Open vSwitch datapath developer documentation

The Open vSwitch kernel module allows flexible userspace control over flow-level packet processing on selected network devices. It can be used to implement a plain Ethernet switch, network device bonding, VLAN processing, network access control, flow-based network control, and so on.

The kernel module implements multiple "datapaths" (analogous to bridges), each of which can have multiple "vports" (analogous to ports within a bridge). Each datapath also has associated with it a "flow table" that userspace populates with "flows" that map from keys based on packet headers and metadata to sets of actions. The most common action forwards the packet to another vport; other actions are also implemented.

When a packet arrives on a vport, the kernel module processes it by extracting its flow key and looking it up in the flow table. If there is a matching flow, it executes the associated actions. If there is no match, it queues the packet to userspace for processing (as part of its processing, userspace will likely set up a flow to handle further packets of the same type entirely in-kernel).

Flow key compatibility

Network protocols evolve over time. New protocols become important and existing protocols lose their prominence. For the Open vSwitch kernel module to remain relevant, it must be possible for newer versions to parse additional protocols as part of the flow key. It might even be desirable, someday, to drop support for parsing protocols that have become obsolete. Therefore, the Netlink interface to Open vSwitch is designed to allow carefully written userspace applications to work with any version of the flow key, past or future.

To support this forward and backward compatibility, whenever the kernel module passes a packet to userspace, it also passes along the flow key that it parsed from the packet. Userspace then extracts its own notion of a flow key from the packet and compares it against the kernel-provided version:

- If userspace's notion of the flow key for the packet matches the kernel's, then nothing special is necessary.
- If the kernel's flow key includes more fields than the userspace version of the flow key, for example if the kernel decoded IPv6 headers but userspace stopped at the Ethernet type (because it does not understand IPv6), then again nothing special is necessary. Userspace can still set up a flow in the usual way, as long as it uses the kernel-provided flow key to do it.
- If the userspace flow key includes more fields than the kernel's, for example if userspace decoded an IPv6 header but the kernel stopped at the Ethernet type, then userspace can forward the packet manually, without setting up a flow in the kernel. This case is bad for performance because every packet that the kernel considers part of the flow must go to userspace, but the forwarding behavior is correct. (If userspace can determine that the values of the extra fields would not affect forwarding behavior, then it could set up a flow anyway.)

How flow keys evolve over time is important to making this work, so the following sections go into detail.

Flow key format

A flow key is passed over a Netlink socket as a sequence of Netlink attributes. Some attributes represent packet metadata, defined as any information about a packet that cannot be extracted from the packet itself, e.g. the vport on which the packet was received. Most attributes, however, are extracted from headers within the packet, e.g. source and destination addresses from Ethernet, IP, or TCP headers.

The <linux/openvswitch.h> header file defines the exact format of the flow key attributes. For informal explanatory purposes here, we write them as comma-separated strings, with parentheses indicating arguments and nesting. For example, the following could represent a flow key corresponding to a TCP packet that arrived on vport 1:

```
in_port(1), eth(src=e0:91:f5:21:d0:b2, dst=00:02:e3:0f:80:a4),
eth_type(0x0800), ipv4(src=172.16.0.20, dst=172.18.0.52, proto=17, tos=0,
frag=no), tcp(src=49163, dst=80)
```

Often we ellipsize arguments not important to the discussion, e.g.:

```
in_port(1), eth(...), eth_type(0x0800), ipv4(...), tcp(...)
```

Wildcarded flow key format

A wildcarded flow is described with two sequences of Netlink attributes passed over the Netlink socket. A flow key, exactly as described above, and an optional corresponding flow mask.

A wildcarded flow can represent a group of exact match flows. Each '1' bit in the mask specifies a exact match with the corresponding bit in the flow key. A '0' bit specifies a don't care bit, which will match either a '1' or '0' bit of a incoming packet. Using wildcarded flow can improve the flow set up rate by reduce the number of new flows need to be processed by the user space program.

Support for the mask Netlink attribute is optional for both the kernel and user space program. The kernel can ignore the mask attribute, installing an exact match flow, or reduce the number of don't care bits in the kernel to less than what was specified by the user space program. In this case, variations in bits that the kernel does not implement will simply result in additional flow setups. The kernel module will also work with user space programs that neither support nor supply flow mask attributes.

Since the kernel may ignore or modify wildcard bits, it can be difficult for the userspace program to know exactly what matches are installed. There are two possible approaches: reactively install flows as they miss the kernel flow table (and therefore not attempt to determine wildcard changes at all) or use the kernel's response messages to determine the installed wildcards.

When interacting with userspace, the kernel should maintain the match portion of the key exactly as originally installed. This will provides a handle to identify the flow for all future operations. However, when reporting the mask of an installed flow, the mask should include any restrictions imposed by the kernel.

The behavior when using overlapping wildcarded flows is undefined. It is the responsibility of the user space program to ensure that any incoming packet can match at most one flow, wildcarded or not. The current implementation performs best-effort detection of overlapping wildcarded flows and may reject some but not all of them. However, this behavior may change in future versions.

Unique flow identifiers

An alternative to using the original match portion of a key as the handle for flow identification is a unique flow identifier, or "UFID". UFIDs are optional for both the kernel and user space program.

User space programs that support UFID are expected to provide it during flow setup in addition to the flow, then refer to the flow using the UFID for all future operations. The kernel is not required to index flows by the original flow key if a UFID is specified.

Basic rule for evolving flow keys

Some care is needed to really maintain forward and backward compatibility for applications that follow the rules listed under "Flow key compatibility" above.

The basic rule is obvious:

```
==================================================================
New network protocol support must only supplement existing flow
key attributes.  It must not change the meaning of already defined
flow key attributes.
==================================================================
```

This rule does have less-obvious consequences so it is worth working through a few examples. Suppose, for example, that the kernel module did not already implement VLAN parsing. Instead, it just interpreted the 802.1Q TPID (0x8100) as the Ethertype then stopped parsing the packet. The flow key for any packet with an 802.1Q header would look essentially like this, ignoring metadata:

```
eth(...), eth_type(0x8100)
```

Naively, to add VLAN support, it makes sense to add a new "vlan" flow key attribute to contain the VLAN tag, then continue to decode the encapsulated headers beyond the VLAN tag using the existing field definitions. With this change, a TCP packet in VLAN 10 would have a flow key much like this:

```
eth(...), vlan(vid=10, pcp=0), eth_type(0x0800), ip(proto=6, ...), tcp(...)
```

But this change would negatively affect a userspace application that has not been updated to understand the new "vlan" flow key attribute. The application could, following the flow compatibility rules above, ignore the "vlan" attribute that it does not understand and therefore assume that the flow contained IP packets. This is a bad assumption (the flow only contains IP packets if one parses and skips over the 802.1Q header) and it could cause the application's behavior to change across kernel versions even though it follows the compatibility rules.

The solution is to use a set of nested attributes. This is, for example, why 802.1Q support uses nested attributes. A TCP packet in VLAN 10 is actually expressed as:

```
eth(...), eth_type(0x8100), vlan(vid=10, pcp=0), encap(eth_type(0x0800),
ip(proto=6, ...), tcp(...)))
```

Notice how the "eth_type", "ip", and "tcp" flow key attributes are nested inside the "encap" attribute. Thus, an application that does not understand the "vlan" key will not see either of those attributes and therefore will not misinterpret them. (Also, the outer eth_type is still 0x8100, not changed to 0x0800.)

Handling malformed packets

Don't drop packets in the kernel for malformed protocol headers, bad checksums, etc. This would prevent userspace from implementing a simple Ethernet switch that forwards every packet.

Instead, in such a case, include an attribute with "empty" content. It doesn't matter if the empty content could be valid protocol values, as long as those values are rarely seen in practice, because userspace can always forward all packets with those values to userspace

and handle them individually.

For example, consider a packet that contains an IP header that indicates protocol 6 for TCP, but which is truncated just after the IP header, so that the TCP header is missing. The flow key for this packet would include a tcp attribute with all-zero src and dst, like this:

```
eth(...), eth_type(0x0800), ip(proto=6, ...), tcp(src=0, dst=0)
```

As another example, consider a packet with an Ethernet type of 0x8100, indicating that a VLAN TCI should follow, but which is truncated just after the Ethernet type. The flow key for this packet would include an all-zero-bits vlan and an empty encap attribute, like this:

```
eth(...), eth_type(0x8100), vlan(0), encap()
```

Unlike a TCP packet with source and destination ports 0, an all-zero-bits VLAN TCI is not that rare, so the CFI bit (aka VLAN_TAG_PRESENT inside the kernel) is ordinarily set in a vlan attribute expressly to allow this situation to be distinguished. Thus, the flow key in this second example unambiguously indicates a missing or malformed VLAN TCI.

Other rules

The other rules for flow keys are much less subtle:

- Duplicate attributes are not allowed at a given nesting level.
- Ordering of attributes is not significant.
- When the kernel sends a given flow key to userspace, it always composes it the same way. This allows userspace to hash and compare entire flow keys that it may not be able to fully interpret.