0,priority=1,action=drop
```

```
table=0,priority=10,arp,action=normal
table=0,priority=100,ip,ct_state=-trk,action=ct(table=1)
table=1,in_port=1,ip,ct_state=+trk+new,action=ct(commit),2
table=1,in_port=1,ip,ct_state=+trk+est,action=2
table=1,in_port=2,ip,ct_state=+trk+new,action=drop
table=1,in_port=2,ip,ct_state=+trk+est,action=1
```

If **ct** is executed on IPv4 (or IPv6) fragments, then the message is implicitly reassembled before sending to the connection tracker and refragmented upon output, to the original maximum received fragment size. Reassembly occurs within the context of the zone, meaning that IP fragments in different zones are not assembled together. Pipeline processing for the initial fragments is halted. When the final fragment is received, the message is assembled and pipeline processing continues for that flow. Packet ordering is not guaranteed by IP protocols, so it is not possible to determine which IP fragment will cause message reassembly (and therefore continue pipeline processing). As such, it is strongly recommended that multiple flows should not execute **ct** to reassemble fragments from the same IP message.

Conformance

The **ct** action was introduced in Open vSwitch 2.5. Some of its features were introduced later, noted individually above.

The ct_clear action

Syntax:

ct_clear

Clears connection tracking state from the flow, zeroing **ct_state**, **ct_zone**, **ct_mark**, and **ct_label**.

This action was introduced in Open vSwitch 2.7.

The learn action

Syntax:

learn(*argument***...)**

The **learn** action adds or modifies a flow in an OpenFlow table, similar to **ovs−ofctl −−strict mod−flows**. The arguments specify the match fields, actions, and other properties of the flow to be added or modified.

Match fields for the new flow are specified as follows. At least one match field should ordinarily be specified:

*field***=***value*

Specifies that *field*, in the new flow, must match the literal *value*, e.g. **dl_type=0x800**. Shorthand match syntax, such as **ip** in place of **dl_type=0x800**, is not supported.

*field***=***src*

Specifies that *field* in the new flow must match *src* taken from the packet currently being processed. For example, **udp_dst=udp_src**, applied to a UDP packet with source port 53, creates a flow which matches **udp_dst=53**. *field* and *src* must have the same width.

field Shorthand for the previous form when *field* and *src* are the same. For example, **udp_dst**, applied to a UDP packet with destination port 53, creates a flow which matches **udp_dst=53**.

The *field* and *src* arguments above should be fields or subfields in the syntax described under *Field Specifications* above.

Match field specifications must honor prerequisites for both the flow with the **learn** and the new flow that it creates. Consider the following complete flow, in the syntax accepted by **ovs−ofctl**. If the flow's match on **udp** were omitted, then the flow would not satisfy the prerequisites for the **learn** action's use of **udp_src**.

If **dl_type=0x800** or **nw_proto** were omitted from **learn**, then the new flow would not satisfy the prerequisite for its match on **udp_dst**. For more information on prerequisites, please refer to **ovs−fields(7)**:

```
udp, actions=learn(dl_type=0x800, nw_proto=17, udp_dst=udp_src)
```

Actions for the new flow are specified as follows. At least one action should ordinarily be specified:

load:*value*−>*dst*
Adds a **load** action to the new flow that loads the literal *value* into *dst*. The syntax is the same as the **load** action explained in the *Field Modification Actions* section.

load:*src*−>*dst*
Adds a **load** action to the new flow that loads *src*, a field or subfield from the packet being processed, into *dst*.

output:*field*
Adds an **output** action to the new flow's actions that outputs to the OpenFlow port taken from *field*, which must be a field as described above.

fin_idle_timeout=*seconds* **/ fin_hard_timeout=***seconds*
Adds a **fin_timeout** action with the specified arguments to the new flow. This feature was added in Open vSwitch 1.6.

The following additional arguments are optional:

idle_timeout=*seconds*

hard_timeout=*seconds*

priority=*value*

cookie=*value*

send_flow_rem
These arguments have the same meaning as in the usual flow syntax documented in **ovs−ofctl(8)**.

table=*table*
The table in which the new flow should be inserted. Specify a decimal number between 0 and 254 inclusive or the name of a table. The default, if table is unspecified, is table 1 (not 0).

delete_learned
When this flag is specified, deleting the flow that contains the **learn** action will also delete the flows created by **learn**. Specifically, when the last **learn** action with this flag and particular **table** and **cookie** values is removed, the switch deletes all of the flows in the specified table with the specified cookie.

This flag was added in Open vSwitch 2.4.

limit=*number*
If the number of flows in the new flow's table with the same cookie exceeds *number*, the action will not add a new flow. By default, or with **limit=0**, there is no limit.

This flag was added in Open vSwitch 2.8.

result_dst=*field*[*bit*]
If learn fails (because the number of flows exceeds **limit**), the action sets *field*[*bit*] to 0, otherwise it will be set to 1. *field*[*bit*] must be a single bit.

This flag was added in Open vSwitch 2.8.

By itself, the **learn** action can only put two kinds of actions into the flows that it creates: **load** and **output** actions. If **learn** is used in isolation, these are severe limits.

However, **learn** is not meant to be used in isolation. It is a primitive meant to be used together with other Open vSwitch features to accomplish a task. Its existing features are enough to accomplish most tasks.

Here is an outline of a typical pipeline structure that allows for versatile behavior using **learn**:

- Flows in table **A** contain a **learn** action, that populates flows in table **L**, that use a **load** action to populate register **R** with information about what was learned.
- Flows in table **B** contain two sequential resubmit actions: one to table **L** and another one to table **B + 1**.
- Flows in table **B + 1** match on register **R** and act differently depending on what the flows in table **L** loaded into it.

This approach can be used to implement many **learn**−based features. For example:

- Resubmit to a table selected based on learned information, e.g. see *https://mail.openvswitch.org/pipermail/ovs−discuss/2016−June/021694.html* .
- MAC learning in the middle of a pipeline, as described in the **Open vSwitch Advanced Features Tutorial** in the OVS documentation.
- TCP state based firewalling, by learning outgoing connections based on SYN packets and matching them up with incoming packets. (This is usually better implemented using the **ct** action.)
- At least some of the features described in T. A. Hoff, **Extending Open vSwitch to Facilitate Creation of Stateful SDN Applications**.

Conformance

The **learn** action is an Open vSwitch extension to OpenFlow added in Open vSwitch 1.3. Some features of **learn** were added in later versions, as noted individually above.

The fin_timeout action

Syntax:

fin_timeout(*key***=***value***...)**

This action changes the idle timeout or hard timeout, or both, of the OpenFlow flow that contains it, when the flow matches a TCP packet with the FIN or RST flag. When such a packet is observed, the action reduces the rule's timeouts to those specified on the action. If the rule's existing timeout is already shorter than the one that the action specifies, then that timeout is unaffected.

The timeouts are specified as key−value pairs:

idle_timeout=*seconds*

Causes the flow to expire after the given number of seconds of inactivity.

hard_timeout=*seconds*

Causes the flow to expire after the given number of *seconds*, regardless of activity. (*seconds* specifies time since the flow's creation, not since the receipt of the FIN or RST.)

This action is normally added to a learned flow by the **learn** action. It is unlikely to be useful otherwise.

Conformance

This Open vSwitch extension action was added in Open vSwitch 1.6.

PROGRAMMING AND CONTROL FLOW ACTIONS

The resubmit action

Syntax:

resubmit:*port*
resubmit([*port*]**,**[*table*][,ct])``

Searches an OpenFlow flow table for a matching flow and executes the actions found, if any, before continuing to the following action in the current flow entry. Arguments can customize the search:

- If *port* is given as an OpenFlow port number or name, then it specifies a value to use for the input port metadata field as part of the search, in place of the input port currently in the flow. Specifying **in_port** as **port** is equivalent to omitting it.
- If *table* is given as an integer between 0 and 254 or a table name, it specifies the OpenFlow table to search. If it is not specified, the table from the current flow is used.
- If **ct** is specified, then the search is done with packet 5−tuple fields swapped with the corresponding conntrack original direction tuple fields. See the documentation for **ct** above, for more information about connection tracking, or **ovs−fields(7)** for details about the connection tracking fields.

  This flag requires a valid connection tracking state as a match prerequisite in the flow where this action is placed. Examples of valid connection tracking state matches include **ct_state=+new**, **ct_state=+est**, **ct_state=+rel**, and **ct_state=+trk−inv**.

The changes, if any, to the input port and connection tracking fields are just for searching the flow table. The changes are not visible to actions or to later flow table lookups.

The most common use of **resubmit** is to visit another flow table without *port* or **ct**, like this: **resubmit(,***table***)**.

Recursive **resubmit** actions are permitted.

Conformance

The **resubmit** action is an Open vSwitch extension. However, the **goto_table** instruction in OpenFlow 1.1 and later can be viewed as a kind of restricted **resubmit**.

Open vSwitch 1.3 added **table**. Open vSwitch 2.7 added **ct**.

Open vSwitch imposes a limit on **resubmit** recursion that varies among version:

- Open vSwitch 1.0.1 and earlier did not support recursion.
- Open vSwitch 1.0.2 and 1.0.3 limited recursion to 8 levels.
- Open vSwitch 1.1 and 1.2 limited recursion to 16 levels.
- Open vSwitch 1.2 through 1.8 limited recursion to 32 levels.
- Open vSwitch 1.9 through 2.0 limited recursion to 64 levels.
- Open vSwitch 2.1 through 2.5 limited recursion to 64 levels and impose a total limit of 4,096 resubmits per flow translation (earlier versions did not impose any total limit).
- Open vSwitch 2.6 and later imposes the same limits as 2.5, with one exception: resubmit from table **x** to any table **y > x** does not count against the recursion depth limit.

The clone action

Syntax:

clone(*action***...)**

Executes each nested *action*, saving much of the packet and pipeline state beforehand and then restoring it

afterward. The state that is saved and restored includes all flow data and metadata (including, for example, **in_port** and **ct_state**), the stack accessed by **push** and **pop** actions, and the OpenFlow action set.

This action was added in Open vSwitch 2.7.

The push and pop actions

Syntax:

push:*src*
pop:*dst*

The **push** action pushes *src* on a general−purpose stack. The **pop** action pops an entry off the stack into *dst*. *src* and *dst* should be fields or subfields in the syntax described under *Field Specifications* above.

Controllers can use the stack for saving and restoring data or metadata around **resubmit** actions, for swapping or rearranging data and metadata, or for other purposes. Any data or metadata field, or part of one, may be pushed, and any modifiable field or subfield may be popped.

The number of bits pushed in a stack entry do not have to match the number of bits later popped from that entry. If more bits are popped from an entry than were pushed, then the entry is conceptually left−padded with 0−bits as needed. If fewer bits are popped than pushed, then bits are conceptually trimmed from the left side of the entry.

The stack's size is limited. The limit is intended to be high enough that **normal** use will not pose problems. Stack overflow or underflow is an error that stops action execution (see **Stack too deep** under *Error Handling*, above).

Examples:

- **push:reg2[0..5]** or **push:NXM_NX_REG2[0..5]** pushes on the stack the 6 bits in register 2 bits 0 through 5.
- **pop:reg2[0..5]** or **pop:NXM_NX_REG2[0..5]** pops the value from top of the stack and copy bits 0 through 5 of that value into bits 0 through 5 of register 2.

Conformance

Open vSwitch 1.2 introduced **push** and **pop** as OpenFlow extension actions.

The exit action

Syntax:

exit

This action causes Open vSwitch to immediately halt execution of further actions. Actions which have already been executed are unaffected. Any further actions, including those which may be in other tables, or different levels of the **resubmit** call stack, are ignored. However, an **exit** action within a group bucket terminates only execution of that bucket, not other buckets or the overall pipeline. Actions in the action set are still executed (specify **clear_actions** before **exit** to discard them).

The multipath action

Syntax:

multipath(*fields*,*basis*,*algorithm*,*n_links*,*arg*,*dst***)**

Hashes *fields* using *basis* as a universal hash parameter, then the applies multipath link selection *algorithm* (with parameter *arg*) to choose one of *n_links* output links numbered 0 through *n_links* minus 1, and stores the link into *dst*, which must be a field or subfield in the syntax described under *Field Specifications* above.

The **bundle** or **bundle_load** actions are usually easier to use than **multipath**.

fields must be one of the following:

eth_src Hashes Ethernet source address only.

symmetric_l4
Hashes Ethernet source, destination, and type, VLAN ID, IPv4/IPv6 source, destination, and protocol, and TCP or SCTP (but not UDP) ports. The hash is computed so that pairs of corresponding flows in each direction hash to the same value, in environments where L2 paths are the same in each direction. UDP ports are not included in the hash to support protocols such as VXLAN that use asymmetric ports in each direction.

symmetric_l3l4
Hashes IPv4/IPv6 source, destination, and protocol, and TCP or SCTP (but not UDP) ports. Like **symmetric_l4**, this is a symmetric hash, but by excluding L2 headers it is more effective in environments with asymmetric L2 paths (e.g. paths involving VRRP IP addresses on a router). Not an effective hash function for protocols other than IPv4 and IPv6, which hash to a constant zero.

symmetric_l3l4+udp
Like **symmetric_l3l4+udp**, but UDP ports are included in the hash. This is a more effective hash when asymmetric UDP protocols such as VXLAN are not a consideration.

symmetric_l3
Hashes network source address and network destination address.

nw_src Hashes network source address only.

nw_dst Hashes network destination address only.

The *algorithm* used to compute the final result **link** must be one of the following:

modulo_n
Computes **link = hash(flow) % n_links**.

This algorithm redistributes all traffic when **n_links** changes. It has **O(1)** performance.

Use 65535 for **max_link** to get a raw hash value.

This algorithm is specified by RFC 2992.

hash_threshold
Computes **link = hash(flow) / (MAX_HASH / n_links)**.

Redistributes between one−quarter and one−half of traffic when **n_links** changes. It has **O(1)** performance.

This algorithm is specified by RFC 2992.

hrw (Highest Random Weight)
Computes the following:

```
for i in [0, n_links]:
    weights[i] = hash(flow, i)
link = { i such that weights[i] >= weights[j] for all j != i }
```

Redistributes **1 / n_links** of traffic when **n_links** changes. It has **O(n_links)** performance. If **n_links** is greater than a threshold (currently 64, but subject to change), Open vSwitch will substitute another algorithm automatically.

This algorithm is specified by RFC 2992.

iter_hash (Iterative Hash)
Computes the following:

```
i = 0
repeat:
    i = i + 1
    link = hash(flow, i) % arg
while link > max_link
```

Redistributes **1 / n_links** of traffic when **n_links** changes. **O(1)** performance when **arg / max_link** is bounded by a constant.

Redistributes all traffic when **arg** changes.

arg must be greater than **max_link** and for best performance should be no more than approximately **max_link * 2**. If *arg* is outside the acceptable range, Open vSwitch will automatically substitute the least power of 2 greater than **max_link**.

This algorithm is specific to Open vSwitch.

Only the **iter_hash** algorithm uses *arg*.

It is an error if **max_link** is greater than or equal to **2**n_bits**.

Conformance
This is an OpenFlow extension added in Open vSwitch 1.1.

OTHER ACTIONS

The conjunction action

Syntax:
conjunction(*id*, *k*/*n***)**

This action allows for sophisticated **conjunctive match** flows. Refer to **Conjunctive Match Fields** in **ovs−fields(7)** for details.

A flow that has one or more **conjunction** actions may not have any other actions except for **note** actions.

Conformance
Open vSwitch 2.4 introduced the **conjunction** action and **conj_id** field. They are Open vSwitch extensions to OpenFlow.

The note action

Syntax:
note:[*hh***]...**

This action does nothing at all. OpenFlow controllers may use it to annotate flows with more data than can fit in a flow cookie.

The action may include any number of bytes represented as hex digits *hh*. Periods may separate pairs of hex digits, for readability. The **note** action's format doesn't include an exact length for its payload, so the provided bytes will be padded on the right by enough bytes with value 0 to make the total number 6 more than a multiple of 8.

Conformance

This action is an extension to OpenFlow introduced in Open vSwitch 1.1.

The sample action

Syntax:

sample(*argument***...)**

Samples packets and sends one sample for every sampled packet.

The following *argument* forms are accepted:

probability=*packets*

The number of sampled packets out of 65535. Must be greater or equal to 1.

collector_set_id=*id*

The unsigned 32−bit integer identifier of the set of sample collectors to send sampled packets to. Defaults to 0.

obs_domain_id=*id*

When sending samples to IPFIX collectors, the unsigned 32−bit integer Observation Domain ID sent in every IPFIX flow record. Defaults to 0.

obs_point_id=*id*

When sending samples to IPFIX collectors, the unsigned 32−bit integer Observation Point ID sent in every IPFIX flow record. Defaults to 0.

sampling_port=*port*

Sample packets on *port*, which should be the ingress or egress port. This option, which was added in Open vSwitch 2.6, allows the IPFIX implementation to export egress tunnel information.

ingress

egress Specifies explicitly that the packet is being sampled on ingress to or egress from the switch. IPFIX reports sent by Open vSwitch before version 2.6 did not include a direction. From 2.6 until 2.7, IPFIX reports inferred a direction from **sampling_port**: if it was the packet's output port, then the direction was reported as egress, otherwise as ingress. Open vSwitch 2.7 introduced these options, which allow the inferred direction to be overridden. This is particularly useful when the ingress (or egress) port is not a tunnel.

Refer to **ovs−vswitchd.conf.db(5)** for more details on configuring sample collector sets.

Conformance

This action is an OpenFlow extension added in Open vSwitch 2.4.

INSTRUCTIONS

Every version of OpenFlow includes actions. OpenFlow 1.1 introduced the higher−level, related concept of **instructions**. In OpenFlow 1.1 and later, actions within a flow are always encapsulated within an instruction. Each flow has at most one instruction of each kind, which are executed in the following fixed order defined in the OpenFlow specification:

1. **Meter**
2. **Apply−Actions**
3. **Clear−Actions**
4. **Write−Actions**
5. **Write−Metadata**

6. **Stat−Trigger** (not supported by Open vSwitch)
7. **Goto−Table**

The most important instruction is **Apply−Actions**. This instruction encapsulates any number of actions, which the instruction executes. Open vSwitch does not explicitly represent **Apply−Actions**. Instead, any action by itself is implicitly part of an **Apply−Actions** instructions.

Open vSwitch syntax requires other instructions, if present, to be in the order listed above. Otherwise it will flag an error.

The meter action and instruction

Syntax:

meter:*meter_id*

Apply meter *meter_id*. If a meter band rate is exceeded, the packet may be dropped, or modified, depending on the meter band type.

Conformance

OpenFlow 1.3 introduced the **meter** instruction. OpenFlow 1.5 changes **meter** from an instruction to an action.

OpenFlow 1.5 allows implementations to restrict **meter** to be the first action in an action list and to exclude **meter** from action sets, for better compatibility with OpenFlow 1.3 and 1.4. Open vSwitch restricts the **meter** action both ways.

Open vSwitch 2.0 introduced OpenFlow protocol support for meters, but it did not include a datapath implementation. Open vSwitch 2.7 added meter support to the userspace datapath. Open vSwitch 2.10 added meter support to the kernel datapath. Open vSwitch 2.12 added support for meter as an action in OpenFlow 1.5.

The clear_actions instruction

Syntax:

clear_actions

Clears the action set. See *Action Sets*, above, for more information.

Conformance

OpenFlow 1.1 introduced **clear_actions**. Open vSwitch 2.1 added support for **clear_actions**.

The write_actions instruction

Syntax:

write_actions(*action***...)**

Adds each *action* to the action set. The action set is carried between flow tables and then executed at the end of the pipeline. Only certain actions may be written to the action set. See *Action Sets*, above, for more information.

Conformance

OpenFlow 1.1 introduced **write_actions**. Open vSwitch 2.1 added support for **write_actions**.

The write_metadata instruction

Syntax:

write_metadata:*value***[/***mask***]**

Updates the flow's **metadata** field. If *mask* is omitted, **metadata** is set exactly to *value*; if *mask* is specified, then a 1−bit in *mask* indicates that the corresponding bit in **metadata** will be replaced with the corresponding bit from *value*. Both *value* and *mask* are 64−bit values that are decimal by default; use a **0x** prefix to specify them in hexadecimal.

The **metadata** field can also be matched in the flow table and updated with actions such as **set_field** and **move**.

Conformance

OpenFlow 1.1 introduced **write_metadata**. Open vSwitch 2.1 added support for **write_metadata**.

The goto_table instruction

Syntax:

goto_table:*table*

Jumps to *table* as the next table in the process pipeline. The table may be a number between 0 and 254 or a table name.

It is an error if *table* is less than or equal to the table of the flow that contains it; that is, **goto_table** must move forward in the OpenFlow pipeline. Since **goto_table** must be the last instruction in a flow, it never leads to recursion. The **resubmit** extension action is more flexible.

Conformance

OpenFlow 1.1 introduced **goto_table**. Open vSwitch 2.1 added support for **goto_table**.

AUTHOR

The Open vSwitch Development Community

COPYRIGHT