Breaking Open Linux Switching Drivers

Andy Gospodarek

Cumulus Networks, Inc
Mountain View, California, USA
gospo@cumulusnetworks.com

Abstract

Ever wonder what the next generation of network forwarding devices will look like? Some will simply switch and route traffic in a manner that is similar to what is seen now, others may communicate with an external controller to determine forwarding options. Yet another class of systems will be created and their use will not be known as they will stay hidden inside datacenters with features completely unknown to the users whose data transits the network. It may be difficult to predict exactly which features will be included, but one feature that is an almost certainty is they will be running a Linux kernel. While many of the devices in use today are running Linux there is little collaboration around this effort as the forwarding ASICs are only enabled by the use of closed-source SDK and out-of-tree kernel drivers and modules provided by the hardware vendor. This paper will provide some historical background on the current situation and propose a way to enable rapid, zero-cost development on datacenter-grade ASICs by starting the process of *Breaking Open* these SDKs to provide an infrastructure that may allow for inclusion in the upstream Linux kernel.

Introduction

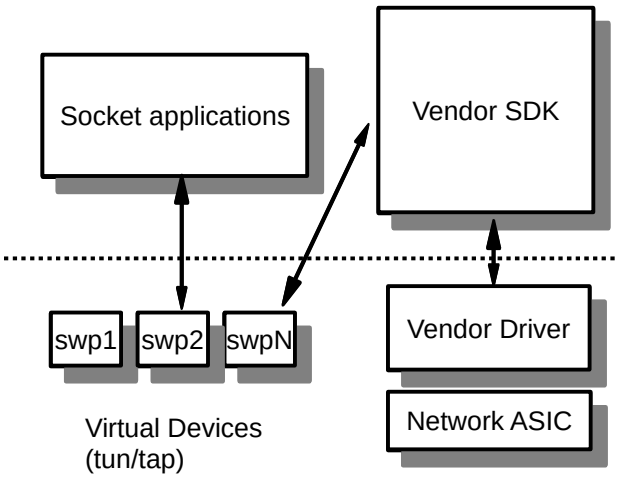
Linux has been the operating system of choice for hardware switches and routers for the better part of the last two decades. Most users did not know this as direct access to the operating system and hardware were hidden behind a shiny (or dull) user-interface. Community projects like (OpenWRT/DD-WRT/etc) provided users the first chance to use standard FOSS networking tools to configure and manage devices and products like Cumulus Linux and projects like Open Route Cache have taken this a step further to support enterprise and data-center grade top of rack switches using open-source tools and infrastructure -- though today they still rely on out-of-tree kernel drivers and a vendor-licensed SDK.

The goal of this paper is to present a viable alternative for how current vendor switching and routing hardware can be made significantly more usable by kernel and application developers by moving away from the current model and towards a model with zero software integration cost. Today, most Linux users of datacenter hardware currently interact with network devices that are presented as tun/tap devices and use of tree kernel drivers to access hardware. This combination does not allow access to hardware information or configuration (how are the ethtool_ops for tun/tap working?) and it currently provides no ability to leverage the recently merged offload (switching and routing) infrastructure that has recently made it into the upstream kernel. Though this paper proposes a solution that requires reliance on a vendor-licensed SDK, releasing and adding this driver to the upstream kernel is the first step towards a goal of simplified data-plane programming. This paper includes: a description of the architecture of this driver as it compares to the typical vendor implementation, the relationship and communication between this driver and the current vendor-licensed SDK, and plans for the future growth of this into a stand-alone driver with minimal (if any) SDK reliance.

How did we get here?

ASIC Vendor SDK Architecture

Despite the fact that Linux was an operating system supported by network ASIC vendors for 10+ years, little has changed in the way of architecture or methods by which systems were administered or controlled over that time period. Most ASIC vendors provide a minimal kernel driver that binds to the PCIe devices and allows a userspace SDK to read and write to the mmap'd hardware as needed. The control-plane path is shown below:



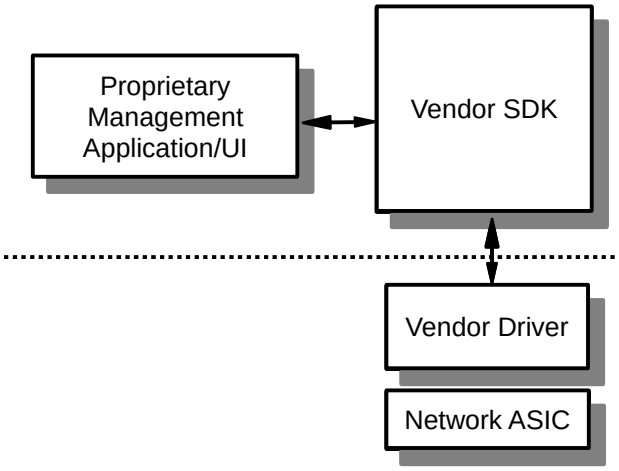
This architecture has largely remained unchanged over the last decade. There are some newer alternatives from some ASIC vendors that provide kernel modules that allow switch ports to look more like actual in-kernel netdevs, but

are designed in the same spirit as those that rely on tun/tap for control-plane traffic and do not offer assistance with data-plane programming.

System Vendor Software Architecture

Due to the fact that no Linux kernel integration exists for data-plane programming of these ASICs, system and software vendors who want to use them in their platform must write a non-trivial amount of software to provide date-plane programming to the chosen network ASIC.

Though recent efforts to develop a switching abstraction layer (SAI) for offload devices by the Open Compute Project[1] aim to make software integration easier for vendors, there is no mention of a requirement to use the Linux kernel or kernel drivers to be compliant with the specification. This means that data-plane programming can vary widely. Some system vendors may choose a model where the kernel has little to no knowledge of the forwarding hardware and a proprietary management application will communicate directly with a vendor SDK to program the data-plane.



The specific reasoning behind the apparent lack of interest in Linux kernel integration in SAI remains unknown, but the fact remains that is is not a requirement and therefore significant effort will need to be taken to integrate support for any vendor hardware that supports SAI. It is important to note that each vendor will want to have their own method for configuration and will likely wrap this configuration around a user-interface that does not revolve around the standard FOSS tools used to configure network devices on traditional Linux systems (servers, workstations, laptops, et al).

These user-interfaces are not problematic by themselves (in fact a large contingent of network administrators have spent years gaining certifications centered around the ability to control these devices). They do not offer the flexibility one gets from being able to configure systems using tools familiar to most server administrators and therefore do not enable rapid feature development.

Hardware Platform Performance and Availability

Most of the commercial networking platforms sold in the past contained low-power and lower-than-server-performance host processors. This was mainly to save cost and power, but also quite practical. The network ASIC performed most of the forwarding and little work was done by the host processor. Due to the fact that these systems were not powerful, there was minimal interest from the traditional Linux distribution vendors or from the upstream Linux kernel community. Now that we are in the middle of what some would call the rise of *Bare-Metal Switching*, more powerful systems with networking ASICs are available, used in large-scale datacenters[1], and available for purchase with or without software installed[2]. The availability of these platforms and their ability to easily boot an upstream kernel or standard Linux distributions makes developing applications on these platforms a viable option.

In-Kernel Network Offload Infrastructure

Based on efforts by developers and reviewers from multiple organizations there is some initial infrastructure[3] in the upstream Linux kernel to support network ASICs capable of offloading data-plane forwarding. Included with this infrastructure was the ability to program an emulated offload device known as *rocker*.

This infrastructure is an excellent first step on the road to full in-kernel support for hardware devices, but Linux kernel drivers for datacenter-class network ASICs that use this infrastructure will enable the development of the next-generation of forwarding devices more rapidly than support for an emulated device.

What are the next steps?

Opening of most SDKs seems unlikely

Whether speaking to someone directly who works for one of the network ASIC vendors or reading the specifications produced like those from the Open Compute Project's Networking Group[4] it becomes clear that most hardware vendors have little interest in sharing all code available in their user-space SDK with the community. There are likely a variety of reasons why this is the case, but the fact remains that re-licensing the SDK in a manner that will allow inclusion in the Linux kernel or some other FOSS package is unlikely. At least until the current Linux kernel offload infrastructure is given a chance to develop further.

This places those that share my vision for how the Linux kernel can become the abstraction layer for network ASICs in a difficult position. If we cannot convince ASIC vendors to open-source their SDKs are those with access to those SDKs forced to keep the code they develop

proprietary? Are we destined to spend the next decade like the last with no upstream code added to drive this hardware?

The case for a bifurcated driver

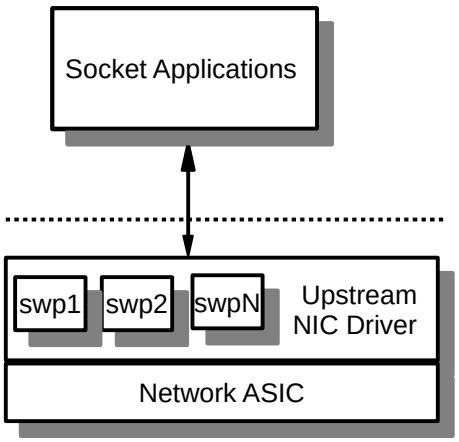
Any vendor that wants to have an upstream driver for their network ASIC must realize there are two components needed to drive upstream acceptance for any network offload device: the ability to abstract front panel ports as network devices and the ability to program the network ASIC offload the forwarding of traffic when requested.

Network Driver

This portion of their driver should be quite familiar to anyone who has written a network driver in the past. Basic functionality to make ports on a NIC appear as net devices to the Linux kernel are:

- Knowledge of platform port topology
- Capability to interact with PHY and other components needed to establish connectivity
- Ability to request interrupts and service them
- Ability to transmit and receive frames on individual ports

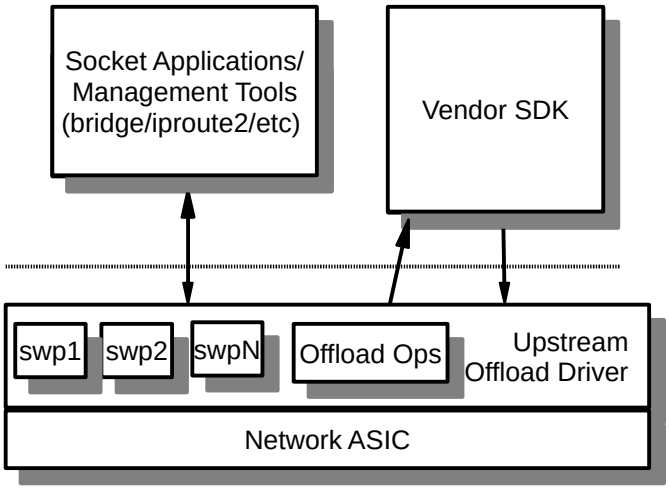
Any network ASIC vendor should be able to easily meet the above criteria without the need to rely on an SDK. This portion of the driver would not enable offload capability, but would give the vendor, users, and developers the opportunity to test and develop applications using a full FOSS stack with software-based paper forwarding. The architecture would be something like the following of no offload capabilities were enabled with the upstream Linux kernel driver and each switch port was exported as a single NIC:



Support for Offload Operations

Though the dream of a zero-cost software integration model could be lost without inclusion of the SDK in the kernel, there is hope! If one accepts that today and for the foreseeable future user-space SDKs are not going away, it is easy to envision a design where userspace and kernel-space co-operatively program hardware through the Linux kernel's offload infrastructure.

In this implementation communication between an offload compatible kernel driver and userspace would allow kernel operations to call back to userspace and perform the needed date-plane programming. Some would all this a *Trampoline Driver*. The architecture of a driver like this might look like the following:



A sample driver implementation of a call to an external application like this may look like the following:

```
static const struct net_device_ops VENDOR_swp_netdev_ops = {

        [...]
        .ndo_fdb_add                    = VENDOR_swp_fdb_add,
        [...]
};

static int VENDOR_swp_fdb_add(struct ndmsg *ndmsg,
                              struct nlattr *tb[],
                              struct net_device *dev,
                              const unsigned char *addr,
                              u16 vid,
                              u16 nlmsg_flags)
{
        struct VENDOR_msg_struct *vendor_msg;

        if (!VENDOR_prep_msg(vendor_msg, dev, addr, vid, flags))
                return -EINVAL;

        if (!VENDOR_msg_send(vendor_msg))
                return -ENODEV;

        return VENDOR_msg_ack(vendor_msg->cookie);
}
```

I do not specifically want to advocate for what type of messaging implementation is used by **`VENDOR_msg_send`**. It could be backed by a synchronous socket send/receive or ioctls. It may be backed by asynchronous genl netlink messages or hardware mailboxes. One might say this interface is an implementation detail left up to the user.

The reasons not to specifically standardize that communication channel or API used between a userspace SDK and the Linux kernel is to prevent that API from becoming *the* standard. That will serve no purpose other than to virtually guarantee that these SDKs will unlikely migrate to the Linux kernel since there is already a commoditizing interface.

References

1. Bare-Metal/White-Box Switching Market Share
Accessed Jan 2015
http://blogs.gartner.com/andrew-lerner/2014/09/02/hellowhitebox/

2. White-box platform availability based on hardware supported by ONIE
Accessed Jan 2015
http://www.opencompute.org/wiki/Networking/ONIE/HW_Status

The Case for a Trampoline Offload Driver?

The design of this proposed driver may be somewhat controversial to the upstream kernel community. While the software architecture does differ from most other pure-hardware drivers, it provides a balance that allows developers to create a fully-featured network offload software implementation for the Linux kernel backed by real hardware while still allowing vendors to hold on to userspace SDKs. Once this in-kernel offload model is proven and working well, the race will begin to see who decides to drop their closed-source SDK and *Break Open* first.

3. net: introduce generic switch devices support
Accessed Jan 2015
http://patchwork.ozlabs.org/patch/415861/

4. Open Compute Project - Networking Specs and Designs
Accessed Jan 2015
http://www.opencompute.org/wiki/Networking/SpecsAndDesigns#Switch_Abstraction_Interface