Hardware switches - the open-source approach

Jiří Pírko

Red Hat
Prague, Czech Republic
jiri@resnulli.us

Abstract

Imagine buying off the shelf switch hardware, install Fedora (or any other distribution) and configure it using standard Linux tools. This is not possible at the moment primarily because of lack of unified and consistent platforms and driver interfaces.

The current state of support for switch chips in Linux is not good. Each vendor provides userspace binary SDK (Software Development Kit) that only works with their chips. Each of this SDKs has proprietary APIs. To get switch chips properly supported there's need to introduce a new infrastructure directly into Linux kernel and to work with vendors to adopt it.

This paper presents the current effort to unify and uphold the Linux networking model across the spectrum of devices which is necessary to make Linux the cornerstone of industrial grade networking. The scope of this paper covers state of art with current implementation of standard commodity switches such as top of rack switches, small home gateway device as well as SR-IOV NIC embedded switches.

A device model and driver infrastructure will be presented for accelerating the Linux bridge, Linux router, accelerated host virtual switches and flow level offloads when supported by the hardware underneath.

Keywords

switch, offload, ASIC, bridge, router, OVS, SR-IOV

Introduction

This paper focuses on getting switch chips supported in Linux kernel. Nevertheless, getting the proper free and open-source support into Linux kernel will certainly help other non-Linux based free OSes as well.

In this paper we are going to use the term “switch” as a generic term to refer to ASICs (Application-Specific Integrated Circuit) that not only support L2 but also L3, flow-based forwarding, etc.

Note that the scope of this paper does not include customizing Linux for particular switch boards. It only covers Linux kernel infrastructure needed to enable existence of switch chip drivers.

Each vendor provides binary a SDK (Software Development Kit) to support their chips. These SDKs have proprietary API. SDKs does not fit to the Linux open-source infrastructure, they cannot not be included in the major distributions.

The paper provides an alternative support solution, using free and open source approach. That includes proper in-kernel switch chip support with proper infrastructure witch allows to use existing tools.

The paper begins with describing the status quo of current switch support in Linux. Then, the tools needed to make the switch ASICs relevant in Linux will be described. The comparison of the merits of putting the driver in kernel in compare to userspace will follow. Then, the desired model will be presented. The current upstream Linux kernel state of switch chip integration and description of future vision including L3 routing and flow-based forwarding offloads will follow. At the end, the SR-IOV use-case will be presented.

Switch Chips Support And How Enhance It

This section describes the current state of switch chips support in Linux. A proposal to bring this forward is introduced together with desired model, behavior and features.

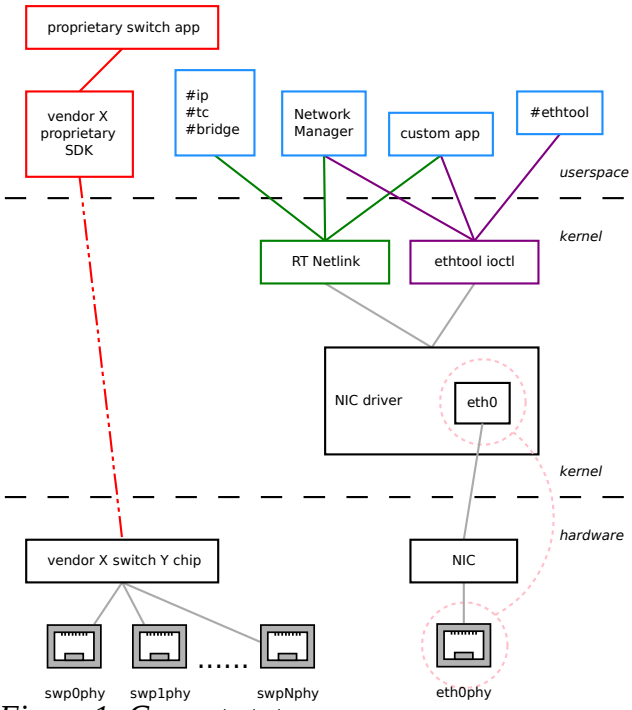

Figure 1: Current state

Switches In The Ice Age

The current poor state of Linux switch support could be easily described as being in the ice age. Figure 1 shows the current state on the left side, in red color. For comparison, on the right side there is shown how standard NIC ecosystem looks like. The pink dotted lines emphasize the relationship of a physical port and netdevice instance.

There are at least two parties involved: The switch chip vendor and switch box vendor. Usually, the switch box vendors do not manufacture the switch chips. They buy chips from other companies and only integrate them in custom switch boards, including other chips, including CPU etc.

Chip vendors believe not exposing interfaces to their ASICs gives them a technical advantage over their competitors. Often times the real reason is economical in locking in their customers given that these ASICs are commodity. Therefore, they came up with their own wrapping library, in the form of SDK. This allows user to communicate with the chip only using exposed set of wrapper functions.

These SDKs are userspace binaries which directly access switch chip hardware, without kernel involvement. This causes great inconsistency of approach comparing to other class of devices, for example NICs. Naturally, every SDK by different vendor looks differently. Each SDK's API is different. So this brings a great vendor lock-in for the switch box vendor.

Great number of switch boxes use custom Linux-based OSes. The existence of SDKs has another consequence. The switch box vendors are pushed to create their own proprietary tools for switch chip manipulation, even though they use Linux kernel. So this brings a great vendor lock-in for switch box users – customers.

Getting Out Of The Ice Age

So it is clear that in order to break this chain and make the switch environment vendor lock-in free, open and maintainable, there is a need for free and open-source approach. There is need for an infrastructure so the switch chip vendors can create their own (hopefully) open-source drivers using the mentioned infrastructure.

That may eventually lead to the “perfect” state where a customer can go buy any switch box, install Linux distribution of choice and run every switch box in the same uniform way, no matter what specific switch chip hardware it is based on. Something similar to how the desktop and server environment looks like at this moment.

The question was if this infrastructure should exist in kernel or rather in userspace. There were some people who preferred the second option. Their main argument was that the amount of code is so big it would not be appropriate to have it in kernel. This paper argues that the kernel approach is the preferred approach to solving the problem.

Given the fact Linux is monolithic kernel, all hardware drivers reside in kernel. Even those, what once were in userspace. There is need to be able to use standard tooling for switch chips manipulation and that is not possible without driver being in kernel. Of course, in theory, this could be achievable with userspace approach. But that would require ugly and most likely unacceptable “trampoline model” in which kernel acts as a repeater between multiple userspace applications (tools and userspace drivers).

Desired model

The main goal certainly is to provide an infrastructure which would allow to reuse existing tools without need to do any or only minimal modifications.

The most fundamental building block for the model is a switch port. It makes a lot of sense to represent these ports in form of a network device so every port would be seen as an independent network device.

- These port devices should be also able to to work as independent NICs. Users can assign L3 addresses to them and use them for sending and receiving packets. In case a user sets up routing among the ports, the driver should be notified about it. In case hardware is capable, the driver should offload set up offloading so the routing happens directly in hardware.
- Another important use-case would be to put the port devices into a layered devices, such as bridge, bond, Open vSwitch datapath and so on (those are referred to as “layered devices” later in text). In this case, the functionality of layered device may be fully or partially offloaded into switch hardware. Driver should take care of the hardware offloading setup and also it should provide info about used offloads to higher layers.

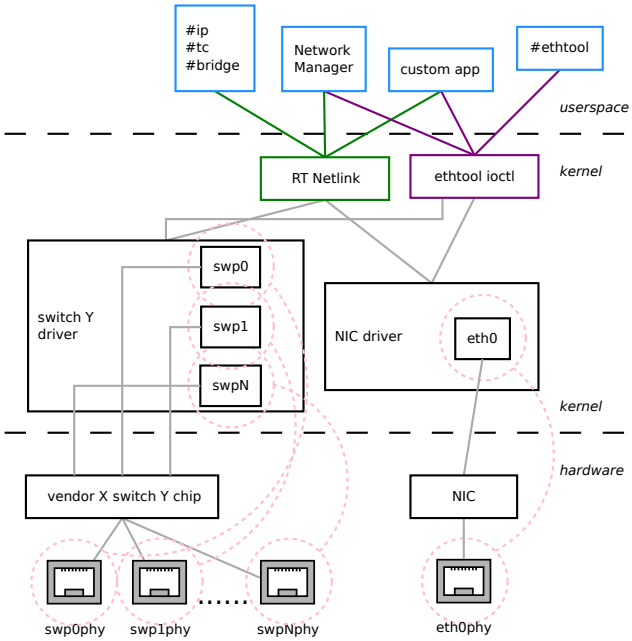

Figure 2: Desired model

- Drivers should provide ethtool API for standard option settings, such as link speed, duplex, offload features, PHY type so as getting port statistics.
- There is need to provide a way to find out if two ports belong to the same switch chip. That is not only informal for user, but it also provides an information for application and layered devices if there is possible to offload features between the ports.

Figure 2 shows the desired model. It is apparent that the ecosystem will look very much like the existing NIC one.

Tooling

Over the years, a standard set of tools was assembled in Linux networking world. Following is a brief overview of related tools to switch chips. Most of them communicate with kernel using the Netlink interface. Only ethtool is using ioctl interface.

ip This is a very essential tool. For switches, it make sense to use ip for port devices for following purposes:

- port devices listing
- setting the link
- setting L3 addresses
- putting port devices into bridges
- adding VLAN devices on port devices
- getting port statistics
- getting information if two port devices belong to the same switch chip

ethtool This is another widely used tool. It has many weaknesses, for example ioclt interface and inability of kernel to propagate asynchronous events to userspace. But as long as it exists, we need switch drivers to implement ethtool API. For switches, it also make sense to use ethtool for port devices for following purposes:

- getting and setting speed and duplex
- setting carrier state
- getting port statistics
- manipulating existing NIC offloads

bridge A tool for controlling bridge devices. It allows to setup bridge-specific options, manipulate FDB entries and to monitor PF_BRIDGE events. For port devices, user of this tool should be able to push bridge port setting down to hardware. It should be also able to see hardware event, for example FDB learns, etc.

tc This tool is used to configure kernel traffic control system. It is based on matching incoming or outgoing packets in filters and executing predefined chain of actions[1]. For port devices, it is desirable to offload filter matching and actions in case hardware supports it.

open vSwitch toolset There are many tools in open vSwitch[2] package. Many of the are either directly of via a daemon (ovsd) working with kernel datapath module. Together they implement OpenFlow based virtual switch. For port devices, it is desirable to provide possibility to offload flow-based packet forwarding into hardware.

Linux Switchdev infrastructure

There are two main items which the switchdev infrastructure brings along:

- Switch device specific set of network device operations (ndos). These should be added when code needs to pass some information to switch driver and also when core needs to query the driver for some information back. For stacked setups, where for example port device is not directly a port of a bridge, but there is a bonding instance in between, the middle-man driver should take care of propagating the ndo call further down to the port device.
- Switch device notifier. This should be used by switch drivers in order to propagate hardware events to networking core. For stacked setups, middle-man drivers should take care of propagating notifier calls up to their masters.

Figure 3 shows two call chains. On the left side, there are users calling switch actions. Those may be resolved in the driver directly but most of the times, it will be propagated to hardware. On the right side, there is a event notification chain which is called in case an event happens in hardware.

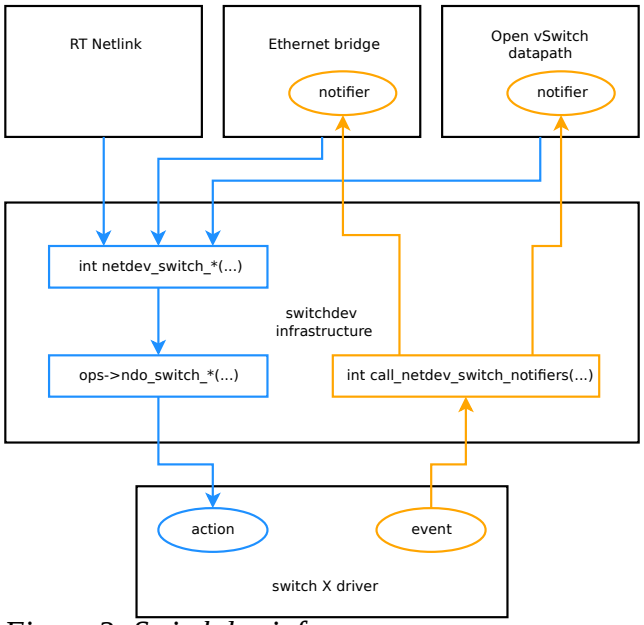

Figure 3: Switchdev infrastructure

L2 forwarding offload

At the moment, the first phase of switchdev infrastructure has been merged into upstream kernel. It includes support for Linux bridge datapath offloading. It also includes the Rocker[3] switch driver which is the first consumer of this infrastructure.

Rocker switch is hardware emulated in QEMU. It follows Broadcom OF-DPA model, but it is possible to extend it easily with another model in future. The main purpose of Rocker switch existence is that it provides possibility to do switch device infrastructure prototyping.

Two new ndos were introduced:

- ndo_switch_parent_id_get – Is called to obtain ID of a switch port parent (switch chip).
- ndo_switch_port_stp_update – Is called to notify switch driver of a change in STP state of bridge port.

There has been introduced switchdev notifier along with two events:

- NETDEV_SWITCH_FDB_ADD
- NETDEV_SWITCH_FDB_DEL

This events are raised by Rocker switch driver in case hardware learns to add of delete a FDB entry and the learned FDB is propagated to bridge. It is possible to disable this from userspace for particular port using IFLA_BRPORT_LEARNING_SYNC flag.

Plans for future – L3 forwarding offload

There has been an attempt[4] to introduce L3 IPv4 support into switchdev infrastructure by Scott Feldman. This patchset tried to introduce two new ndos:

- ndo_switch_fib_ipv4_add
- ndo_switch_fib_ipv4_del

These ndos are called by the core IPv4 FIB code when installing/removing FIB entries to/from the kernel FIB. The patchset also includes implementation of the interface extension in Rocker switch driver.

There is need to determine on which port device the ndo should be called on. Note that in this case port device is used as an entry point to switch device, since the route is not directly anchored on the port. Any of the switch ports could be used. The FIB entry (route) nexthop list is used for this. The route's fib_dev (the first nexthop's dev) is used find the port device by recursively traversing the fib_dev's lower_dev list until a port device is found.

This approach was discussed on the mailing list and some concerns were raised regarding the recursive search in case of a layered devices are involved. One solution may be to allow layered devices to propagate the ndo call themselves.

Also, there must be considered a case when multiple switch chips are involved. In that case, there is need to find an entry port device for all of them.

Another issue was discussed on the mailing list regarding the virtual routing tables. There was a suggestion to include virtual routing tables support into switchdev interface. But the inclusion of virtual routing tables support in kernel does not seem likely at the moment.

Plans for future – Flow-based forwarding offload

There have been couple of attempts[5] to introduce flow-base offloading offload support by John Fastabend, called “Flow API”[6].

The patchset introduces a new Generic Netlink interface called “net_flow_nl” which should be used for offloaded flows maintenance. This is not only supposed to be used to flow insertion, deletion and statistics gathering, but also for obtaining hardware capabilities. Userspace app should query these capabilities and process the insertion accordingly.

The patchset introduces a set of ndos to query the capabilities:

- ndo_flow_get_actions
- ndo_flow_get_tbls
- ndo_flow_get_tbl_graph
- ndo_flow_get_hdrs
- ndo_flow_get_hdr_graph

And to insert and remove flow rules:

- ndo_flow_set_rule
- ndo_flow_del_rule

Together with the infrastructure, the patchset introduces its implementation in Rocker switch driver.

In the future, it is plan to support not only getting the tables and table graph, but for the hardware which supports it, also to set it according to user needs.

There has been a discussion whether it is correct to introduce a new user interface just for the hardware offloading purposes. There might make sense to provide use the same interface for in-kernel flow-based forwarding implementations as well. These are of course TC filter-action subsystem Open vSwitch kernel datapath and nftables.

All of these interfaces have separate and independent user interface. User app which would like to maintain both in-kernel and offloaded flows setup would have to use two separate interfaces. That seems incorrect.

Another problem with “Flow API” is that it allows userspace to directly set-up hardware, without the same features being implemented in kernel. So in that case, the correct term is not “feature offloading” but rather “hardware dataplane configuration”. This approach in not acceptable in mainline kernel, where every offloaded functionality much be implemented in kernel as well.

TC-based flow manipulation interface

There has been a proposal[7] to expose the Flow API not via a separate Generic Netlink, but to rather reuse TC filter-action interface for both software implemented (Open vSwitch) and offloaded datapath.

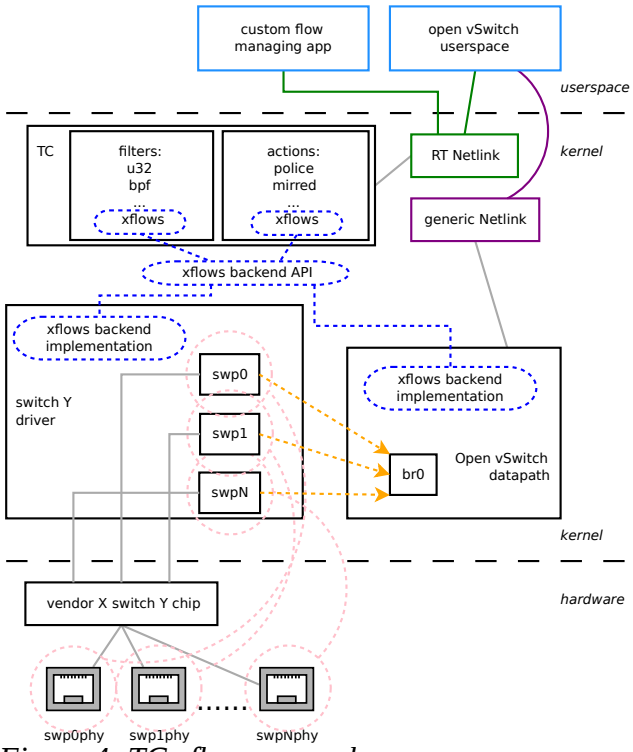

Figure 4: TC xflows example

That is shown in Figure 4. For seamless communication between a userspace application and kernel, a new TC interface called “xflows” is used. It consists of xflows filter and xflows action. These two act as a front-end to userspace. There can be multiple back-ends implemented for xflows, each representing hardware or software datapath with specific capabilities.

There is a downside using TC interface for flows API. That is RTNL mutex which is held for all TC Netlink actions. However, it is very likely possible to change the interface so that RTNL mutex could be avoided. This needs further research.

This approach is also unacceptable in mainline kernel because of the same reason “Flows API” is not. It allows to configure hardware for features which are not implemented in kernel.

SR-IOV use-case

Network interface cards with SR-IOV capability include an embedded switch. This switch implementation differs from card to card.

There is no reason to look at this switch differently and it should be treated as ordinary standalone switch. Figure 5 shows a SR-IOV example use-case with one PF and 3 VFs. It does not matter if particular VFs are passed to virtual machines or not. There is a separate driver for NIC embedded switch and for NIC itself (might be a separate driver or PF and VFs as well).

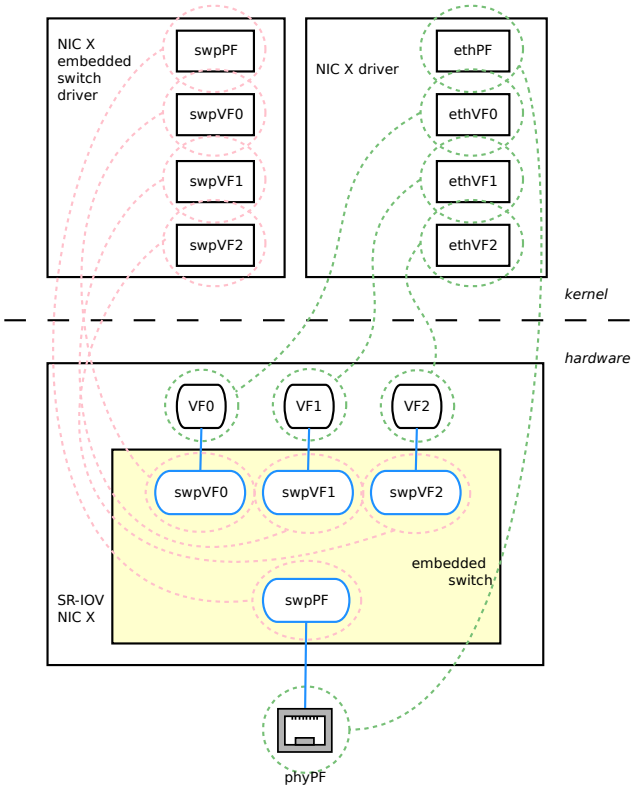

Figure 5: SR-IOV example

Currently, in virtualization setups, it is very challenging to achieve line rate when forwarding small packets (64bytes) to and from virtual machine, even on 10Gbit links. This could be resolved by SR-IOV NIC with an embedded switch capable of offloading Open vSwitch functionality.

DSA use-case

DSA (Distributed Switch Architecture) is a driver for multiple similar switch chips. They mostly act as a PHY connected to MII (Media Independent Interface) bus.

Figure 6 shows how DSA is modeled in kernel. Each port is represented with a netdevice. In order to transmit and receive packets via this netdevices, DSA transparently uses the host facing interface. DSA tags are used to distinguish through what port a packet should be send and from what port a packet was received.

It makes sense to extend DSA to use switchdev API as well in order to provide desired offload possibilities.

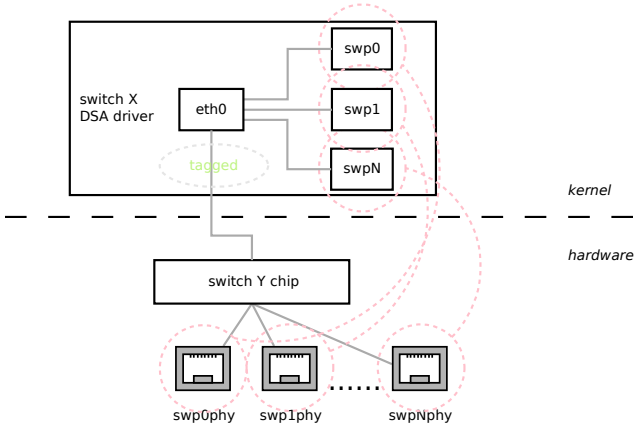

Figure 6: DSA example

Conclusion

The current effort to handle switch chips in a open-source way is around for about a year. Over the time, it is notable that some of the people are changing view. First, the chip vendors were very happy with their SDKs and showed minimal or no interest in doing things differently. This is slowly changing, and it is great to see vendors contributing their ideas to the discussions and by sending patches.

There is a large interest in supporting SR-IOV cards with advanced embedded switch functionality. The current kernel model allows initial support for bridge features offload. Proper and full support of SR-IOV embedded switches may be the breaking point. Support for the ASICs will probably follow.

It is only a matter of time when the solution for offloading flows will be introduced and merged to mainline kernel as well. After that, there will be relatively easy to offload flow-based forwarding solutions, including very popular Open vSwitch.

Acknowledgements

Thanks belongs to all people involved in the discussions on this topic, namely Jamal Hadi Salim, Thomas Graf, Scott Feldman, John Fastabend and Roopa Prahbu.

References

1. Jamal Hadi Salim, “TC Classifier Action Subsystem Architecture”, Proceedings of Netdev 0.1, Feb 2015
2. Open vSwitch website, http://openvswitch.org/
3. Scott Feldman, “Rocker: switchdev prototyping vehicle”, Proceedings of Netdev 0.1, Feb 2015
4. Scott Feldman, “patchset: swdev: add IPv4 routing offload”, http://www.spinics.net/lists/netdev/msg310259.html
5. John Fastabend, “patchset: Flow API”, http://www.spinics.net/lists/netdev/msg313071.html
6. John Fastabend, “Flow API: An abstraction for hardware flow tables”, Proceedings of Netdev 0.1, Feb 2015
7. Jiří Pírko, “xflows proposal”, http://www.spinics.net/lists/netdev/msg313485.html

Author Biography

Jiří Pírko is a Linux kernel hacker who has been digging in networking subsystem for some while. He is an author of Team device, a bonding replacement. He is also a co-author of Rocker switch and its support in Linux kernel including the switchdev infrastructure. His life purpose is to keep things open, nice, clean and easy.