A Flow API for Linux Hardware Devices

John Fastabend

Intel
Portland, Oregon, USA
john.fastabend@gmail.com

Abstract

This paper proposes a Linux kernel-user API called Flow API that can be used to leverage hardware device flow tables. One goal of this API is to be generic enough to support a wide range of networking hardware and use cases. We believe this is essential to a successful API on an operating system that supports host network interface, top of rack switch devices and everything in between. In this paper we outline the insights that have guided the development of this API as well as illustrate how the API can be used by developers to write useful programs that can work across a wide array of devices without resorting to vendor specific code. To enable this API we have readily available C code that implements the API on top of rocker, an emulated switch supported by Linux. Additionally we provide a user space package to illustrate usages of the API which is also available. Finally we want to highlight some ongoing work and open problems under development.

Keywords

UAPI, Linux, switch, NOS, OVS, FlowAPI, offload, network accelerators.

Introduction

The user-kernel API called FlowAPI proposed here addresses a notable gap presented to hardware driver developers and user space application developers when attempting to use flow tables in hardware devices. Currently, developers have two options in Linux (a) Ethtool or (b) implement/use hardware specific APIs. The Ethtool model provides a simple view of the hardware with many limitations. Some of the notable limitations are only supporting a single ingress table, limited match support, limited number and type of actions and lack of capability queries. Additionally Ethtool uses a locking scheme which is likely restrictive for applications that want to modify the hardware state quickly possibly thousands of times per second. On the other side of the spectrum hardware specific APIs often packaged as software developer kits (SDKs) historical released by hardware vendors for switching silicon typically expose much more of the hardware to the user at the expense of portability. In general SDK's only support a specific vendor's hardware requiring developers to write vendor specific code. Also SDK's tend to be released with user-mode drivers and its unclear how in-kernel drivers would export this functionality except through proprietary and in this authors opinion ugly vendor specific Netlink or IOCTL interfaces.

When defining the Flow API we attempted to resolve the limitations of Ethtool and maintain a vendor neutral implementation. Towards this goal we identified some basic requirements:

- The API should be flexible enough to support many different hardware pipelines
- The API incorporates new packet types and actions easily. Preferably at run time
- The API should allow user space applications to configure the hardware pipeline and query the hardware for resource constraints
- The API should be vendor neutral

To support this the Flow API uses a set of API calls to expose hardware capabilities including supported headers, supported actions, supported tables, header graphs, and table graphs. Using this consumers of the API can create a usable model of most hardware devices we have been working with. Then a set of API calls to configure the device can be used to add rules, delete rules, create tables, and destroy tables.

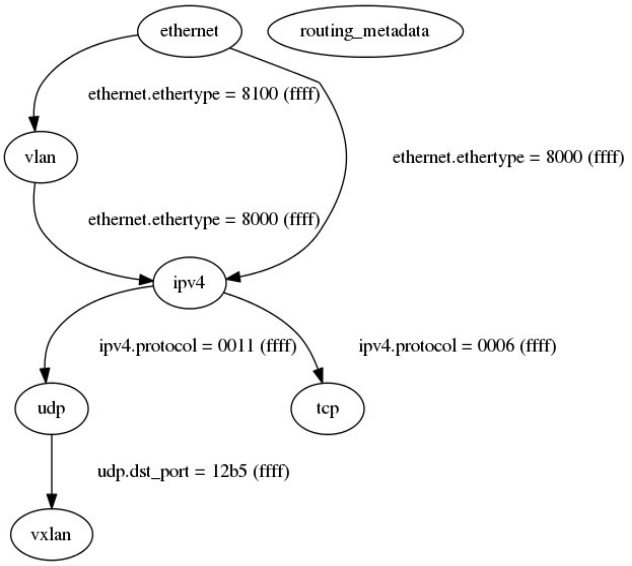
In the following sections we will show how a hardware model can be generated from the API, how it can be configured, and then finally provide some basic examples showing how programs can use this to handle hardware devices with different capability sets. The reader may use the publicly available source code as a reference.[1][2]

Device Model

The Flow API capability queries consists of a set of API calls built on top of the Netlink infrastructure. Netlink was chosen primarily because it provides a backwards compatible way to extend the API as needed. Additionally, we expect Netlink can support the performance requirements of most applications. The user API to query capabilities consists of the following commands:

- net_flow_table_cmd_get_headers
- net_flow_table_cmd_get_header_graph
- net_flow_table_cmd_get_actions
- net_flow_table_cmd_get_tables
- net_flow_table_cmd_get_table_graph

Illustration 1: Example header graph



Using these commands a model of the hardware pipeline can be created.

Supported Headers

The packet types supported by the model are queried through the *net_flow_table_cmd_get_headers* and *net_flow_table_cmd_get_header_graph* API commands. The first command *net_flow_table_cmd_get_headers* describes the supported packet headers as an ordered set of fields. Where each field is described with a unique identifier and bit width. For example a 802.1Q VLAN header is described here,

vlan { pcp:3 cfi:1 vid:12 ethertype:16 }

The user friendly strings are only provided as a convenience and are not required. If the user friendly strings are not provided the unique identifier which is a 32bit unsigned integer is used. Changing the user visible strings does not change the header layout for example,

foo { bar:3 bax:1 bay:12 baz :16 }

describes exactly the same packet header. This is just one example of a header. The query command returns an array of headers similar to the one illustrated above that expose the complete header space supported by the devices. This would typically include definitions for IPv4, IPv6, TCP/UDP, VXLAN, etc. Each of these definitions can be packed/unpacked into Netlink messages and passed from the hardware driver to the user space application. The Netlink definitions can be found in the source code in the UAPI file *./include/uapi/linux/if_flow.h*. Additional Netlink packing and unpacking helper functions provide program friendly structures that hardware drivers can provide and programs can consume. In kernel consumers of the API can use the data structures directly. These structures are not defined as part of the user-kernel interface but currently the same structures are being used in both the kernel and user space code. By not being embedded in the user-kernel API the structures can evolve independently.

With the above commands the set of supported headers can be exposed but the supported combination of headers still needs to be queried to get a complete view of hardware header support. This is relevant even on common host network interfaces. Considering the 802.1Q VLAN example used above some hardware may support stacked VLANs sometimes referred to as Q'in'Q others may only support offloads on the outer VLAN and still others may support 3 or more stacked VLAN headers. To expose this dependency and others like it the *net_flow_table_cmd_get_header_graph* API command is used to return a graph of the supported headers.

Illustration 1 provides an example graph of a simple device. The graph has a root node which is denoted with the user friendly string Ethernet. Following the Ethernet header it expects either a 802.1Q VLAN header or an IPv4 header. Any other headers will not be supported. Continuing through the graph shows support for UDP/TCP and VXLAN. We note that stacked VLANs are not supported by the single VLAN node.

The reader may also note fields that are not connected to the graph. In Illustration 1 there is a *routing metadata* node which is not connected. Metadata is used by the hardware for many reasons some common examples are to pass information between tables or encode information about the packet that is outside the table pipeline. This may be used to indicate the port the packet was received on, the queue it was received in, etc. Metadata referencing external information such as the two listed examples will need to be standardized. If the metadata is only used to pass information between tables it may be inferred from the device model. However “knowing” when this is a safe assumption is not knowable so we expect all metadata will need to be standardized.

6: dec_ttl (void)

7: set_dst_mac (u48 mac)

8: push_vlan (u16 vlan)

9: drop(void)

Illustration 2: Action Signatures

```
tcam: 1 src 1 apply 1 size 4096
  matches:
   field: ethernet [dst_mac (mask) src_mac (mask) ethertype (mask)]
   field: vlan [pcp (mask) cfi (mask) vid (mask) ethertype (mask)]
   field: ipv4 [dscp (mask) ecn (mask) ttl (mask) protocol (mask) dst_ip (lpm) src_ip (lpm)]
   field: tcp [src-port (mask) dst-port (mask)]
   field: udp [src-port (mask) dst-port (mask)]
  actions:
   1: set_egress_port (u32 egress_port)
   3: set_dst_mac (u48 mac_address)
   4: set_src_mac (u48 mac_address)
   5: trap()
```

Illustration 3: Table Definition

Denoting headers and header graphs this way provides a flexible mechanism for devices to expose packet headers in a way that consumers of the API can query. After obtaining the headers and header graph users can search for support of any packet headers they will require. To do this we only need to compare a proposed header graph and headers with the device. If the proposed header graph is a subgraph of the device then it can be supported. This is similar to other subsystems in Linux such as netfilter and traffic classifiers like u32 that use length, offset tuples to describe packets. [3,4] In our opinion this is more elegant then reporting bit masks of supported fields and is also easier to extend.

Supported Actions

In addition to headers and header graphs hardware devices also support a set of actions that can be applied to the packets and associated metadata. To query for supported actions the *net_flow_table_cmd_get_actions* command can be used.

Actions are expressed as a list of n-tuples. Each n-tuple includes a unique identifier, an optional string key, a signature and an action primitive list. In Illustration 2 an excerpt from an emulated device is shown. This particular excerpt is from the CLI included in the user space package it shows the unique identifiers and action signatures.

Currently the API supports a small subset of possible action primitives. We expect to expand this as needed. Although it is important the set continue to be a minimal set so that consumers of the API can evaluate the equivalence of actions. The current API supports these actions primitives

- set_field (field_id, value)
- push_header(header_id, value)
- pop_header(header_id)
- inc_field(field_id), dec_field(field_id)
- drop(void)

To see where this is relevant consider a device that supports an action with signature,

route(u48 DMAC, u16 VLAN)*

Querying the device may result in the following action primitive specification,

```
set_field (DMAC_FIELD, DMAC)
push_header(VLAN_HEADER, VLAN)
dec_field(IPV4_TTL)
```

as it may be possible to guess from the above description this action will set the destination MAC address, push a VLAN header on the packet and decrements the IPV4 TTL. Referencing Illustration 2 the same set of primitives (although we do not list the primitives in Illustration 2 we hope they can be inferred) can be achieved with this chain of actions

set_dst_mac(DMAC); push_vlan(VLAN); dec_ttl(void)

Using the primitives lists and signatures a consumer of the API can *normalize* the action sets across multiple devices. Its worth noting however that this will, in most cases need to be done up front at or near initialization time as we do not expect the computation can be done at runtime without impacting performance. For example we are experimenting with *normalizing* both actions and headers when devices are brought online.

At this point we should point out that these concepts have appeared in multiple places before we happily borrowed them to be used here. Some examples we are aware of

include the work by the Netfilter team, the traffic classifier "u32" and ongoing work at p4.org.[4] The team at p4.org for example has defined a more complete set of action primitives.

Tables and Table Graphs

The final piece to building the device model is to query the device for the tables and the table graph. Each table is identified by a unique identifier and an optional user friendly string. Tables consist of a set of supported actions and a set of supported matches. Illustration 3 shows an example of a possible table returned from a *net_flow_table_cmd_get_tables query.*

In addition to matches and actions tables may have various table attributes. At the time this was written supported table attributes include *size*, *source*, and *apply action group*. *Size* is used to indicate the total number of *rules* that can fit into a table. In the *Future Work* section we will discuss possible refinements of the size attribute for tables where *rules* may consume different amount of space in the table depending on the match/actions used. *Source* is used to indicate where multiple tables may be using the same physical resources. This will also be discusses in more detail later. *Apply action group* denoted by the apply value in Illustration 3 is used in combination with the table graph in Illustration4 to indicate when actions are applied to packets. Tables with the same *apply* value will always apply actions at the same time. Similarly actions with different *apply* values will be applied in serial. For a concrete example Illustration 3 shows a table labeled *tcam* with an apply group of '1'. Assuming there is another table *tcam2* with an apply group of '2'. In this case we can assume any actions from the table labeled *tcam* are already applied when the packet traverses the table labeled *tcam2*. However if both tables, *tcam* and *tcam2*, have an *apply group* value of '1' then any actions from *tcam* will not be seen by *tcam2*. One case where this is relevant is where we have actions to both set a field and match on it. In this case depending on the *apply group* value we may miss or match a rule.

The match fields give the supported header and associated fields along with the mask types supported by the field. Currently three types are supported, *mask*, *longest prefix,* and *exact.* As noted before the interface is designed to be extended so if another mask type is needed it can be easily added.

As noted above a device may support multiple tables (in fact this is the common case for many classes of devices!) and in order to effectively program such a device we need to show how packets are sent through the set of tables. The *net_flow_table_cmd_get_table_graph* as may be expected is used to provide this. The command will return a graph structure where each node is a table and the edges are the packet transition qualifiers. For shorthand we use unlabeled edges to indicate any unmatched packets. Illustration 4 shows one possible table graph. Here we show that Ethernet multicast packets are sent to a table labeled *bridge* and and all other packets are sent to a table labeled *ucast_routing.*

Using this along with a the other commands to query the tables, matches and actions consumers of the API can construct a functional model of the device pipeline.

Configuration

Adding and Removing Rules

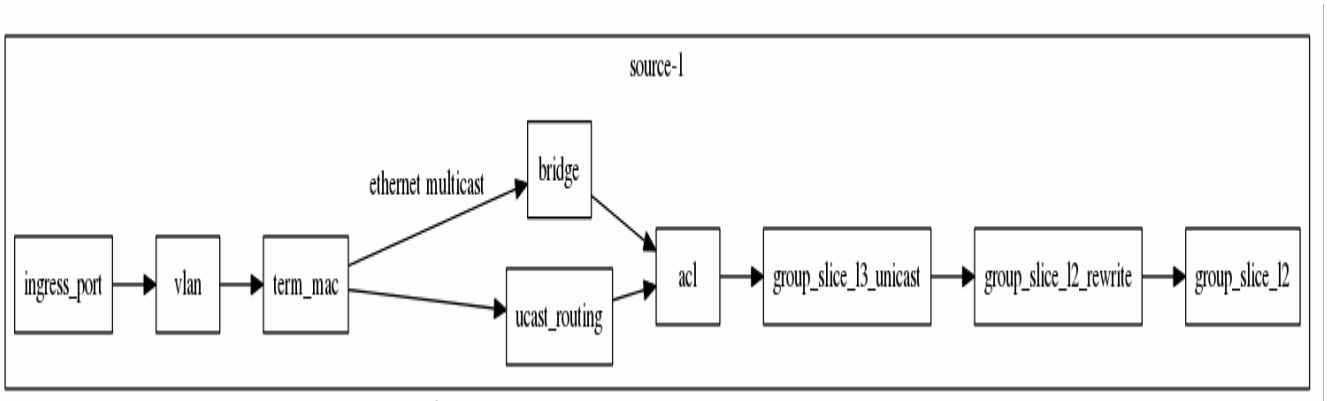
Once consumers of the API have queried the device the next item is to program rules into the tables. The following API commands allow this:

- net_flow_table_cmd_set_rule
- net_flow_table_cmd_del_rule
- net_flow_table_cmd_get_rule

The *net_flow_table_cmd_set_rule* encodes the following structure into Netlink messages,

```
struct net_flow_rule {
    u32 table_id;
    u32 uid;
    u32 priority;
    struct net_flow_field_ref *matches
    struct net_flow_action *actions}
```

Illustration 4: Table Graph



The table_id field references the unique identifier of the table. For reference in Illustration 3 the identifier given to the *tcam* table is '1'. The *uid* field gives a unique reference for the rule. The *uid* is scoped at the table level so it is only unique within a table. For a globally unique identifier the tuple (*ifindex, table_id, uid)* is sufficient where *ifindex* is the unique device id for the network device. If the device is not a kernel visible network device with an ifindex another field will be needed for the device identifier. The API supports a device type field but at the moment is only used to specify network devices.

The *matches* structure provides a NULL terminated array of match identifiers along with match values and a mask value. The specifics for each match can be determined by queries from the previous section. To meet the *matches* criteria a packet must match all entries in the array.

The *actions* structure provides the list of actions to apply when a packet meets the *matches* criteria along with the arguments required by the action. Similar to *matches* the arguments can be learned by the queries previously described.

Perhaps an interesting point to make is validation code to verify flows are valid can be auto-generated using the model. This allows a common block of validation logic to be used by all devices that expose a device model. The verification code ensures that the input match and action signatures match the model and are valid for the table/device being targeted by the rule.

The *net_flow_table_cmd_del_rule* is somewhat easier then the add rule counterpart. To delete a flow only the globally unique identifier *(ifindex, table_id, uid)* needs to be specified.

Finally having support for adding and deleting rules querying a device for currently installed rules may be useful. For this the *net_flow_table_cmd_get_rule* command is supported. As we will see in the a subsequent section this does not require any driver support and is handled by the core API logic. To get flows a *table_id* is specified along with a rule identifier range to report. This is useful for large tables that may contain many rules. If the range is omitted the API will report the entire table.

Device Drivers

Writing a device driver to support the API requires implementing the following operations to support the query API:

- ndo_flow_get_actions
- ndo_flow_get_tbls
- ndo_flow_get_tbl_graph
- ndo_flow_get_hdrs
- ndo_flow_get_hdr_graph

These are specified as part of the *struct net_device_ops* in *./include/linux/netdevice.h*. These are called by the query API previously outlined to return a portion of the device model. For the class of networking devices that use static models, meaning the model is fixed for the device, these routines simply need to return a structure describing the query request. However this does not preclude a class of device that may be reconfigured at runtime. These devices will require a more complicated handler to retrieve the current configuration. There is ongoing work to configure the model at runtime starting with the creation and deletion of tables discussed in future work.

Additionally there are two driver operations to support adding and removing rules:

- ndo_flow_set_rule
- ndo_flow_del_rule

Each of these routines are exercised by the *Configuration* API commands discussed previously. The set rule operation consumes a structure *net_flow_rule* shown in the previous section. The driver can then use a set of case statements or a lookup table to translate the rules into a hardware specific implementation. No validation of the rule needs to occur in the driver itself because as noted previously the API validates flow messages to ensure they are valid in the model. Similarly deleting a rule will call the *ndo_flow_del_rule* op and pass a tuple identifying the flow *(device_id, table_id, uid)* to the driver. The driver needs to translate this into the hardware specific rule deletion operations.

Netlink Implementation

The Netlink implementation creates a new Netlink family that is used to consume Netlink messages and translate those into the device driver calls. As noted previously the Netlink interface also provides validation for the configuration side and a cache of the current flow state of the device. By maintaining the flow state in the interface we avoid having to query the device driver for to handle query requests. This table is implemented as a re-sizable hash table. This way we avoid consuming memory when tables are not in use.

Another concern is locking specifically at what granularity to do the locking. The available implementation does locking at the Netlink interface level which is not particularly granular. At minimum this should be done at the per device level. An further enhancement would be to lock on a per table basis. We are currently investigating how easily per table locks would be implementable across devices. We are aware of at least one device where this could be supported.

We are not opposed to embedding the API into existing Netlink families if this helps to consolidate code and tooling in the Linux networking subsystem. The primary

concern is trying to maintain a locking model that will allow the Flow API to scale to many operations per second. One proposal has been to use the existing 'tc' infrastructure as a container for the user-kernel API.

Future Work

The described Flow API provides a base implementation to expose the hardware device. Although it is in our opinion a reasonably good first implementation there are many areas that can be expanded.

Configurable Models

The Flow API as discussed previously primarily used static models as examples. Further commands to configure the model at runtime are notable missing.

We are aware of devices that support creating and destroy additional tables. There is current work ongoing to incorporate this into the API. This involves using the *source* attribute in the table attributes section to indicate sub-tables of a parent table. This can be used as a hint to the hardware device the only a subset of the actions or matches in a table will be needed. For example a table that supports both IPv4 and IPv6 may allow for certain performance or space optimization if ahead of time the user indicates that only IPv4 is being used. In this case we could allow a create table operation with the source of the parent table and a list of required actions/matches specifying a subset of the parents to include only IPv4 fields. The driver when receiving this command can create the optimized table.

For devices which support programming the header graph or even action lists it may be possible to add similar commands to insert new header types and actions.

Device Model Maps

The observant reader will notice we provided hints on how to determine equivalent headers and actions between devices, but did not provide any hints on how to map rule additions between tables.

We are working on a couple approaches here. If you can make the assumption that the rules are orthogonal. Then by traversing the graph we can insert the rule in all paths or throw an exception if it is not possible to insert the rule. It might be possible to further refine this to throw specific exceptions if there are escape cases where specific paths may fail but some paths through the pipeline can support the rule. It appears that it may be possible to generate this mapping up front at initialization time allowing the computation to be skipped when rules are added. Further requiring orthogonal rules at first seeming a bit extreme has some “real-world” examples today. The predominate one being the common kernel datapath used by Open-vSwitch. This however does somewhat under utilize hardware resources.

Another approach currently under investigation is to provide a mapping from a single table to the device model provided by the hardware. This mapping can optimize the flows as much as possible. If another map can be provided from the user visible map onto the single table model then the composition of the functions gives a mapping from the user model onto the hardware model. Illustration5 attempts to highlight this idea. More investigation is needed to decide if it is feasible in practice.

Validation

Providing maps between API consumer models and driver models raises questions of correctness. While algebraically proving correctness may be useful we may also want to test models with actual packets. This becomes even more relevant when the pipeline is configurable.

Using the query APIs we believe there should be enough information to generate test cases and test packets that can then be injected into the pipeline. One possibility being to generate pcap files and use tcpreplay to inject packets into the pipeline. This would allow a test coverage metric to give the amount of the pipeline that has been tested.

Conclusion

The Flow API proposed in this paper provides consumers both in the kernel and in user space with an available API to manage hardware devices that support flow tables. We believe it as an implementation combining many good ideas from the various sources noted in the Reference and throughout the paper. In addition to the base API we also provide a user space CLI that can be used to directly manage the device. Perhaps more interesting the API is designed so that future extensions can be made as needed. We expect that existing applications and virtual switches can leverage the API to offload operations that are currently being done in software. Finally continuing to explore the work items in the *Future Work* should enable more seamless integration and validation.

References

1. John Fastabend, “net-next-rocker”, github website, accessed February 10, 2015, https://github.com/jrfastab/rocker-net-next
2. John Fastabend, “user space tool”, github website, accessed February 10, 2015 https://github.com/jrfastab/iprotue2-flow-tool
3. Pablo Neira Ayuso“nftables HOWTO”, nftables website, accessed February 10, 2015 http://wiki.nftables.org/
4. Bert Hubert, Gregory Maxwell, et. al,“Linux Advanced Routing & Traffic Control HOWTO”, tldp.org, accessed February 10, 2015 http://tldp.org/HOWTO/Adv-Routing-HOWTO/
5. Pat Bosshard, Dan Daly, Glen Bibb, et. al., “P4: programming protocol-independent packet processors”

ACM SIGCOMM Computer Communication Review
Volume 44 Issue 3, (July 2014): 87-95

Author Biography

John Fastabend is a network engineer employed by Intel Corp. he works on Linux kernel networking components, various hardware devices, some user space applications and miscellaneous other software components.