Building a Better NOS with Linux and Switchdev

David Ahern
Cumulus Networks
dsahern@gmail.com

Shrijeet Mukherjee
Google
shrijeet@gmail.com

Abstract

Whitebox switches, disaggregation, and open networking are all the rage today. The success of this movement shows network operators want better choices in hardware, software and tools for deploying, managing, and troubleshooting switches. Because of this networking operating systems have evolved over the years from all userspace architectures with highly inter-dependent processes to using more and more of the Linux kernel and its APIs.

While the choice in white box switches and "open" networking operating systems has proliferated in recent years, switching ASICs are still predominantly programmed using SDKs, where the SDKs merge against various architectures, versions and capabilities in the ASIC. Traditionally each vendor's solution has been so unique that these SDKs have needed to run in userspace where the rigor of deployment can be relaxed. The adherence to SDKs impacts the architecture of a NOS, its choices for user APIs (how the switch is configured, debugged and monitored) and by extension the performance of the control plane.

Over the past few years a lot of effort has been put into a new approach for Linux called switchdev. The switchdev model brings the natural order of Linux to switches. It enables a NOS to fully leverage the Linux kernel and its APIs with the ASIC managed like any other hardware in the system - by a driver running in the kernel.

This paper discusses the evolution of software architectures for network operating systems, discusses why the switchdev model is the logical next step in this evolution, and provides an option that ASIC vendors can leverage as a transition path from the userspace SDK model to an in-kernel driver.

Keywords
Linux, ASIC, switchdev, whitebox hardware, open networking.

Introduction

The promise of white box switches, open networking and the associated disaggregation movement is to provide better hardware and software choices to users in building their networks and to enable operational efficiencies through automation. While choice of hardware has improved dramatically in recent years (Cumulus Linux, for example, supports over 75 models from 10 vendors [1]), the software movement has not evolved as quickly. Yes, there are a lot of network operating systems, networking vendors and OS frameworks, many of them claiming to be 'open', but choice in software for switches still largely refers to a silo and not choices in individual components as expected by solutions built on a Linux OS. Though most networking operating systems are based on Linux, with a few exceptions Linux is mainly limited to managing memory, process scheduling and other "housekeeping" tasks with the networking functionality implemented in a non-standard way with non-standard interfaces.

NOS architectures are largely influenced by ASIC vendors to highlight their capability which is manifested in their SDKs. The SDK requirement tends to create a NOS with highly interdependent components since anything related to configuring and monitoring the switch's front panel ports has to go through the SDK. The end result is custom CLIs, custom implementations of otherwise standardized networking protocols, and NOS dependent steps for managing everything from physical port settings such as speed and duplex, port breakouts (e.g., splitting a 40G port into 4 10G subports), and reading port module information (e.g., eeproms for SFP+, QSFP) to network protocols (vxlan, bgp, etc) and forwarding information.

In spite of a plethora of automation frameworks that are natively compatible with Linux (Ansible, Puppet, etc) the first step for a network operator to adopt a new NOS invariably involves porting the framework to that NOS. To fully realize automation requires, to some degree, standardization in the backend APIs used by those tools, and the standardization needs to consider the full context of today's data centers which are composed of racks of servers and hosts, several layers of switches, firewalls, load balancers, networked storage, etc. all working in concert to provide services. Since each of one of those nodes is most likely running some variant of a Linux OS, it is natural to envision a data center run by Linux, using common tools to administer, monitor and troubleshoot switches, servers, load balancers, firewalls, NAS, on and on. In such an environment the operational tools would be able to rely on standard Linux APIs rather than wrappers that attempt to accommodate the various closed CLIs which ultimately fail to provide the feature depth and feature velocity needed in today's data center environment.

As FRR, GoBGP, Bird and other routing suites have shown, having Linux be the reference model for the forwarding dataplane and link state management enables innovation AND standardization. This is a trend we see growing in scope over time. The next step in this evolution is to simplify the NOS with an in-kernel driver for the ASIC. It makes the ASIC programming consistent across Linux distributions and removes the need for a userspace ecosystem designed around pushing network configuration data to or retrieving that data from a userspace ASIC driver.

Before diving into network operating systems, let's do a brief review of Linux networking and its APIs from the server and host perspective.

Linux Networking

There are two sets of networking APIs for Linux: one set is for using the networking stack to communicate with a remote peer, and the other is used to configure and monitor the networking stack. This paper focuses on the latter. How applications that want to use the networking stack to communicate is outside the scope of this paper; the important point here is that for these applications to function the kernel needs to be configured with networking data on how to reach the peer. For example, for a BGP process to communicate with a peer using the Linux kernel's socket APIs (set 1), the kernel needs to be configured (set 2) with the network routing information needed to reach that peer.

Figure 1 shows the Linux networking model from a configuration and monitoring perspective. The primary APIs for modern networking in Linux are netlink[1] and ethtool. Netlink is used for most networking features with ethtool used for link specific features such as speed and duplex and typical NIC offload features such checksum and segmentation.

A core object of the networking stack is a kernel network device, or netdev for short (eth1 - ethN in Figure 1). The netdev serves as the anchor for networking addresses, routes, ACLs, filters, traffic classification and shaping, and programmability using ebpf. Furthermore, most networking concepts (bridging, bonds / LAGs, VRF, vxlan) involve a netdev for control and configuration, with a kernel driver for the netdev type.

During boot, the kernel probes for hardware and loads associated drivers. In the case of networking, the drivers are expected to determine the number of network ports on the device and create a netdev for each port. In addition, the NIC driver manages the view of the carrier state for each port and handles the receive and transmit of packets through the port. This port identification and netdev creation typically happens before userspace starts, simplifying the boot order.

Once the netdevs for each port are created, interface managers such as ifupdown2 or systemd-networkd can build upper layer features such as connecting ports to a bridge domain, a LAG (bond) or a VRF, create vlans and apply network addresses and static routes.

[1] The older ioctl API works for some basic capabilities, but a number of networking features especially those expected for a NOS cannot be configured and viewed using the ioctl API.

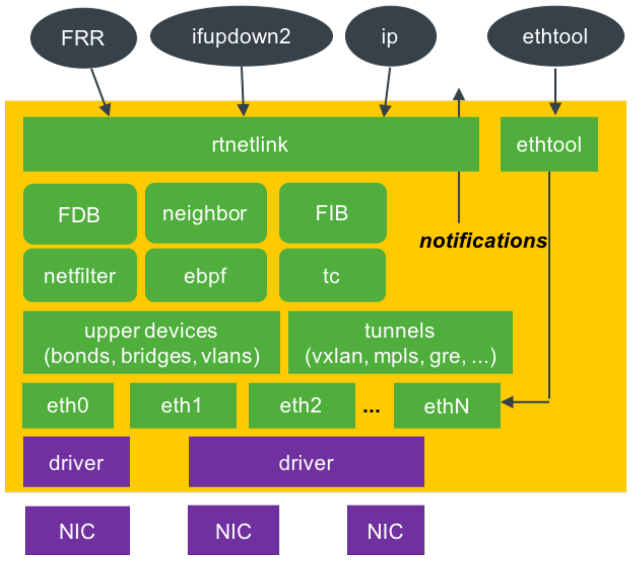

Figure 1. Linux networking stack.

One of the key features of Linux networking is change notifications: when networking configuration is applied, the kernel generates notifications to userspace. Userspace applications can register to receive the notifications via netlink allowing them to see changes to addresses, routes, features such VRF, vxlan, bridges and bonds, and link state changes (admin up/down or carrier up/down). By processing the messages and building a cache for different objects, it is possible for a userspace process to track the configuration and state of the Linux networking stack.

With this very quick overview of Linux networking, let's look at network operating systems and see how they have changed over the years, adopting more and more of the Linux model.

Switch ASICs and SDKs

Since the forwarding plane of switches and routers are complete SOC's that operate as a peripheral, ASIC vendors typically have large and growing codebases that include everything from the initialization firmware to memory controller configuration and everything in between, presented as a complete SDK.

All commands for programming the ASIC (from forwarding information to port settings) have to be funneled through the SDK (Figure 2). Similarly, any commands for monitoring traffic or gathering statistics needs to go through the SDK. In addition, the SDK will typically have to spawn threads to poll the port states (e.g., carrier up/down) and often even control plane traffic (packets) is funneled through the SDK.

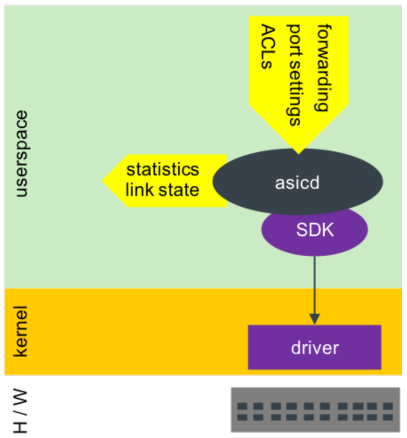

Figure 2. SDK model and userspace driver.

A kernel side driver is still needed to handle the PCI interface, but it tends to be a minimal driver handling interrupts from the device and moving blobs of data between the device and the SDK. All of the business logic is contained in the SDK.

This model means a NOS using commodity ASICs typically has one userspace process (asicd in Figure 2) that links with the SDK and becomes a key part of the NOS architecture. Over time the SDK gained "features" and became the networking portion of the NOS, while software vendors concentrated on how to call into the SDK, wrapping the interface around the SDK and optimizing code paths.

Because of this design every NOS vendor goes through the same set of steps to support an ASIC: sign the NDA to get access to the SDK (or use the "open sourced" versions released in binary format only [2]), write the ASIC daemon to call into the SDK, and then design the NOS around interactions with asicd.

Initiatives such as SAI attempt to normalize the interface between the userspace process and the SDK, so in a way there is already a push to standardize interactions with the switch ASIC. However, SAI only solves part of the problem: SAI makes it easier for a NOS to move from one switch ASIC to another, but it does not solve the standardization problem faced by network operators which is a standardized networking API across the data center.

Typical Early NOS Architecture

In the early days a NOS tended to be all in userspace, with highly inter-dependent processes using custom IPC mechanisms (Figure 3). Though Linux is typically the boot OS, none of the kernel's networking APIs are used (configuration APIs or communication APIs). Instead, everything is custom software written by the vendor for its NOS: administration through a custom CLI, custom IPC between processes to share data or query state, custom userspace networking stacks that relay packets between processes (e.g., BGP) and the SDK, and wrappers to direct networking calls to the userspace stack. In this model, even drivers for platform components such as temperature sensors, fans, and LEDs run in userspace relying on inefficient polling instead of interrupts. This closed NOS then needed custom debugging and monitoring tools, even packet captures (something ever so basic for a networking device) needed to be adapted to function.

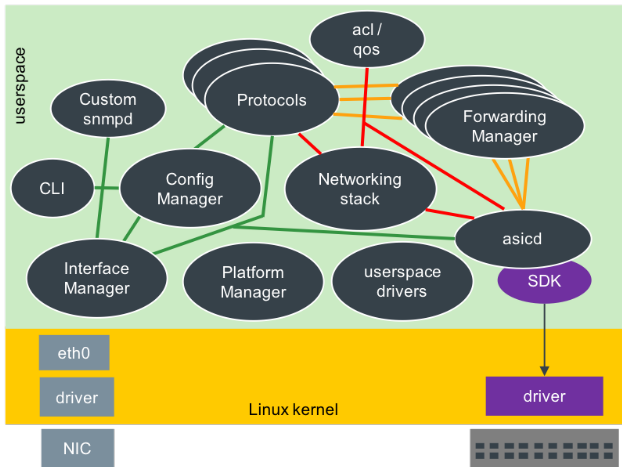

Figure 3. Early NOS architecture.

These all userspace solutions are extremely inefficient, with excessive CPU utilization from polling hardware states, high context switching between processes, and high latencies in the control plane packet paths to name a few points. Furthermore, none of the open source networking tools could be used with the NOS and none of the network based open source software would work over the front panel ports. But, this model worked with the userspace SDK and aligned with the NOS vendor's concern about GPL issues with Linux and the lack of NOS features and scalability with the Linux networking stack. At the time (early to mid-2000's), these were arguably valid concerns.

As time progressed, Linux began to dominate as the OS on servers in the data center. This standardization on Linux as the OS, meant customers became accustomed to installing Linux based software on servers in the data center. A natural extension then is to be able to install the same agents on the switches for example or to leverage open source or 3rd party S/W since, after all, the switches run Linux.

The first reaction by the NOS vendor was to release their own SDK that customers could use to link their applications to the mother ship. This meant open source software and 3rd party agents, for example, needed to be recompiled or worse re-coded against the vendor's SDK. This was tried many times and was always a resounding failure. It was time consuming, fraught with problems and in the end only existed to further the vendor lock-in. Customers responded with a clear "no way, fix your OS."

Toe in the Water with the Kernel Stack

As mentioned earlier for networking tools or network based programs to function, the Linux kernel needs to be populated with netdevices, network addresses, routes and features such as VRF, bridges and LAGs. But this is a Pandora's box: if a NOS vendor does not believe Linux has the feature set or scalability how does the vendor decide what to put into the kernel? Putting only partial information can be confusing at best (incomplete route dumps, packet counters or packet traces) and lead to wrong connections at worst (e.g., an agent intends to connect to customer A through VRF red but instead connects to customer B through VRF blue). Further, what if a customer wants to inject routes based on its own agent talking to its controller or control interface state using Linux tools such as the iproute2 package? Now the NOS needs support for listening to kernel notifications and understanding the messages. But again where is the boundary on what is supported?

To give an appearance of supporting Linux APIs a NOS can create kernel netdevices for front panel ports and features such as port channels and bridges and even populate some information into the kernel so that control plane communications work, but it is ad hoc and far from complete. For example, some commands will work, such as setting a link up or down, but others fail, such as creating a vxlan endpoint or bond for a LAG or port channel. Worst, standard networking features like VRF might appear to work (e.g., for an OS using namespaces for VRF 'ip netns add myvrf' creates a network namespace), but the desired configuration is not understood by the proprietary userspace blobs.

When looking at the antiquated kernel versions or lack of kernel modules for the feature (e.g., modinfo vxlan), it is easy to see that the NOS does not truly understand the Linux way and really relies on its custom CLI for configuration making its "Linux support" for appearances. In that case it is user beware in knowing what actions can be controlled by the automation tools of choice and which actions need to go through the CLI or vendor wrappers. Maybe the most important thing of all is that while you are now interacting with Linux kernel objects, the behavior and model behind those objects are not that of the kernel stack which will be a surprise at some point.

This path of 'kind of, sort of' supporting Linux networking allows a marketing or sales angle to check a box, but anyone trying to use automation tools will quickly find out it is for show only.

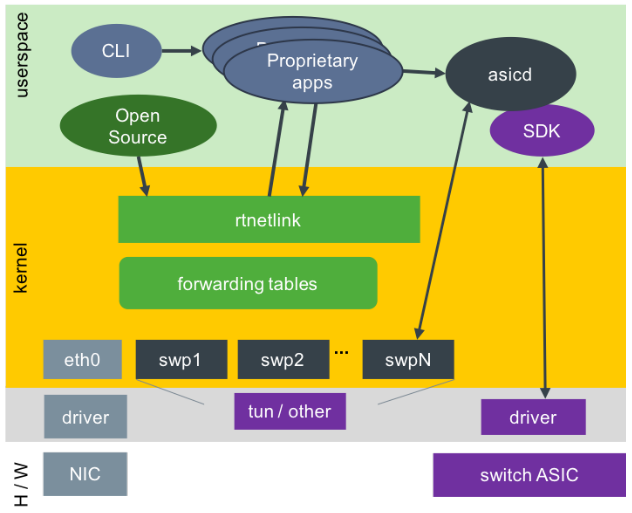

Figure 4. NOS with some kernel data.

So, let's move to the next step – buying into the Linux APIs but still programming the hardware by the SDKs.

Kernel Networking with SDKs

Buying into the Linux model the first step is to create netdevs for each front panel port. There are two standard options:

1. Use the tun/tap driver to create the netdevs and have the asic daemon relay packets between the SDK and the Linux kernel. This is certainly an easy and flexible approach for getting started, but it does incur a significant overhead and additional latency due to the trampolining of packets.
2. Most ASIC vendors now provide kernel drivers to create netdevs and associate them with the front panel ports, evidence that the ASIC vendors are aware of this trend to move to the Linux kernel networking stack. In this model, the kernel driver will disperse packets received from the ASIC to the kernel netdevs based on meta data indicating the port the packet was received. This method avoids the userspace trampoline and is much more efficient.

With the port netdevs in place, any interface manager that speaks the Linux netlink APIs (e.g., ifupdown2 or systemd-networkd) can be used to create networking features (bridges, bonds, vlans, vrf, etc), configure network addresses on the devices and inject static routes. All of this is done based on the "Linux Way" and works the same on any Linux OS - be it a switch, a server or a host.

But, the "Linux Way" only means the kernel has been populated with data. For switches, we still need to program the ASIC with the networking configuration. To that end, a userspace process can listen to the kernel notifications, keep a cache of objects to track the kernel state, and then feed that data to the ASIC daemon to program the hardware. Or, for simplicity, the ASIC daemon itself can handle the kernel notifications and invoke the SDK APIs to keep the hardware in sync (Figure 5).

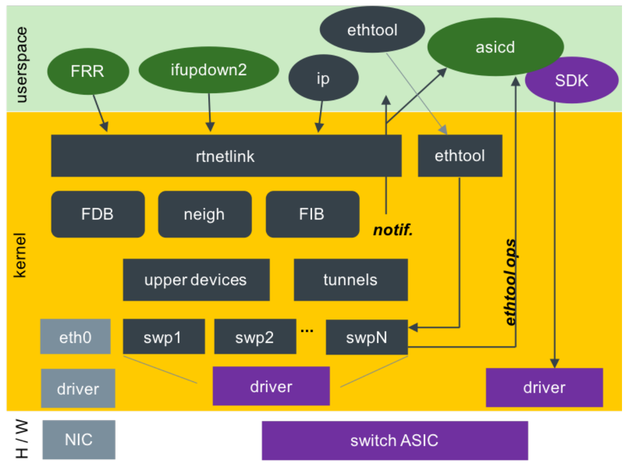

Figure 5. Linux networking stack with the SDK.

To date, vendor-based kernel network devices do not handle ethtool operations for link configuration. Because of the userspace SDK, those operations need to be bounced through the userpsace asic driver which is unique to a NOS. This does not involve customizations of the kernel itself, only providing hooks that use an RPC to send the commands to userspace.

The end result of this model (full buy in to the kernel APIs) is that the kernel tables hold the networking configuration and state with the ASIC programmed via snooping. That means any software that understands the Linux APIs can be used to configure links, administer features, inject routes, and monitor and troubleshoot the switch. Thus, this model is a huge improvement over the all-userspace, black box model from the early days. It achieves the Linux API standardization trend and truly provides an open networking solution for customers. Any software that understands Linux APIs can be used based on the network operator's choice.

With that achievement, how can it be improved after all networking and Linux are always evolving, always improving? Each step forward resolves problems of the past and exposes the next steps for improvement and simplicity.

switchdev Architecture

Switchdev[3] brings the natural Linux order to network switches: with switchdev the ASIC driver is a kernel module that is loaded at boot, identifies the number of ports in the specific version of the switch ASIC and creates netdevs for each front panel port. Further, the driver knows how to push networking configuration down to the ASIC in direct response to commands from userspace. When a process makes a change to the networking configuration (e.g, adds a network address, creates a VLAN, inserts a route), the switchdev driver is notified via in-kernel APIs and programs the hardware – a typical top-down process from user to hardware with a single source of truth for networking configuration and state (the kernel tables) and direct control that configured networking features are supported by the hardware and the hardware has available resources (i.e., how an OS should manage devices using a device driver framework).

Switchdev APIs

Linux has a number of in-kernel and user APIs that a switchdev driver can leverage to learn about changes to the networking configuration and program the hardware:

1. **Switchdev operations**. Switchdev_ops allow drivers to learn of port level changes.
2. **Notifiers**. Similar to userspace notifications, in-kernel notifiers allow drivers to learn about changes to devices (e.g., LAGs, bridging, VRF), FIB entries, neighbors, sysctl settings (eg., multipath hash policy).
3. **Devlink API**. The devlink API provides a means to do device level changes and dumps. For example, the devlink API can be used to configure limits to ASIC table sizes and split or unsplit ports (e.g., split 40G port into 4 10G ports). The devlink API also provides a means to extract device information such as current resource utilization and driver tables.
4. **ethtool API**. The ethtool API is currently the standard method for discovering and configuring speed and duplex options as well as dumping eeprom for a port or plugin module (e.g., SFP+, QSFP). This API is typically used for physical layer management.

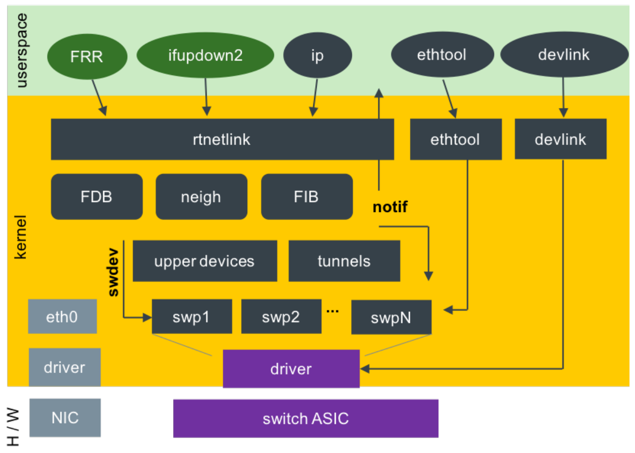

Figure 6. switchdev architecture.

As a side note, the devlink API can be leveraged outside of an ASIC driver. As an example, a resource controller was added to netdevsim showing how a driver can register for in-kernel FIB notifiers and manage (e.g., limit) FIB entries and FIB rules[4]. This is one example of how to improve error handling and resource overflow with the SDK-based solutions without going all-in on the switchdev model.

Benefits of switchdev

In general, the switchdev approach has fewer moving parts and makes for simpler, cleaner architecture. For starters, it does not rely on a userspace process to learn the networking configuration and make calls into an SDK to program the hardware. There is a single source of data – the kernel's tables – so no cache to get out of sync or double implementation of features. Furthermore, the model allows hardware state (e.g., carrier on / off) to be managed via interrupts instead of the SDK approach which relies on polling threads thus making the switch more efficient with lower CPU utilization.

The switchdev model provides a better user experience. For starters, standard and well-known APIs (netlink, devlink and ethtool) are used to configure, manage, and monitor a switch. The in-kernel driver allows the Linux APIs to be leveraged while still maintaining control over error handling as it has the ability to fail a change if it does not match with its capabilities or overflows some resource.

With switchdev, packets punted to the CPU from the switch ASIC are distributed efficiently to the kernel netdevs representing the front panel ports, not relying on a userspace trampoline required by tun based devices. From there the packets flow up to any stacked features such as bridging and bonding similar to standard Linux distributions. This makes packet captures, statistics monitoring, etc all equivalent. And, no NOS customizations are required to handle and implement ethtool based configurations. Splitting ports is standardized. Viewing port speed and eeprom data is standardized.

If switchdev is a better model and the future of Linux-based network operating systems, how do we get there? The next section proposes a model that allows vendors to migrate to the switchdev way without having to go all in at once.

Transitioning the SDK Model to switchdev

ASIC vendors are not going to spend the time to support switchdev until market forces push them. That means switch buyers and/or NOS vendors need to see the value of a switchdev based solution and put pressure on the supply chain. That leads to a chicken-and-egg problem: how do you prove switchdev is really a better, simpler, more robust, more performant solution for a preferred vendor's ASIC?

As noted in the previous section, a switchdev driver is expected to handle and implement a number of in-kernel APIs as well the devlink and ethtool APIs for userspace. These APIs can be seen as a burden on driver writers, overhead each needs to deal with on top of creating a driver to manage and program the ASIC. So, a reasonble first step is to reduce this overhead and simplify the barriers to entry (and limit the excuses of avoiding the support).

Common Layer for Switchdev

A lot of this in-kernel driver code can be abstracted into a common layer for switchdev drivers (clsw). As shown in Figure 7, clsw can manage the in-kernel events and operations handlers and maintain a cache to translate between kernel objects and ASIC objects (the belief is that in time even this mid-level object cache can be removed). clsw can also provide default handlers for netdev, switchdev, devlink and ethtool operations. In turn, clsw invokes SDK operations (e.g., create/delete/modify a VLAN, host route, RIF, nexthop, etc) via backend ops that are effectively a HAL (similar to what SAI provides). With clsw, ASIC vendors can wade into the switchdev waters by starting first with the ability to run their SDK in kernel mode and provide support for the clsw backend ops.

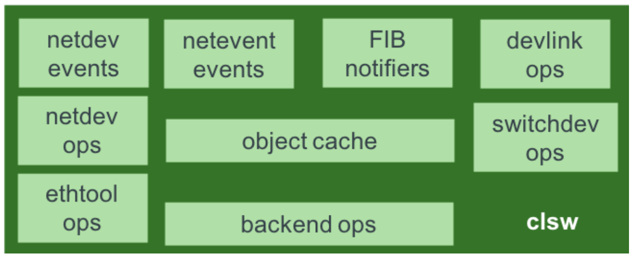

Figure 7. Common Layer for switchdev drivers.

The clsw source code is available on-line[5]. It is still very rough around the edges and should be considered proof of concept only. A lot of the code is based on or modeled after Mellanox's mlxsw driver [6].

Using clsw and a kernel-mode SDK, the NOS architecture simplifies to Figure 8. No userspace trampolines are required, one source for networking configuration and a top-down flow for configuring the switch.

To be clear, the clsw based architecture should be viewed as a stepping stone for full, in-kernel support of switch ASICs, a path for a reasonable transition. In time, it should result in a proper driver that handles the various operations handlers directly, though there still is a lot of duplication in the notifiers and object cache to warrant a common layer.

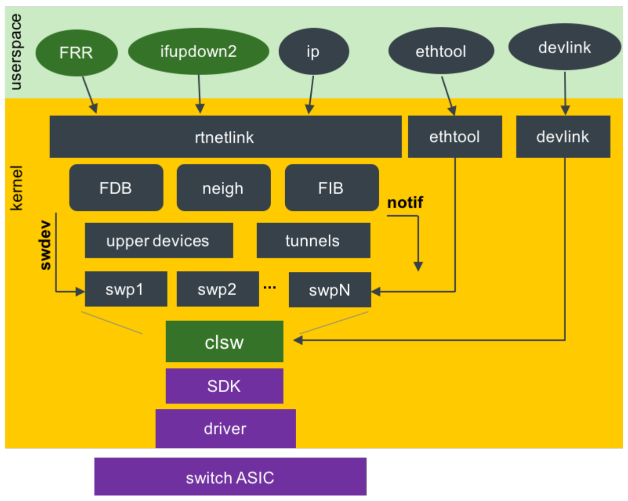

Figure 8. Using clsw to run SDKs as a switchdev driver.

The Elephant in the Room.

At this point Linux purists are probably screaming "WAIT! This amounts to putting a proprietary blob in the kernel." In short, yes, it does. But, ideological stances do not move the needle on Linux support for hardware; market forces do. Switchdev has been in development for over 3 years now, and yet there is only 1 vendor writing a driver for a full-featured switching and routing ASIC. The other vendors either have not seen the light or have no motivation (yet) for making the move. The intent of clsw and a kernel-mode SDK is to facilitate the transition.

There are many examples (e.g., graphics drivers and wifi drivers with NDISwrapper) where an intermediate step such as this one has been necessary to convince hardware vendors to write native drivers. Again, consumers cause a change. For consumers to care about something like an in-kernel driver, they need to see and experience the benefits first hand – the improved operational efficiencies, the improved boot times, etc.

Example Deployment with clsw

To illustrate clsw, a prototype was created using an unnamed vendor's SDK. This SDK has an option to run in kernel mode and supports SAI as an external interface, so SAI is used as the backend operations between clsw and the SDK. SAI is an inefficient API for the kernel, but it does encapsulate the intent of a generic API between an SDK and clsw and allows the prototype to focus on development of clsw rather than writing yet another frontend to an SDK.

To illustrate the advantages of switchdev, we look at two benchmarks:

1. Time from reboot until route is programmed in the ASIC.
2. Time to program 50,000 routes.

Figure 9 shows the simplified topology used for the tests. s1 in the figure is the unit under test and what varies. It is one of 3 switch models:

- **SDK**: Running a userspace daemon linked with an SDK. The daemon listens for kernel notifications and programs the ASIC. The CPU in this switch is a quad core Atom C2538@2.41GHz.
- **swdev**: Running the 4.18.0-rc3 kernel with a native switchdev driver. The CPU in this switch is a dual core Celeron 1047UE@1.40GHz
- **clsw-sdk:** Running the 4.18.0-rc3 kernel with clsw and a kernel mode SDK. The CPU in this switch is a quad core Atom C2538@2.41GHz, same as the sdk switch, but the platform is from a different hardware vendor.

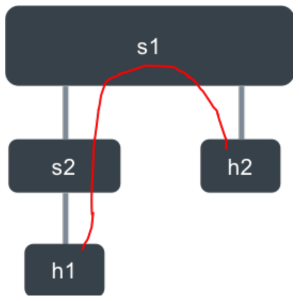

Figure 9. Topology used for testing.

The numbers presented in this section should be taken as ***rough ballpark comparisons only*** as there are a number of variables different between the switch and OS setups used for s1, most notably different CPUs, different kernel versions and different ASICs. The intent of the metric comparison is to illustrate the trend and expectation that the switchdev approach will be much faster.

Boot Time

Fast boot of a switch is one argument against the antiquated notion of an ISSU, so reducing the time it takes to do a reboot and get routes programmed into the ASIC is a huge win against complexity for a feature that rarely works correctly and consumes an inordinate amount of engineering time.

For this test we use kexec to do a fast reboot of the switch and measure the number of missed pings sent at 1-second intervals between h1 and h2. The numbered of misses illustrates the downtime of a switch during a reboot. This time span covers kexec switching from old kernel to new, initialization of hardware with new kernel, starting userspace, applying networking configuration via the ifupdown2 interface manager, starting FRR, bgpd on s1 peering with s2, learning a route from s2, injecting the route into the kernel, and it getting programmed into hardware.

Table 1. Missed pings during kexec reboot.

Mode	MIssed pings
sdk	72
swdev	28
clsw-sdk	27

The numbers shown in Table 1 are missed ping responses on h1. The switchdev approach is much faster most notably because there are fewer moving, coordinating pieces. The flow at boot time is simpler as there is no waiting for a userspace process to start or SDK to initialize (which means starting polling threads, opening sockets with the kernel, etc) before anything network related is started.

The faster time for the clsw-sdk over the pure switchdev model is most likely due to CPU differences (2.4GHz vs 1.4 GHz) with more cores (4 versus 2). Comparing the console message profiles side by side, the clsw-sdk switch was always a few seconds ahead of the swdev switch even before getting to the ASIC driver.

Note that other SDK boot times on other switch models is faster. In one case that is much more comparable to the switchdev model, the SDK approach was able to boot and program the route in about 40 seconds making the switchdev approach only twice as fast. Overall boot times are highly OS dependent and no attempt has been made to optimize the bootup (yet), so the numbers shown in Table 1 will come down over time and more importantly do convey the general trend that an in-kernel driver will always initialize the hardware and be ready to program routes faster than relying on userspace based driver.

50k Routes

Another huge benchmark for switches and routers is convergence time following a link event. For example, a link comes up, bgp establishes a connection with a neighbor, exchanges routes, and new entries are pushed to the hardware.

For simplicity this test uses the 'ip' command from the iproute2 package with a batch file of single nexthop routes to simulate the learning of a batch of routes from a neighbor. The file contains the first 50,000 routes from a full IPv4 route table with the nexthop of each route set to s2. This set of routes is then followed by a single route that enables connectivity of h1 to h2.

Unfortunately, in this test the clsw-sdk case started failing after ~4300 routes due to profile limits and a few other issues. Time ran out trying to make changes to push that number higher. Further, low route numbers were getting programmed too fast making the measurement error a significant part of the overall time. That said the switchdev solution does appear to be consistently faster than the userspace SDK approach, and typically faster by a factor of 2.

There is plenty of room for optimization with both the userspace sdk and switchdev drivers, but we believe the general trend will always hold – an in-kernel driver responding to route injections as they happen will be able to program the routes into hardware faster than relying on a userspace SDK which is subject to context switches and process scheduling.

Future Work

The switchdev approach shows a lot of promise for simplifying network operating systems and enabling standardized APIs. However, there is still much work to be done for feature parity with most widely deployed network operating systems.

The proposed common layer for switchdev drivers, clsw, is very early in its lifecycle and still needs a lot of work to mature into production ready code. In addition, a more efficient backend API is needed than the SAI API used for the demonstration.

Conclusion

The trend in network operating systems is for the Linux kernel to take a more prominent role. The Linux networking stack has improved dramatically over the past few years, adding features, improving speed, and improving resource utilization with more features on the horizon that better align with switch ASIC and hardware offloads. This effort is closing the gap and removing excuses for NOS vendors to avoid the kernel model.

With switchdev, network operating systems become simpler, faster, and allow improved operational efficiencies via consistent APIs and automation. It is time for more ASIC vendors to join the switchdev movement and make the ASIC programming part of an in-kernel driver. To that end, clsw provides a common a layer alleviating some of the overhead of moving from SDKs to a proper kernel driver.

To be clear, customers are not clamoring for more Linux, nor are they specifically asking for switchdev. What they are asking for is what the architecture enables -- operational efficiencies, automation, consistent implementation of networking features across devices in the data center, and the ability to leverage the Linux and open source ecosystem.

Acronyms

ASIC	Application-Specific Integrated Circuit
CLSW	Common Layer for Switchdev
HAL	Hardware Abstraction Layer
ISSU	In-Service Software Upgrade

LAG	Link Aggregation
NOS	Network Operating System
OS	Operating System
SDK	Software Development Kit
SAI	Switch Abstraction Interface
VRF	Virtual Routing and Forwarding

References

[1] Cumulus Networks, Hardware compatibility list, https://cumulusnetworks.com/products/hardware-compatibility-list/

[2] Open Network Switch Library, https://github.com/Broadcom-Switch/OpenNSL

[3] Switchdev kernel documentation, https://www.kernel.org/doc/Documentation/networking/switchdev.txt

[4] netdevsim: Add simple FIB resource controller via devlink https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/drivers/net/netdevsim?id=37923ed6b8cea94d7d76038e2f72c57a0b45daab

[5] Common Layer for switchdev source code https://github.com/CumulusNetworks/net-next clsw

[6]Mellanox switchdev driver: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/drivers/net/ethernet/mellanox/mlxsw