Offloading to yet-another software switch

Michio Honda
NetApp
firstname@netapp.com

Abstract

Recent software switches, such as VALE and DPDK-based Open vSwitch have significant advantages over traditional Linux bridge in terms of throughput, scalability and/or flexibility. For example, VALE, a software switch based on netmap, forwards 64 byte frames at 10 Mpps with L2 learning logic, which is approximately 10 times faster than Linux bridge; and it scales to hundreds of switch ports using a novel packet switching algorithm, which is important when we use a software switch as a backend to interconnect VMs and NICs.

In this paper we present experience with offloading packet switching to VALE under familiar Linux bridge control. We exploit recent extensions in Linux bridge to offload packet switching to switch ASICs like Rocker while keeping control in Linux. Offloading to software switches improves packet switching without switch ASICs. It also improves packet switching for software ports where VMs or applications attach.

1 Introduction

Software switches have played an important role to interconnect virtual machines and NICs as well as to alter hardware switches. Being maintained as a part of operating system's network stack, they are widely used in today's production systems.

Unfortunately, software switches have not benefited from network stack's evolution. For example, extremely general packet representation which supports multiple consumers and non-contiguous buffers is wasteful for packet switching. Further, software switches cannot use NIC's offloading capabilities to reduce *per-byte* cost, such as TCP Segmentation Offload and Large Receive Offload, because they transform the original packets.

Therefore, modern, out-of-tree software switches, such as VALE and DPDK-based Open vSwitch use minimalistic, thus efficient data structures, APIs and techniques like zero-copy, polling and batching. As a result, they can forward minimum-sized packets at 10 Mpps or higher using a single Xeon 3.2 Ghz CPU core, while Linux bridge can do so only at 1 Mpps.

However, it makes their deployment hard that these new switches implement their own control or CLIs. Today's production systems heavily rely on Linux bridge in their operation that is often automated. Slight difference from Linux bridge, or even the fact that what they control is not a Linux bridge, impacts on operation of existing systems.

In this paper we describe solution to address this problem. We exploit a set of recent extensions to Linux bridge called "switchdev API", which is designed to support switch ASICs under the control of Linux bridge, such as attaching and detaching ports to a switch instance with `ip link` command, and adding and deleting a L2 forwarding entry with `bridge` command. In other words, switchdev API enables *offloading* packet forwarding to switch ASICs. We demonstrate that the switchdev API enables not only control of switch ASICs but also that of out-of-tree software switches.

This approach enables offloading packet forwarding without switch ASICs, because even on the same hardware setup, out-of-tree software switches achieve much higher performance, lower latency and/or higher scalability to the number of ports than Linux bridge.

Offloading to a fast software switch also accelerates virtualization backend. It is quite common to run a large number of VMs that share the same NIC, while the software switch routes packets between one of these VMs and the NIC, or between VMs based on installed rules. The software switch also often performs encapsulation and decapsulation of tunneling protocols. Note that sharing the NIC using SR-IOV or similar hardware features is not always possible, because it provides much lower flexibility in routing and tunneling.

2 Overview of VALE software switch

In the rest of this paper we describe a case study with VALE which is a netmap-based, modular software switch that runs in the kernel. VALE outperforms Linux bridge by approximately ten times, because it inherits preallocated, compact data structures from netmap API, and it extensively exploits batching to amortize costs of device register access, system-call and output port locking. VALE also scales to a large number of switch ports because of a novel packet switching algorithm. VALE is implemented in the same kernel module with netmap API.

VALE by default operates as a L2 learning bridge which is a; however, it can be replaced by a separate kernel module that implements a different switching logic. More features of VALE are described in [1, 2].

VALE interconnects regular network interfaces represented as `struct net_device`, but it operates these interfaces with enabling netmap or in *netmap-mode* in order for efficient packet I/O. In this mode, when a series of packets arrives,

packet processing is intercepted before sk_buffs are allocated; instead, these packets are stored in a simple, preallocated netmap ring that accompanies netmap slots. [1] netmap thus extends struct net_device with a pointer to a netmap-specific structure that contains netmap rings and slots. But since netmap reuses an unused pointer, we do not need to recompile the entire kernel. VALE processes packets on these data structures throughout packet forwarding.

3 Extending VALE to meet switchdev API

Overall, extending VALE for switchdev API is pretty easy. Most of necessary code can be taken from Rocker (emulated) switch ASIC driver which is at drivers/net/ethernet/rocker/.

We exploit 1.) a callback on network device events that is registered by register_netdevice_notifier() to configure VALE switch ports, 2.) new net_device_ops callbacks to forward commands for L2 forwarding database and 3.) an interface of call_netdev_switch_notifiers() to notify the Linux bridge of VALE switch configuration change. For simplicity, we use the term switchdev API to refer to all of them.

When loading a VALE kernel module, we simply register_netdevice_notifier() to listen to bridge creation, port attach and port detach in the Linux bridge, which is usually triggered by the ip link command. Upon reception of each event VALE manipulates its switch instance to be equivalent to the corresponding Linux bridge.

This process is specific to software switches. While switch ASICs just want to *know* logical structure of the Linux bridge, VALE or other software switches actually construct its switch instance by attaching and detaching a port, or creating a switch instance itself, because their switch instances and ports are dynamic. Therefore, while switch ASICs only process NETDEV_CHANGEUPPER, VALE or software switches also have to process NETDEV_REGISTER to notice bridge creation.

Next, we have to forward commands for L2 forwarding database from Linux bridge to VALE. These commands are usually issued by the bridge command. Here we can use new callbacks in ndo_ops defined for switchdev API: ndo_fdb_add, ndo_fdb_del and ndo_fdb_dump. This part is a bit tricky, because we have to change net_device_ops dynamically after the net_device is initialized. Since net_device_ops is defined as constant in struct net_device, we have to replace entire net_device_ops.

This problem does not happen to a switch ASIC. While it exports all the ports as net_devices, the switch ASIC driver can initialize them using its own net_device_ops, because these net_devices are specific to the switch ASIC. In software switches, on the other hand, they are latecomers or a net_device_ops has been already set by driver's initialization routine. Therefore, the need for dynamic update of net_device_ops is not specific to VALE but common to other software switches that attach existing net_devices.

Once ndo_fdb_* callbacks are registered, the next step is manipulating VALE's forwarding database. This step is easy because this forwarding database is completely internal within VALE. The final step is VALE notifying Linux of changes in tis forwarding database using call_netdev_switch_notifiers().

Here we are ready to configure the VALE switch like a Linux bridge, using familiar ip link and bridge command.

4 Conclusion

We learnt that software switches easily meet switchdev APIs. However, we suggest two improvements in Linux in order to support out-of-tree software switches.

First, it could be useful if we have a better way to dynamically update net_device_ops. At least for in-kernel software switches, it is essential that we need to update bridge-related callbacks in net_device_ops.

Second, it could be useful if Linux bridge can skip calling netdev_rx_handler_register() when attaching a port to an offloading software switch. netmap API supports *generic mode* which uses unmodified device driver. When a NIC operates in this mode, packets are intercepted using netdev_rx_handler_register(). This prevents Linux bridge from doing so to attach the network interface. We believe it is possible because this callback is also unnecessary to switch ASICs.

We also noticed that switchdev API would not meet user-space software switches, such as DPDK vSwitch as easily as in-kernel software switches. User-space software switches could need to create a *fake* kernel switch that implements callbacks on net_device ports to synchronize with a user-space real switch. (These net_device ports are also fake if we run device drivers in user space.)

For future work we plan to work on L3 forwarding. As described in Section 2, VALE's switching logic can be replaced by a separate kernel module. It could be useful if we implement a module that performs very fast IPv4 lookup like DXR [3] and connects its control to callbacks in net_device_ops.

References

[1] L. Rizzo and G. Lettieri, *Vale: a switched ethernet for virtual machines*, Proc. acm conext, 2012December.

[2] L. Rizzo, G. Lettieri, and M. Honda, *Netmap as a core networking technology*, AsiaBSDCon (2014).

[3] M. Zec, L. Rizzo, and M. Mikuc, *Dxr: towards a billion routing lookups per second in software*, SIGCOMM Comput. Commun. Rev. **42** (September 2012), no. 5, 29–36.

[1] Intercepting packets before sk_buff allocation is done by patching device driver code. However, netmap also supports a mode that uses unmodified device driver at the expense of performance. This mode is implemented by registering callback using familiar netdev_register_rx_handler().