Leveraging Kernel Tables with XDP

David Ahern
Cumulus Networks
Mountain View, CA, USA
dsahern@gmail.com

Abstract

XDP is a framework for running BPF programs in the NIC driver to allow decisions about the fate of a received packet at the earliest point in the Linux networking stack. For the most part the BPF programs rely on maps to drive packet decisions, maps that are managed, for example, by a user space agent. While high performant, this architecture has implications on how a solution is coded, configured, monitored and debugged.

An alternative approach is to make the existing kernel networking tables and implementations accessible by BPF programs. This approach allows the use of standard Linux APIs and tools to manage networking configuration and state while still achieving the higher performance provided by XDP. Allowing XDP programs to access kernel tables enables consistency in automation and monitoring across a data center – software forwarding, hardware offload and XDP "offload".

An example of providing access to kernel tables is the recently added helper to allow IPv4 and IPv6 FIB and nexthop lookups in XDP programs. Routing suites such as FRR manage the FIB tables in response to configuration or link state changes, and the XDP packet path benefits by automatically adapting to the FIB updates in real time. While a huge first step, a FIB lookup alone is not sufficient for many networking deployments. This paper discusses the advantages of making kernel tables available to XDP programs to create a programmable packet pipeline.

Keywords

XDP, eBPF, forwarding, Linux.

Introduction

Linux is a general purpose OS with established implementations for networking features and APIs for configuring, monitoring and troubleshooting. In addition, Linux has a rich ecosystem of software written to those APIs – standard utilities for configuring networking (e.g., iproute2 and ifupdown2), routing suites such as FRR, standard monitoring tools such as net-snmp, collectd, and sysdig, and automation via ansible, puppet and chef. The standard Linux APIs allow this software ecosystem to work across OS versions (to various degrees based on kernel features) providing a consistent and stable means for automation and management across data center deployments using the Linux networking stack.

As a general purpose OS, Linux has a lot of features making it useful across a variety of use cases, however each of those features impacts the performance of specific deployments. The need for higher performance has led to the popularity of specialized packet processing toolkits such as DPDK. These frameworks enable unique, opaque power-sucking solutions that bypass the Linux networking stack and really make Linux nothing more than a boot OS from a networking perspective.

Recently, eBPF has exploded in Linux with the ability to install small programs at many attach points across the kernel enabling a lot of on-the-fly programmability. One specific use case of this larger eBPF picture is called XDP (eXpress Data Path) [1]. XDP refers to BPF programs run by the NIC driver on each packet received to allow decisions about the fate of a packet at the earliest point in the Linux networking stack (Figure 1). Targeted use cases for XDP include DDOS protections via an ACL [2], packet mangling and redirection as a load balancer [3], and fast path forwarding by updating the ethernet header and redirecting to a different interface [4][5]. XDP provides the potential for a more Linux friendly alternative to DPDK, allowing custom, targeted processing on packets that scales linearly with CPU cores [6].

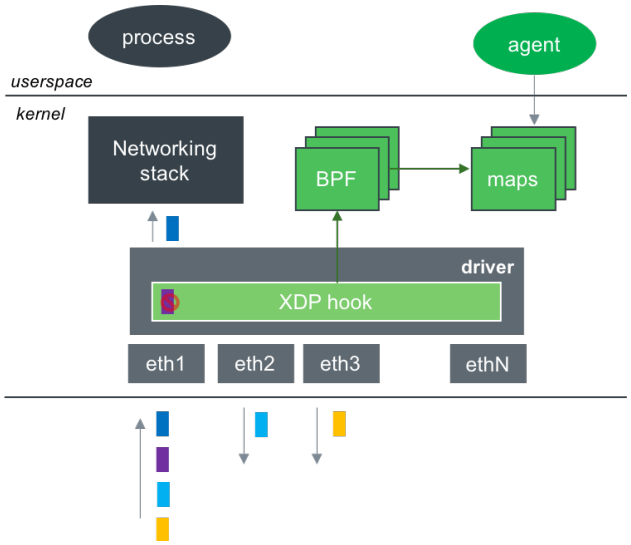

Figure 1. XDP Architecture.

The performance benefit of XDP is because programs are run at the earliest point in the packet processing path, before any allocations have been done for example. A side effect of this design is that BPF programs run in the XDP context have few kernel helpers (kernel code that can be invoked by a BPF program), most notably because the

Linux networking stack operates on skbs and the XDP context is before an skb is allocated. This means bpf programs have to do their own packet parsing, processing and rewrites, and it drives a tendency to reimplement networking functionality in BPF form, be it something as simple as an allow/drop ACL to iptables and standard protocols such as bridging or layer 3 routing.

Furthermore, there are few options for managing configuration data and state, most notably through BPF maps or hardcoded data embedded in the program. From a configuration perspective BPF maps are managed by a userspace agent meaning the design needs to rely on orchestration software or it needs to track kernel state (e.g., via notifications) and replicate the data in the maps.

The end result is either a duplication of implementations for networking features (which means subtle differences between Linux forwarding and XDP forwarding) or each XDP solution is a unique one-off from a Linux perspective.

It is good to see Linux gain high performance improvements and innovations that enable new solutions such as Cilium [7] with its L4 to L7 policy enforcement, but for a lot of established packet processing use cases (e.g., firewall, NAT, forwarding), a better end goal is for XDP to integrate into the Linux stack in a more consistent way - one that maintains the bigger Linux picture and does not require duplicating code to leverage the new performance capability. If XDP means protocols are reimplemented in BPF with opaque data maps and users have to abandon their existing tools, processes and workflow to work with XDP solutions, then is it really any different than a DPDK based solution? In both cases the Linux networking stack is bypassed in favor of non-standard code with non-standard management, non-standard monitoring, non-standard troubleshooting - all hallmarks of the userspace bypass techniques.

Tools like bpftool [8] certainly help to a degree in that it provides a common tool for introspection of programs and maps, but the programs and meaning of the maps can and will vary by solutions, vendors and versions. Debugging or understanding a deployment that leverages BPF everywhere is unique to each instance. This creates havoc on a support organization when every node in the data center is a unique solution creating a configuration, monitoring and support nightmare and negating powerful capabilities like automation and devOps. While new contexts like XDP will inevitably affect the use of some tools (e.g., packet captures like tcpdump), it does not mean all tools and APIs need to be abandoned.

The rest of this paper focuses on one use case to drive home the need for better integration of XDP with existing Linux APIs: fast path forwarding with XDP. There are 2 options for an XDP forwarding architecture that relies on a bpf program driven by maps managed by a user space agent: 1. the agent is managed by an SDN controller for example, receiving updates from a remote controller, or 2. the user space agent listens for rtnetlink notifications and updates the maps as changes happen (Figure 2). The former is another example of bypassing all of the standard

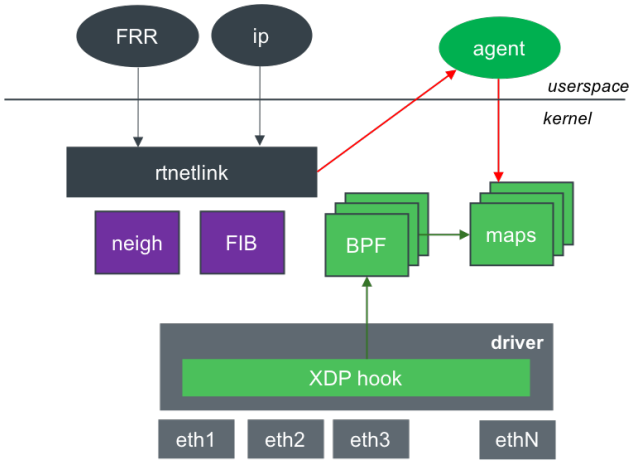

Figure 2. Forwarding with XDP and rtnetlink snooping.

networking protocols and Linux APIs; the latter is the model used by the sample router in the kernel source tree [4]. While such a technique could be useful in simple deployments, it has many shortcomings including lack of support for policy routing and VRF, essential features such as VLANs and bonds, multipath routes, MTU checks, encapsulations, etc. Adding support for this feature list essentially amounts to duplicating Linux.

A lot of work has been put into making Linux a scalable NOS complete with an in-kernel ASIC driver using switchdev hooks in the kernel (Figure 3). With this approach software and hardware forwarding use the same data and the same model. The control plane uses standard Linux tools to configure networking and manage the forwarding tables. The kernel is updated to send notifications that the ASIC driver can use to program the hardware as well deny a configuration change (with proper error messages back to the user) if it is something not supported by the hardware. This design allows consistency between servers and switches where the same open source

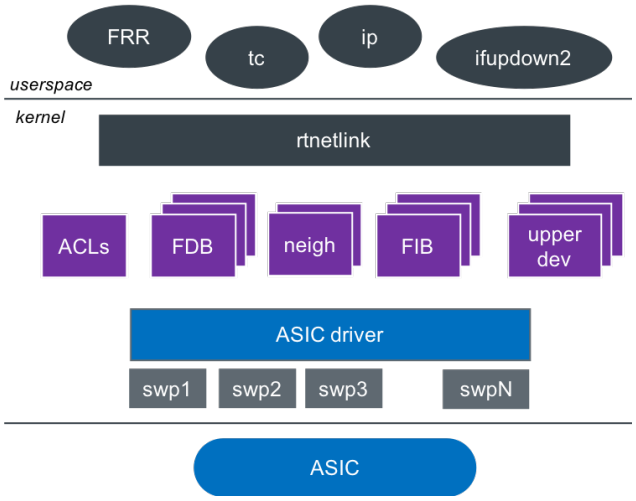

Figure 3. Linux networking with switchdev.

ecosystem works everywhere - from devOps tools such as ansible to automate bring up to route management via FRR and iproute2 to monitoring via net-snmp, sysdig and collectd. In short, the introduction of and support for a hardware offload does not change the operational model.

The same can hold true for forwarding via XDP. In a way XDP is really just a "software offload" with the forwarding decision being made in the driver. If programs running in XDP context had helpers to access kernel tables, then bpf programs would rely less on reimplementing the forwarding layer – both protocols and data management, and it would enable better coordination between XDP and the full networking stack. For example, XDP handles fast path forwarding and the full Linux stack serves as a slow path for exception cases such as fragmentation, broadcast and multicast, fdb aging, neighbor resolution, etc. handling a number of the limitations noted by Miano et al [9].

The end result is a design that allows XDP solutions to better align with Linux and its ecosystem and enables consistency in management, monitoring and troubleshooting across Linux deployments, from software only forwarding, to fast path forwarding with XDP, to hardware offload via switchdev or true pure Linux NOS solutions (e.g., Cumulus Linux). Such consistency and standardization in operational models is essential for a viable support organization.

As an example a BPF helper was recently added to the Linux kernel allowing XDP programs to access the kernel's FIB tables and serves as a primary example of how this can be done for generic forwarding without sacrificing much of the performance gains of XDP.[6]

However, the FIB helper is just the beginning. Alone, it serves a very limited use case: packets forwarded at layer 3 from one network port to another without VLANs, LAGs, or other modern networking features such as ACLs, priority and traffic shaping. Those missing features are essential to making forwarding in XDP really useful.

The next section discusses key elements of a forwarding pipeline. This is followed by a discussion of how the pipeline elements fit into the current Linux networking stack and then a discussion of what is needed for XDP to leverage the kernel tables.

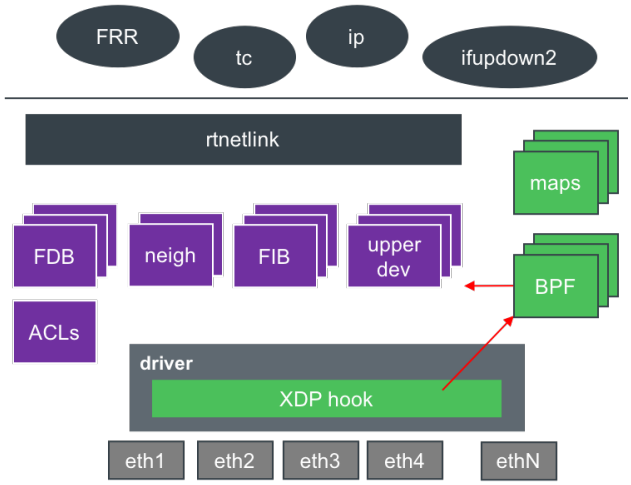

Figure 4. Forwarding with XDP and kernel tables.

Forwarding Pipeline

This section discusses typical elements in a forwarding pipeline (Figure 5). A general packet pipeline has a never ending feature list, and each of those features has an impact on performance. This paper aims at gaining uniformity between the XDP and general Linux forwarding model, something that will evolve over time. The feature list discussed here is kept to the more essential elements: features such VLANs and bonds, ACLs and filters at various attach points, forwarding lookups and related processing (fdb and neighbor entries), and packet scheduling.

Interfaces and Network Features

Physical ports are often used as trunks, carrying traffic for any number of VLANs. Ports can also have other networking features built on top such as LAGs, and the

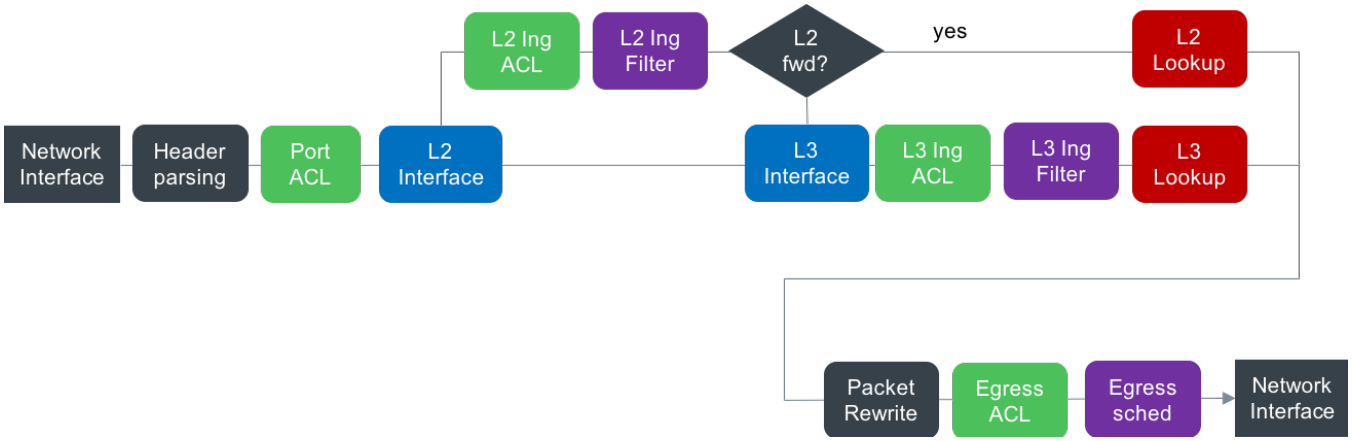

Figure 5. Example forwarding pipeline.

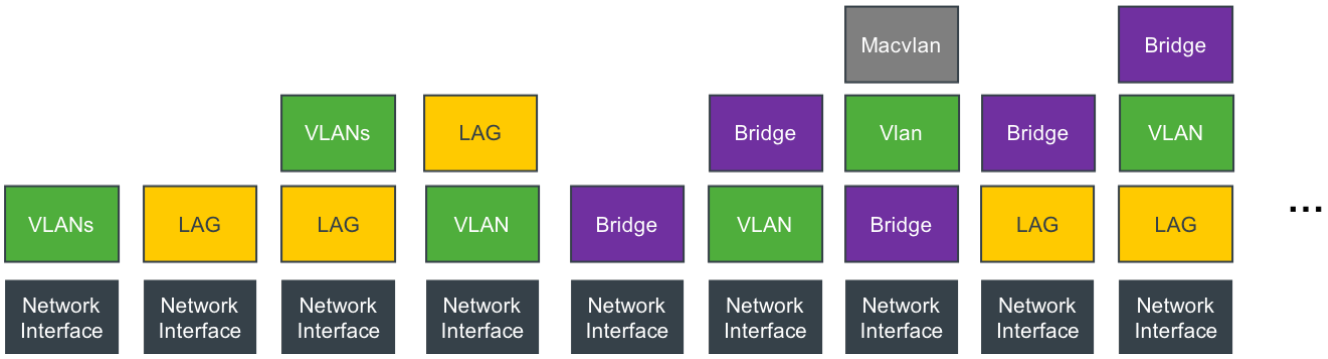

Figure 6. Feature stacking on a port.

features can be layered in any number of combinations (Figure 6). For example, a port can be associated with a LAG, and the LAG can be a bridge port member. Or the port can support traffic for a VLAN and the VLAN "object" is associated with a LAG - or vice versa, a VLAN on a LAG on port and the VLAN is the bridge port member. When processing a packet received on a port, the forwarding program needs to convert the ingress {port, VLAN} pair into other ids – such as a LAG id, a bridge port member for L2 forwarding, or a router interface for L3 forwarding.

In addition to ingress, these features also apply on the egress, after the forwarding lookup. This means the egress reference returned from the lookup eventually needs to be converted to an actual NIC port. For example, if the FIB lookup returns an egress device that is a VLAN on a bond, then in addition to having the VLAN header added to the packet the forwarding program needs to be able to resolve the VLAN to its lower device, a bond, and then have the bond logic select the egress port to use for that packet.

ACLs and Filters

As illustrated in Figure 5, a typical packet processing pipeline allows ACLs and filters to be applied at multiple attach points and based on different object references. For example, an ACL or a filter can be attached to the ingress port itself, an upper entity such as a LAG, the bridge port member or router interface, and similar layers on the egress side.

The ACL can be a simple allow / deny list based on network or ethernet addresses. A filter can be added for policing or to determine a packet priority based on the VLAN header, any priority remapping, or traffic classification.

Forwarding Lookup

The end goal is to redirect the packet from one physical port to another, by either an L2 or L3 forwarding lookup. If the packet resolves to a bridge port member, a lookup is done in the bridge's FDB based on the port index (or LAG id), VLAN id and destination mac. If the packet resolves to a router interface, then a FIB lookup is based on the L3 interface and any VRF it is associated with. The result of the lookup is an egress interface (which may also be a higher level device). For L3, the lookup can require the nexthop address to be resolved; for L2, MAC learning may be enabled, so if this is a new source mac for the port, it gets added to the FDB.

Packet Rewrite

Once the egress path and new destination mac address is known, the ethernet header is updated, potentially adding or removing an 802.1Q header before redirecting the packet to the egress port.

Packet Scheduling

At egress a packet scheduler and queueing can be used, for example, to handle different bandwidths between the ingress and egress port (e.g., ingress port is 100G and egress port is 25G) or to transmit packets based on priorities.

Forwarding with Linux Kernel Stack

Figure 7 shows a simplified version of packet processing by the full Linux stack, highlighting the parts that relate to the packet pipeline discussed in the previous section.

A packet is received by the NIC, pushed to kernel memory via DMA, and the NIC driver allocates an sk_buff for it, the main data structure for packet handling in Linux. The packet is then run through the general packet loop currently named __netif_receive_skb_core.

When the loop starts, the skb device is set to the net_device representing the ingress port. The loop in the top of Figure 7 shows the potential for ingress policies (via tc or netfilter) to be applied potentially on multiple passes with the skb device changing on each pass as it represents different features. For example, on a given pass through this loop, the skb device could represent the ingress port, a VLAN device representing the VLAN id on the port, an upper device such as a bond or macVLAN, or a VLAN device on a bond. The loop ends when there are no more upper layer features or the upper device is a bridge without an SVI. If the last upper device is a bridge, the previous device represents the bridge port member, otherwise the last device is used for the FIB lookup.

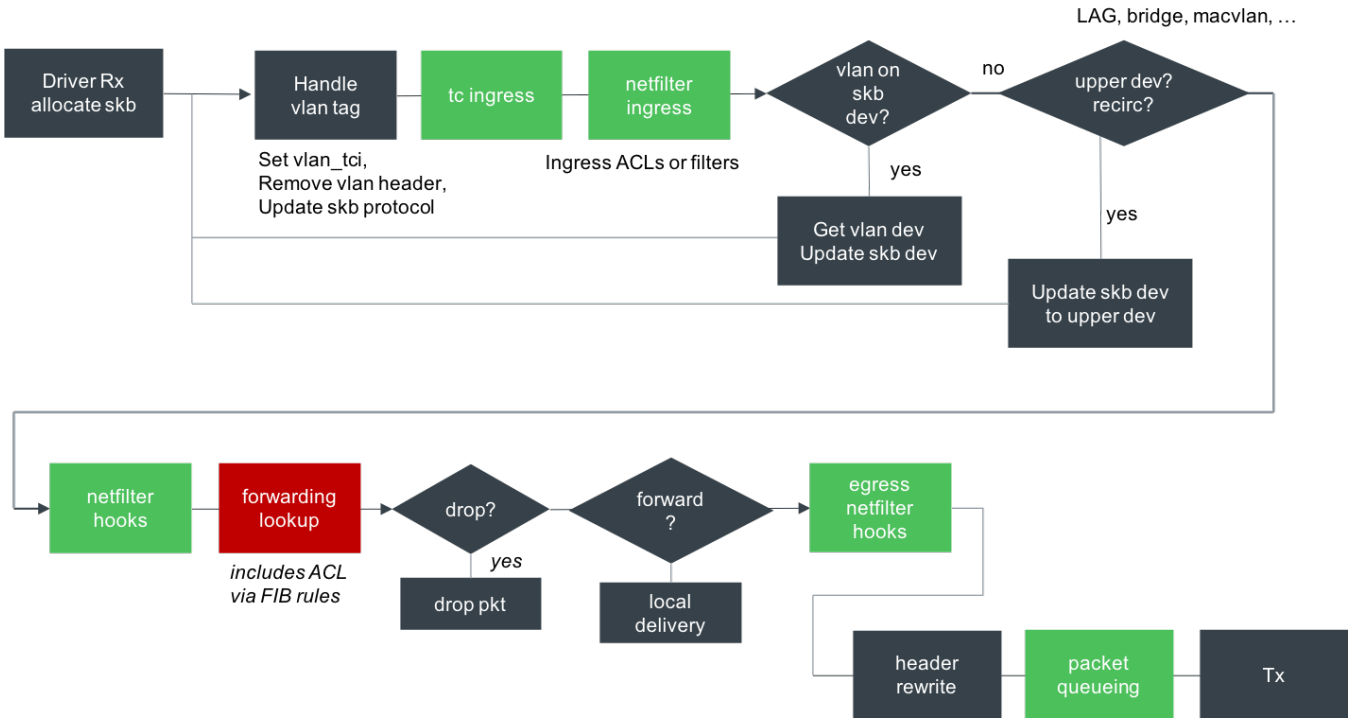

Figure 7. Packet processing loop for Linux.

From there the packet can hit one or more netfilter hooks, followed by a forwarding lookup (either in the bridge FDB or FIB), followed by more netfilter hooks.

Once the egress device is determined, the packet headers are rewritten (e.g., swapping source and destination mac addresses) and, based on features in the egress path, a VLAN header pushed on. The packet is then handed off to any queueing discipline configured for the egress device (e.g., traffic shaping, priority handling).

Forwarding with XDP and Kernel Tables

This section discusses how to map the intentions of the packet pipeline in Figure 5 with the Linux facilities in Figure 7 and considering the feature stacking of Figure 6. But first, some caveats.

To handle forwarding in XDP - and considering the potential for many layers of devices - there are a couple of design choices. The approach advocated here is to only attach bpf programs to the net_devices representing the physical ports (the bottom row in Figure 5) and then allow that program to learn about any upper layer features through kernel helpers.

An alternative solution is to add support for attaching bpf programs to all of the virtual devices built on top of the physcial port - VLANs, LAGs, etc, and then on ingress, iterate over each upper device and run any attached programs.[10] While that might work for some deployment scenarios, for general forwarding the performance gains of XDP are lost if the bridge FDB or routing FIB lookup is done more than once using each virtual device.

Forwarding in XDP on a port should really only be used when the expectation is that the majority of packets received on that port will be forwarded versus going up the networking stack for either slow path handling or local delivery. XDP programs need to determine as soon as possible, with the least number of instructions, whether the forwarding will occur in XDP or if full stack assist is needed.

Finally, the intent of the approach advocated in this paper is to allow modularity - to allow the user to pick which elements are expected to be configured and only invoke those helpers when processing a packet. For example, if tc rules are expected to be installed only on the port or bond device, then there is no need to call the helper for acl or filtering on each of the intermediary devices. Or, the program can avoid the calls completely if nothing is installed.

There are 3 main “tables” or feature sets that need to be opened for use by XDP to get this capability rolling:

1. the device table is needed to go from base port to upper devices used for forwarding and back to NIC port for egress and XDP redirect,
2. traffic control for handling ACLs and filters (policing in particular) are a must now with packet scheduling for shaping as a follow-on,
3. bridge FDB lookups and management (e.g., learning).

Other hooks like netfilter can be added over time.

Device Table

As mentioned earlier, network ports typically have a number of networking features built on top. In Linux most

networking features are implemented as net_devices. The net_device provides a means for holding configuration data unique to the instance, showing relationships between devices (features), providing a programmatic means for accessing the packet on transmit and implementing protocols and characteristics unique to the networking feature. Thus, the feature stacking depicted in Figure 6 is implemented in Linux as device stacking with APIs to find the upper and lower devices.

For example, to handle VLAN traffic through a NIC port, a VLAN net_device is created on top of the net_device representing the port. Private data for the VLAN device holds the VLAN id and protocol. The VLAN driver is responsible for popping the VLAN header on ingress, updating the skb device to the net_device representing the VLAN, and if the VLAN net_device is the egress device the driver pushes the VLAN header onto the packet before passing the packet to the lower device. (VLAN aware bridges complicate this picture a bit as net_devices are not created for each VLAN on a bridge port member.)

Another example is LAG via bonding. A bond net_device is created, and lower interfaces are enslaved to it - be it a net_device representing a nic port or an upper layer virtual device like a VLAN device. The bond driver is responsible for implementing protocols associated with the feature (e.g., LACP and 802.3ad), details such as dropping certain packets received on the backup leg(s) of a bond, and selecting the egress leg of the bond when the forwarding lookup points to the bond device.

Thus, there are a lot of details involved in implementing a networking feature, and those details should not be replicated in BPF form. Rather, the goal is to allow standard Linux tools to configure and manage VLANs, bonds, bridges, VRF, etc and create APIs that bpf helpers can use to enable fast-path forwarding in XDP.

The starting set of device lookups for XDP programs:

1. Convert ingress {port, VLAN} to LAG id if it exists,
2. Convert ingress {port, VLAN, dmac} or {LAG, VLAN, dmac} to a bridge port member (L2 forwarding),
3. Convert ingress {port, VLAN, dmac} or {LAG, VLAN, dmac} to a router interface (L3 forwarding), and
4. Convert an egress device index to an egress port noting any packet affecting data such as a VLANs.

Lookups for other intermediary devices (e.g., VLAN devices or LAG on egress) may be desired in time, but the above is a good start point.

Linux tracks the upper/lower relationships for net_devices via list heads. While traversing these lists provides a working solution that does not involve module code, it is inefficient, especially if a device has a lot of VLANs configured on it.

A more performant solution replicates the logic in Figure 7 and directly considers specific information available such as VLAN ids and net_device flags indicating if a device is a bridge port or LAG slave. Such an approach requires refactoring module code to export or provide new APIs to query about relationships. Examples for VLANs are the ability to retrieve the VLAN device given a real device and a VLAN id and protocol (ie., exporting the existing internal vlan_find_dev) and using the existing vlan_dev_real_dev. The former provides a direct way of knowing if a VLAN id is built directly on top of a net_device - a detail not tracked by the upper and lower lists. The latter allows a quick conversion from a VLAN device to its lower device (the real device).

Another more complicated example is bonding. Bonding supports several modes (e.g., active/backup, round-robin) [11] with potential impacts on ingress as well as egress. On ingress unicast packets associated with inactive slave(s) are dropped (exact delivery to be precise which means the packets are not passed to networking protocols such as IPv4 and IPv6). On egress the bonding driver implements logic to select one (or in the case of broadcast mode, all) of the enslaved ports. Thus, the bonding driver needs to be refactored to provide several APIs to allow a bpf helper to:

1. convert a slave to a bond net_device on ingress,
2. indicate incompatibility with XDP forwarding (e.g., ingress on inactive slave or broadcast mode which needs full stack assist), and
3. determine an egress path given a bond device.

The primary challenge in allowing bpf programs to query the device table is that core, builtin code (e.g., net/core/filter.c) needs access to module specific code. This is really a generic problem that applies to device-based features as well protocols such MPLS. One solution is the stubs approach used for IPv6. That option has been used in prototypes to date to make progress with VLANs and bonds as well as for MPLS support.[12]

ACL/Filtering/Scheduling via tc

As shown in Figure 5, a key networking feature for packet processing is the ability to attach ACLs and filters at multiple points in the ingress and egress paths. Linux has a number of facilities to implement a simple allow/deny ACL (tc, netfilter, FIB rules) and policing (tc and netfilter). But, only tc has the infrastructure for packet scheduling and traffic shaping. Given that, modifying tc to work in XDP context addresses more of the key features. Further, tc already aligns with switchdev and hardware offload, so converting tc to work with XDP continues the alignment with full stack, hardware offload and XDP forwarding.

No work has been to this point to get tc hooks available for xdp. Of all the changes needed for XDP forwarding, this one seems to be the most challenging.

Forwarding

A bpf helper already exists for L3 forwarding and FIB lookups with the associated neighbor entry for nexthops. While it has a few shortcomings (e.g., no support for lwtunnel encapsulations), it is sufficient as a start point for developing fast path forwarding in XDP.

What is missing is support for L2 forwarding and access to a bridge FDB. This has a few pieces:

1. compatibility of a packet with XDP forwarding (e.g., linklocal addresses),
2. FDB lookup,
3. source mac learning, and
4. SVIs (VLANs) on a bridge.

Support for broadcast and multicast can be handled by the full stack.

No work has been done to this point for L2 forwarding. A code analysis suggests that once the device table lookups exist to find the bridge port member, it should not be difficult to create wrappers to existing bridge code to perform an fdb lookup or handle source mac learning for simpler bridge deployments to get the ball rolling. But, as with any feature, the devil is in the details.

Conclusion

For XDP to be a true win for Linux as a network operating system, it needs to better integrate with the Linux stack. If XDP only encourages users to re-implement protocols and forwarding in BPF form, then XDP is really no different than DPDK.

Fast path forwarding with XDP should be able to leverage:

1. existing interface managers such as ifupdown2 to configure and manage standard networking features like VLANs, bonds, bridges and vrf;
2. existing routing suites such as FRR to manage route updates (e.g., based on link events);
3. existing tools and infrastructure for managing ACLs, filters and packet scheduling; and
4. existing tools for automation, configuration management, monitoring and troubleshooting.

All of those really come down to one thing: fast path forwarding with XDP should be visible to and managed by the existing networking APIs and rely less on custom deployments driven by opaque maps. That means more helpers are needed to allow BPF programs running in the XDP context to access data managed via the standard Linux networking APIs.

There will be limitations to what can be done with XDP, just as there are limitations with switchdev. One example is firewall, NAT'ing and routing by skb marks. Marks are held in skb metadata. With xdp forwarding there is no skb. But in the overall big picture, this is manageable for XDP just as it is with switchdev.

Yes, it does require considerable effort to refactor the existing code to work in both contexts: xdp and the current skb based packet processing. But the end result is worth the effort, enabling a consistent operational model across deployments from full stack processing, hardware offload (e.g., switchdev) or XDP fast path (software "offload").

Acronyms

API	Application Programming Interface
BPF	Berkeley Packet Filter
DDOS	Distributed Denial of Service
DPDK	Data Plane Development Kit
eBPF	Extended Berkley Packet Filter
FDB	Forwarding Database
FIB	Forwarding Information Base
FRR	FR Routing
L2	Layer 2 (OSI model)
L3	Layer 3 (OSI model)
LAG	Link Aggregation (bond, team)
NIC	Network Interface Card
OS	Operating System
RIF	Route Interface
SVI	Switch Virtual Interface
VRR	Virtual Router Redundancy
XDP	eXpress Data Path

References

1. IO Visor Project, https://www.iovisor.org/technology/xdp.

2. Gilberto Bertin, "XDP in practice: integrating XDP into our DDoS mitigation pipeline", Netdev 2.1, The Technical Conference on Linux Networking. https://netdevconf.org/2.1/papers/Gilberto_Bertin_XDP_in_practice.pdf

3. Nikita Shirokov and Ranjeeth Dasineni, "Open-sourcing Katran, a scalable network load balancer", https://code.fb.com/open-source/open-sourcing-katran-a-scalable-network-load-balancer/

4. Christina Jacob, IPv4 XDP router example, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/samples/bpf/xdp_router_ipv4_{kern,user}.c

5. David Ahern, XDP forwarding example, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/samples/bpf/xdp_fwd_{kern,user}.c.

6. Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller, "The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel", CoNEXT 2018 - International Conference on emerging Networking EXperiments and Technologies.

7. Cilium, https://cilium.io/

8. bpftool source code, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/tools/bpf/bpftool

9. Sebastiano Miano, Matteo Bertrone, Fulvio Risso, Mauricio Vasquez Bernal, and Massimo Tumolo, "Creating Complex Network Services with eBPF: Experience and Lessons Learned," Proceedings of IEEE International Conference on High Performance Switching and Routing, Bucharest, Romania, June 2018. https://sebymiano.github.io/documents/18-eBPF-experience.pdf

10. Jason Wang, XDP rx handler patch set, https://lwn.net/ml/netdev/1534129513-4845-1-git-send-email-jasowang@redhat.com/

11. Linux Foundation Wiki, https://wiki.linuxfoundation.org/networking/bonding

12. David Ahern, repository on github, https://github.com/dsahern/linux.git, bpf/kernel-tables-wip branch.

Author Biography

David Ahern is a Principal Engineer at Cumulus Networks.