Using eBPF as an Abstraction for Switching

Nicolaas Viljoen, Jakub Kicinski
Netronome Systems
Santa Clara, United States
nick.viljoen@netronome.com, jakub.kicinski@netronome.com

Abstract

eBPF (extended Berkeley Packet Filter) [1] has been shown to be a flexible kernel construct used for a variety of use cases, such as load balancing [2], intrusion detection systems (IDS) [3], tracing [4] and many others [5]. One such emerging use case revolves around the proposal made by William Tu for the use of eBPF as a data path for Open vSwitch [6]. However, there are broader switching use cases developing around the use of eBPF capable hardware. In this paper the focus will be around the use of a the multi-host NIC platform as the first consumer for this type of abstraction, however container switching and isolation will also be touched upon.

Keywords

eBPF, TC, XDP, offload, Switchdev, fully programmable hardware

Introduction

Switching abstractions within the kernel are not a new concept. In 2014 Jiri Pirko introduced switchdev [7]. This provided an in-kernel driver model for the offload of the forwarding plane, and has been applied to OVS offload as well as to switch based Linux [8]. However, previously the forwarding plane has been handled mostly with stateless L2/L3 forwarding. This paper will use the concept of a multihost NIC to motivate the extension of this functionality further.

First this paper will describe the work which is currently upstream. Jakub Kicinski introduced the use of switchdev for the multihost NIC platform, followed by the introduction of the concept of qdisc offload using RED [9]. Thereafter the next steps for the extension of the offload will be introduced. This will involve generalising the qdisc offload and introducing simple classifiers, such as u32. Finally the concept of multihost eBPF offload will be covered. This will include changes in the general offload architecture as well as the use of cls_bpf and XDP offload for switching functionality.

The Multi-Host NIC Concept

Many datacenter operators are starting to utilise multihost platforms, both within large datacenters and small edge datacenters/POPs [10]. This in turn has meant the advent of a new type of networking device, the multi-host NIC. The multi-host NIC is in effect a small L2 switch which provides users the ability to switch traffic to the correct PCIe device attached to the NIC.

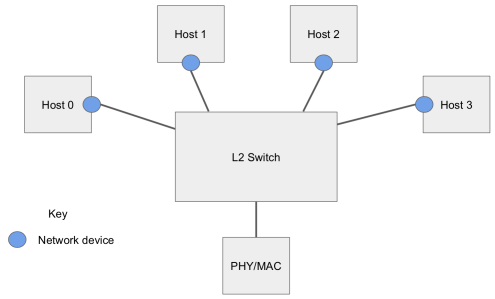

Figure 1: Standard Multi-host system today-Note the lack of switch representers

This however brings up the question of how to represent these devices. The current approach used by most vendors is to represent the NIC to each host as a single netdev. Any statistic related to the host are linked to that netdev. In this model there is no concept of the NIC as a switch. While there are certainly many merits to this model, it does have some shortcomings.

Debugging: Being unaware of what is occurring in the switch and on the other hosts when sharing a device such as a NIC makes it difficult for the user to be able to understand the origin of certain networking problems. Adding counters to be able to monitor the physical port does assist with this, however it does not provide the granularity usually associated with a netdev.

Queueing Discipline: The multihost NIC may be a bottleneck in some circumstances, being able to implement sensible methods of queueing to avoid unexpected reductions in throughput or latency are essential.

Offloads: Being able to link offloads to either the ingress of the host, the egress of the switch or to the entire set of hosts may be significantly beneficial in certain circumstances due to the expansion of use cases as well as per-

formance, code store or FPGA gate constraints in network processing elements.

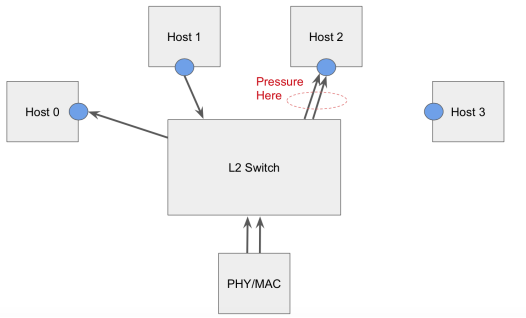

Figure 2: An example of the types of issue which may be challenging to debug without switch representers

Through the use of switchdev as a representation of the NIC, the above shortcomings can be mitigated. As switchdev allows for the use of representers at both the physical ports and the logical switch ports, this allows for the attachment of qdiscs and other offloads. In terms of debugging, the ability to have a full set of statistics at each of these points may be cleaner than adding significant amounts of counters to a single netdev. Debugging may also be assisted through the concept of read only netdevs, which will be explored in the future work section.

Current Work: Switchdev and Basic Qdisc Offload

This section covers the work which has already been upstreamed by the team-Jakub Kicinski deserves the majority of the credit for this work. It provides an overview of the mechanics behind the application of switchdev and simple qdisc offload.

The switchdev based multihost architecture

To represent a switch, representers are created for each of the physical ports, as well as the logical switch ports to the hosts. This is combined with the representers for the host vnic's. The diagram below shows the general layout.

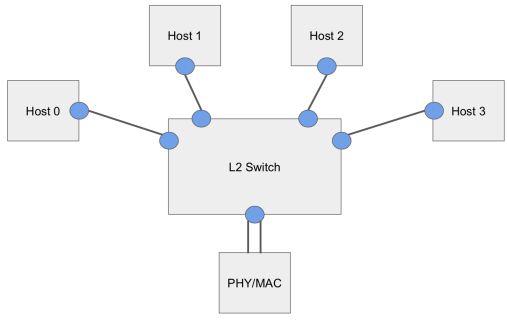

Figure 3: Switchdev based multihost architecture

The Netronome Flow Processor Firmware

The NFP's architecture has been previously outlined [11]. This has not significantly changed. The only key additions are the concept of a queue manager flow processing core and a PF mailbox. The queue manager manages the offloaded egress queueing disciplines and ensures that throughput and latency QoS is handled as expected. The switching is handled by the main application flow processing cores. Much of the total work done for the egress processing requires in-order handling, therefore the queue manager is reached after passing through the reorder block. The PF mailbox is used for communication which is not networking or port related.

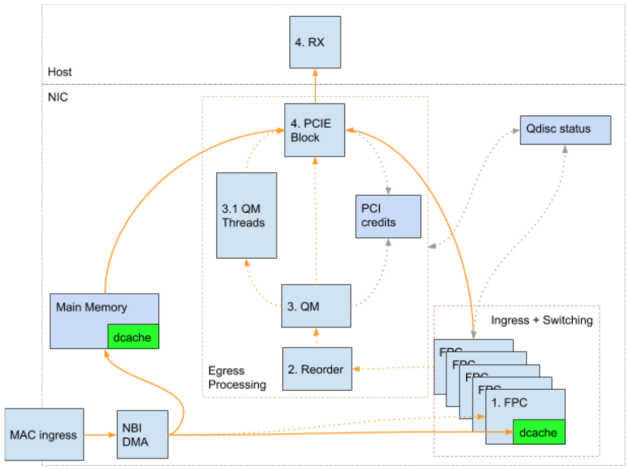

Figure 4: Netronome FW life of a packet-wire to host

Driver Architecture

The driver setup consists of three stages:

a) Initialisation

b) Switching mode

c) qdisc offload

Initialisation The initialisation occurs at boot time when the PCI probe occurs. This process consists of two key steps, app initialisation and vnic allocation.

App Initialisation The concept of the app is one used widely within the NFP driver. App abstractions are designed to match certain flavours of firmware and allow ease of infrastructure reuse. For example, if the firmware detected on the NIC is multi-host capable, then the .init function pointer is directed at the abm_init code in nfp/src/abm. ABM denotes advanced buffer management, which is the overarching term currently given to the design of the multihost NIC architecture with qdisc offload.

```
const struct nfp_app_type app_abm = {
        .id             = NFP_APP_ACTIVE_BUFFER_MGMT_NIC,
        .name           = "abm",

        .init           = nfp_abm_init,
        .clean          = nfp_abm_clean,

        .vnic_alloc     = nfp_abm_vnic_alloc,
        .vnic_free      = nfp_abm_vnic_free,

        .port_get_stats         = nfp_abm_port_get_stats,
        .port_get_stats_count   = nfp_abm_port_get_stats_count,
        .port_get_stats_strings = nfp_abm_port_get_stats_strings,

        .setup_tc       = nfp_abm_setup_tc,

        .eswitch_mode_get       = nfp_abm_eswitch_mode_get,
        .eswitch_mode_set       = nfp_abm_eswitch_mode_set,

        .repr_get       = nfp_abm_repr_get,
};
```

Listing 1: Current app abstraction layer for the abm app

The abm_init function carries out the following functionality

a) Checks that eth table exists and BAR symbols are correctly configured. The eth table is required to ensure that enough MACs are assigned to the device to be able to function as a multihost NIC

b) Create the nfp_abm structure: This structure contains a back pointer to the nfp_app, the pf id, as well as the eswitch mode and data about queue levels/stats.

c) Ensures that at boot time legacy mode is enabled to ensure compatibility-this is done through the disabling of the queue manager and other modified NIC egress processing elements

d) Allocate representors for all of the vnics which could be enabled in switchdev mode.

vNICs Allocation The second key section of the initialisation sequence is the vnic allocation:

a) Sets up the nfp_abm_link structure, the nfp_abm_link structure is what is used to link offloaded qdiscs to the representor/network device.

```
struct nfp_abm_link {
        struct nfp_abm *abm;
        struct nfp_net *vnic;
        /* snip */
        struct nfp_red_qdisc *qdiscs;
};
```

Listing 2: Key elements within struct nfp_abm_link

b) As the system is multi-host, ensure that the MAC/PHY state does not follow any of the ports.

c) set the vnic MAC addresses

At this stage the shared nfp_abm structure is set up as well as abm_link for the vnic (contained within struct nfp_net's private application specific data).

Switching Mode The next step is the setting of the switching mode, this is triggered by the .eswitch_mode_set callback, also contained within the nfp_app structure. If setting switchdev mode, what this process does is that it spawns a PORT and PF representer for each of the vnics. Spawning consists of a number of steps:

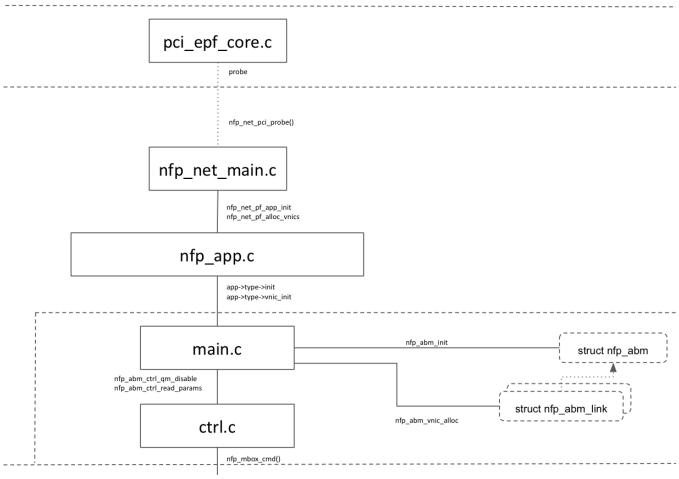

Figure 5: Initialisation

a) Create the netdev

b) Allocate the queues to the netdev. Note that for a physcial port this is currently a single queue.

c) Link the netdev to the representer

d) Attach the link to the representer in the app specific space

e) Initialise the port structure

f) Initialise the representer

g) Add the representer to the list of representers

h) Initialise queue manager on the NIC, ensuring that the FW is ready for offloaded qdiscs

At this point the representers are all attached to the abm_link structures, this allows offloaded RED qdiscs to be easily attached. Note there are also other commands to get/set the size of the shared buffer pool on the NFP for use by the RED qdisc as shown in the figure below.

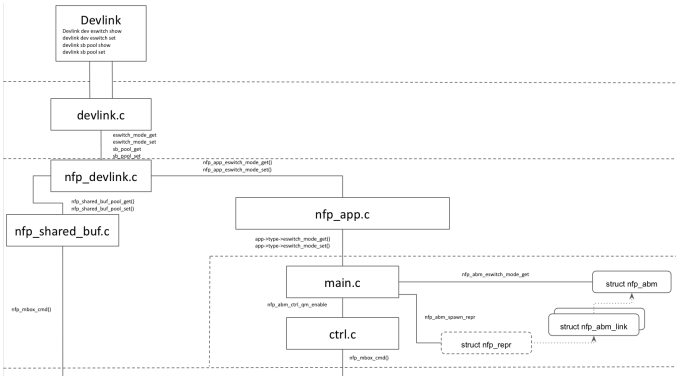

Figure 6: Entering switchdev mode

Qdisc Offload The qdisc offload works through the .ndo_setup_tc call. If looking to use the RED offload, the nfp_abm_ctrl_set_q_lvl() and other associated functions are used to interact with the mailbox abi. This sets the queue levels and handles interaction with the firmware. As previously stated the nfp_abm_link structure is used to keep track of the qdiscs. This is through the nfp_red_qdisc structure. Statisitics are also a key feature, through the use of tc it is possible to maintain access to the standard qdisc statistics, which in this case are extremely helpful for monitoring of state of the qdisc via backlogs, drops etc.

```
struct nfp_red_qdisc {
        u32 handle;
        struct nfp_alink_stats stats;
        struct nfp_alink_xstats xstats;
};
```

Listing 3: the nfp_red_qdisc contained within the nfp_abm_link structure

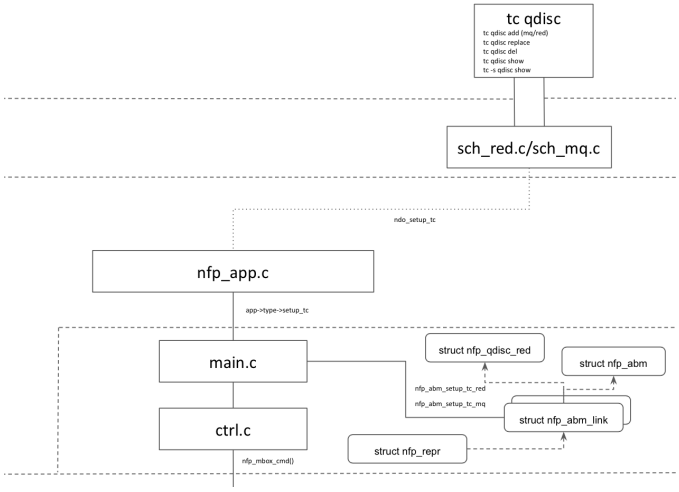

Figure 7: Qdisc offload

Next Steps: Extending the egress representer architecture

As the above section shows, the current upstream code provides a model for offloading the RED qdiscs. This section is focused on how this should be generalised to more types of qdiscs, including clsact, allowing the use of classifiers such as u32.

Generalising Qdisc offload

To allow more forms of qdisc offload, the first step is to create a more generic qdisc structure to be used. This structure would contain an enum to determine the type, as well as a union for qdisc type specific features. It would also be important to ensure that the qdisc structure is associated with a specific netdev, as different netdev's may be able to offload different types of qdiscs. Therefore being able to use this for checks when new child qdiscs are added to a chain is important.

```
struct nfp_qdisc {
        struct net_device *netdev;
                enum nfp_qdisc_type type;

        /* snip */

                struct nfp_qdisc **children;

        /* snip */

                union {
                        /* NFP_QDISC_MQ */
                struct {
                        struct nfp_alink_stats stats;
                                struct nfp_alink_stats prev_stats;
                        } mq;
                        /* TC_SETUP_QDISC_RED */
                        struct {
                        bool ecn;
                                u32 threshold;
                        struct nfp_alink_stats stats;
                        struct nfp_alink_stats prev_stats;
                        struct nfp_alink_xstats xstats;
                        struct nfp_alink_xstats prev_xstats;
                        } red;
                };
};
```

Listing 4: Proposed generic nfp_qdisc structure

The clsact Qdisc

With the introduction of this infrastructure, it may be of significant advantage to be able to offload classifiers, allowing packets to be steered to the correct qdiscs. The addition of a classifier such as u32 allows the utilisation of features like priority maps which could be used for qdiscs such as GRED. This can easily be handled within the current app abstraction.

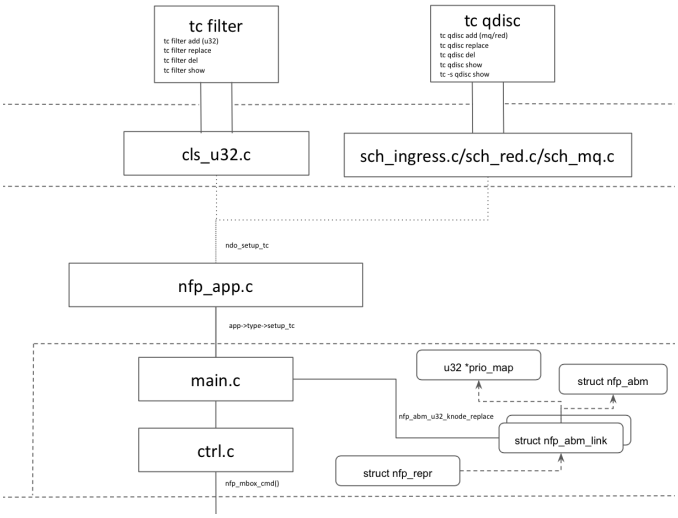

Figure 8: The addition of a cls_u32 qdisc

This feature provides a powerful tool, not only to identify different priorities of traffic in a stateless manner, but also to provide more efficient ways to handle QoS in a heterogenous congestion control environment.

Looking Further: Future Work-Multihost BPF Offload

This section covers work the team is looking at further into the future. The main topic to be covered relates to the archi-

tectural changes required to support multihost switch based BPF offload, including relevant firmware and JIT modifications, cls_bpf offload and finally XDP offload.

Firmware and BPF JIT

There are a number of key changes that would need to be made to the way offload is done currently to be able to use offload effectively within a multihost environment. Firstly, as most offload devices will have limited resources available, such as gates or code store space, it makes sense to be able to share programs between different ports. This could either be done with a simple jump table in the underlying logic, or potentially through the use of a small BPF program in and of itself. This architecture would also allow for easy support of tail calls and also allow circular dependency of programs. There may potentially be some advantages to ensuring this mechanism is as generic as possible to avoid architecture specific quirks.

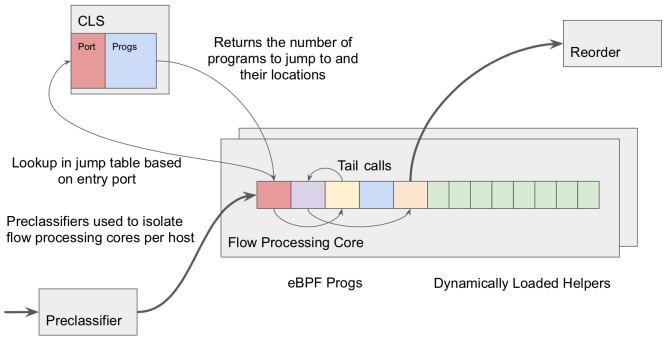

Figure 9: Proposed flexible architecture with jump table to manage the reuse of BPF programs

Secondly, there may be cases where it is important to isolate the resources of a particular host. To ensure this isolation, it is important to be able to isolate certain flow processing cores to handle traffic associated with that host. Finally, as will be covered further in the section below. It will be important for the NIC to have a concept of which type of port the program is attached to as there are now multiple types of representers which the program could be attached to, not only physial port representers, but it may also be attached to the logical switch port ingress as a XDP program, or to it's egress as a cls_bpf program.

cls_bpf and Switchdev

The next steps section described how to offload simple stateless classification such as cls_u32. However for more complex classifiers such as bpf, there is currently a separate app abstraction. The first step would be to move this to the generalised infrastructure. This would need to be combined with the use of the pf id to be able to ensure that the jump table or other similar structure is correctly setup. This program would then be DMA'd into the NIC at the specified offset as is done within the single host case today. This is then controlled via the control message $cmsg$ interface and runs in the out-of order FPCs as per normal.

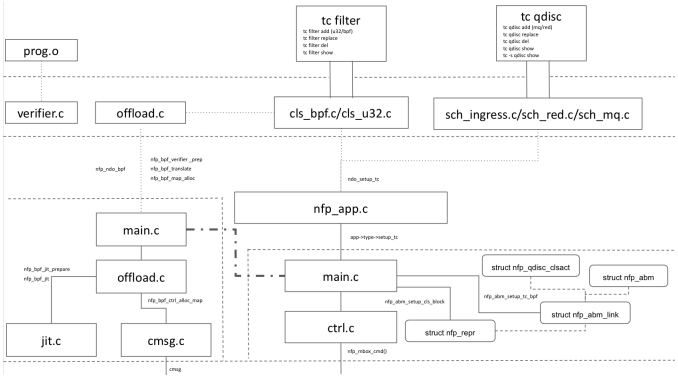

Figure 10: Incorporation of the eBPF infrastructure in ABM

The use of BPF programs on the egress of the logical switch ports means that stateful quality of service may be possible to deliver within a multihost environment. For example, high throughput connections with long RTTs may be identified and could be handled in a manner that is more conducive to the large window sizes required for their handling, such as larger thresholds or different congestion behaviours-drop, mark, buffer etc.

XDP in the Multihost Case

An addition to this architecture which is being explored is the possibility of using offloaded XDP to create a fully flexible datapath via the use of bpf_redirect() to steer traffic. There are however some obvious problems which would arise;

a) XDP is an RX exclusive hook: This means that a single netdev would not be able to provide bidirectional switching.

b) Heterogenous architecture support is nascent: Many switch architectures may not be flexible enough to be used with XDP, therefore this may require further work

c) Security: Who would be trusted to add programs to an interface shared by the four endpoint hosts?

However this is a series of problems which are being addressed in the community. William Tu is currently looking at the development of the P4-XDP compiler infrastructure[13], which may provide some intriguing insights into heterogenous processing. Jakub Kicinski has also presented some initial proposals in the BPF micro conference for the use of BPF as a heterogenous processing ABI[14]. Through the use of switchdev it is possible to provide an ingress hook at both the logical switch port and at the physical port representer. There are also constantly problems which are being addressed; For example, until recently XDP had no infrastructure for access to the FIB table, therefore would waste unnecessary resources updating maps duplicating already established functionality if attempting to provide switching functionality. David Ahern has added this functionality and will discuss providing BPF

access to the FIB table at LPC 2018[12].
The one problem which is not easily addressable is the question of whom would be allowed to add functionality on the external port. Currently the assumption would have to be that the hosts are all trusted if this functionality is enabled. For non-secure cases platforms such as traditional switches or so-called 'Bare-Metal NICs', both of which contain a large controller CPU, may be able to provide a solution.

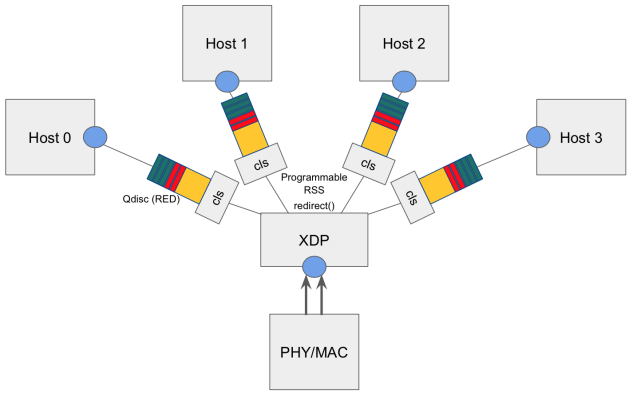

Figure 11: Overall architectural goal-BPF defined datapath

This would allow an architecture such as that shown above, ensuring the end user has significantly more granular control over the entire the NIC datapath through upstream Linux than is possible today. This could then also be used by others with programmable switch or multi-host NIC hardware.

Statistics

As described previously, one of the key advantages of the switchdev based architecture is that it provides a conceivable interface to providing visibility into what is occurring on the entirety of the NIC by providing read only visibility to all the egress ports of the logical switch. This would allow problems such as internal bottlenecks in the switch as described in figure 2 to be easily identified.

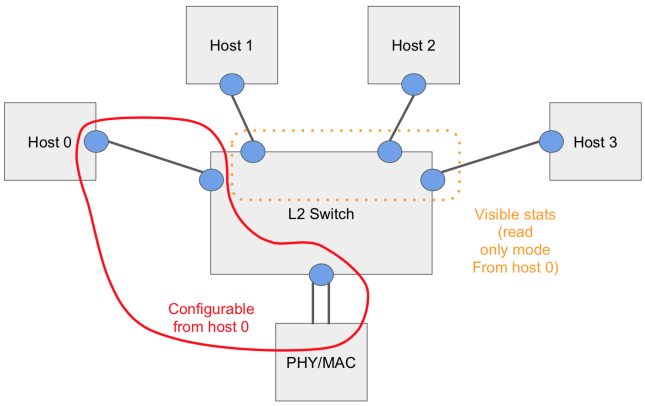

Figure 12: Read only representers ensure ease of debugging

Conclusion

This paper has laid out a proposal for an architecture which can be used to define a fully flexible data path for a mutli-host NIC using the abstraction provided by switchdev combined with the offloading of qdiscs, stateless classifiers and multiple bpf hooks associated with multiple ports on a logical switch. Currently the switchdev architecture and qdisc offload has been upstreamed and the next steps are in development, however further work built on top of that currently described by others [12, 14] could provide the potential for upstream BPF defined pipelines more generally within heterogenous architectures including, NICs and switches.

References

[1] Starovoitov, A. et al., Linux Socket Filtering aka Berkeley Packet Filter (BPF)
Linux Kernel Documentation.

[2] Shirikov, N., Dasineni R., Open-sourcing Katran, a scalable network load balancer
https://code.fb.com/open-source/open-sourcing-katran-a-scalable-network-load-balancer

[3] Yates, T., Using eBPF and XDP in Suricata,
https://lwn.net/Articles/737771/

[4] Gregg, B., Linux eBPF Tracing Tools
http://www.brendangregg.com/ebpf.html

[5] Borkmann, D., On Getting the TC Classifier Fully Programmable with cls_bpf, *NetDev 1.1*.

[6] Tu, W., Patch set for eBPF based OVS
https://mail.openvswitch.org/pipermail/ovs-dev/2018-June/348521.html

[7] Pirko J., Switchdev-No More SDK *NetDev 1.1*.

[8] Ahern, D., Shrijeet N., Building a Better NOS with Linux and switchdev
https://www.files.netdevconf.org/d/cb35a26e23e744318860

[9] Kicinski J. Patch set for initial multi-host NIC switchdev work
https://lists.openwall.net/netdev/2018/05/22/33

[10] Chong, D., Bryan J., Twin Lakes: 1S Server for Yosemite V2
https://www.opencompute.org/files/Intel-FB-TwinlakesBryan-Chong-OCP18.pdf

[11] Kicinski J., Viljoen, N. XDP Hardware Offload: Current Work, Debugging and Edge Cases
https://www.netdevconf.org/2.2/papers/viljoen-xdpoffload-talk.pdf

[12] Ahern, D. Leveraging Kernel Tables with XDP
https://www.linuxplumbersconf.org/event/2/contributions/93/

[13] Tu, W., P4C-XDP: Programming the Linux Kernel Forwarding Plane Using P4
https://www.linuxplumbersconf.org/event/2/contributions/97/

[14] Kicinski J. Using eBPF as a heterogeneous processing ABI
http://vger.kernel.org/lpc-bpf.html