Flow-based tunneling for SR-IOV using switchdev API

Ilya Lesokhin, Haggai Eran, Or Gerlitz
Mellanox
Yokneam, Israel
ilyal@mellanox.com haggaie@mellanox.com ogerlitz@mellanox.com

Abstract

SR-IOV devices present improved performance for network virtualization, but pose limitations today on the ability of the hypervisor to manage the network. For instance, UDP and IP tunnels that are commonly used on the cloud are not supported today with SR-IOV. Flow based approaches like Open vSwitch and TC are common in managing virtual machine traffic. Both technologies are not supported with today's SR-IOV Linux driver model, which only allows to program MAC or MAC+VLAN based forwarding for virtual function traffic. We present a design that facilitates SR-IOV performance while maintaining flow-based management for both non-tunneled and VXLAN tunneled flows and uses the switchdev framework to program the SR-IOV eSwitch. Our prototype uses hardware offloads for most traffic, and a software fallback for traffic we cannot offload. We expose a representor netdev for each port in the SR-IOV eSwitch, one per virtual function and another for the uplink, to enable the management of these ports by the kernel and also the send and receive packets through the software fallback path. Our implementation currently uses openvswitch for managing flows. It should be possible to extend it to other management schemes such as TC. A flow's match and actions are reflected to the underlying device using extended switchdev APIs. For tunneling we also propagate information about the tunnel FDB, and the kernel routing table and neighbor table.

Keywords

SR-IOV, OVS, VXLAN, offload, switchdev, virtualization, flows

Introduction

Traditional hypervisors expose emulated or para-virtual devices to guest virtual machines and multiplexes the I/O requests of the guests onto the real hardware. More recently, there has been an effort to offload those tasks to I/O devices themselves. SR-IOV is a specification by PCI-SIG that allows a single physical device to expose multiple virtual devices. Those virtual devices can be safely assigned to guest virtual machine giving them direct access to the hardware. Using hardware directly reduced the CPU load on the hypervisor and usually results in better performance and lower latency. Those benefits come at a price of management flexibility. Software virtual switches allow implementing complex virtual topologies connecting the virtual machines among themselves and to the data center. On Linux, complex per-packet processing is possible with netfilter, TC as well as with Open vSwitch. In contrast, SR-IOV embedded switches are limited in its expressive power and flexibility, but this limitation is not always due to hardware limitation. In some cases the software model for controlling the SR-IOV switch simply does not allow the configuration of anything more complex than MAC/VLAN based forwarding. In this paper we try to get the best of both worlds: the performance of SR-IOV with the management flexibility of a software switch. We present a richer model for controlling the SR-IOV embedded switch for flow-based switching and tunneling. The model configures the switch dynamically and supports fallback to software in case the hardware is unable to offload all required flows. We show how this model allows integration with Open vSwitch, and describe how VXLAN tunneling can be implemented in this system. Finally, we presents the challenges and open questions we are facing.

SR-IOV flow steering

VF Representors

Each para-virtual device assigned to a guest virtual machine has a TAP device associated with it on the hypervisor. Packets sent through the guests VNIC arrive to the TAP device and packets sent through the TAP device arrive to VNIC. Such TAP devices are connected to virtual switches on the hypervisor and the hypervisor can apply various policies on this virtual switches to control VM traffic. In contrast, traditional SRIOV setups give the VMs direct access the hardware, bypassing the hypervisor. The NIC uses simple MAC+VLAN based forwarding rules that were configured in advance by the hypervisor. We extend the SRIOV setup with the concept of a VF representor netdev per VF, as outlined in [4]. The VF representor plays the same role as the TAP device in the VNIC setup, as illustrated in Figure 1. A packet sent through the VF representor arrives to the VF and a packet sent through the VF arrives to its representor. The existing PF netdev represents the uplink in this model. This configuration gives our SRIOV setup the same management flexibility as the para-virtual or emulated setup but it also removes all the performance benefits of SRIOV as it forces all the traffic to go through the hypervisor. We call the mode of operation the software path.

To get the best of both worlds, management and perfor-

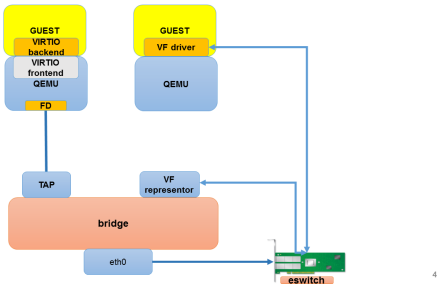

Figure 1: VF representors makes the configuration of VFs similar to the TAP interface

mance, we need to offload the management decisions to the NIC and avoid the software path for most traffic. We stress the importance of offloading most but not all the traffic. Offloading is not all or nothing affair and that we are allowed to continue passing some of the traffic through the software path. For example, when an ARP request is flooded to multiple tunnel, it requires the hardware to send the same packet to the wire with different encapsulation headers, this is rather difficult to do in hardware and offloading such cases is rather pointless as this scenario is quite rare. In our model, we can simply allow such packets to go through the software path. While this model can be used with various virtual bridges, in this paper we discuss the case where the Open vSwitch virtual bridge is used.

Open vSwitch

Open vSwitch is a multilayer virtual switch used for VM traffic management. OVS does flow-based forwarding. The first packet of each flow is passed to a user space daemon called vswitchd. The vswitchd daemon consults an OVSDB database and a set of OpenFlow tables to decide how the packet should be handled. Once the decision is made, a flow is inserted into the OVS kernel data path so that future packets from the same flow would be handled quickly without a context switch to user space. In our work we focus on accelerating SR-IOV setups that use OVS for switching. Our solution adds a hook in the OVS data path that calls switchdev on the ingress port of new flows. In response to that call the device driver of that port attempts to offload that flow the hardware. We note that the ingress port of a flow is well defined because the OVS data path mandates a full match on the ingress port. While it may be possible with some hardware to offload OpenFlow tables directly, we found the format of the OVS data path more suitable for our NICs, as it can be implemented much more efficiently. Currently the OVS data path kernel module will attempt to offload every new flow it receives from user-space, but in the future it should also be possible to allow user-space to prioritize flows and decide which flows to offload.

Switchdev flow API

Offloading flows to the hardware require changes to the switchdev API. There have been several alternatives for similar APIs such as John Fastabends Flow API [1], and updates to TC to expose a flow classifier and actions [5]. As we have been focusing in our prototype on implementing flow offloads for Open vSwitch, we added a primitive flow object to the set of objects switchdev accepts.

Tunneling offload

Open vSwitch tunneling

The OVS data path has push VLAN and pop VLAN actions but does not have similar actions for VXLAN or other tunneling protocols. Instead there is a special VPORT for each tunneling protocol. Encapsulation is represented as two actions. The first action is set attribute which conveys generic tunnel parameters such as destination IP and tunnel ID in a protocol agnostic way. The second action is an output action to a special tunnel VPORT such as a VXLAN VPORT. The type of the output VPORT determines the encapsulation protocol. Similarly, there is no explicit decapsulation, instead forwarding a packet from a VXLAN VPORT to another VPORT implies decapsulation. The user can match the encapsulation protocol by matching the source VPORT and other tunneling parameter are matched in a protocol agnostic way. We note that the VXLAN VPORT does not have switchdev ops and we cant ask it to offload flows coming through it. So instead we had to use fib_lookup() to find the real ingress device and ask it to offload the flow. A Limitation of this approach is that if the routing later changes the new egress device wont be asked to offload the flow unless it is removed and later reinserted into the OVS data path.

Open vSwitch tunneling

Due to the OVS representation of tunnels, a route lookup is required in order to determine through which interface a tunneled flow should be received and through which interface an encapsulated flow needs to be sent. Furthermore, the routing may change while OVS data path rules remain unchanged. This is not an issue when the tunneling is done by software as the routing is done for each packet. But when such flows are offloaded to hardware, the routing changes need to be reflected in the hardware. Our driver keeps track of tunneled flow that it could offload under some routing configuration and tracks routing changes using the already existing hooks for layer 3 support:

```
int netdev_switch_fib_ipv4_add(u32 dst,
  int dst_len, struct fib_info *fi,
  u8 tos, u8 type, u32 nlflags, u32 tb_id);

int netdev_switch_fib_ipv4_del(u32 dst,
   int dst_len, struct fib_info *fi,
   u8 tos, u8 type, u32 tb_id);
```

When our driver receives such an event through one of these hooks, it goes over all tunneled flows and checks

whether they are affected by this change. For flows that need encapsulation we use ip_route_output_key() to see if the output device is still our device and to update the source IP and TTL that will be used in the encapsulation. For flows that need decapsulation we use fib_lookup() to see if that flow is supposed to be received through our device. Since other drivers may need this book keeping as well, and because we need to keep flows even when they are not offloaded to any device, it makes sense to put these data structure in a common module, or in switchdev itself. Having a common module to manage tunneled flow, would also help us with the issue of finding the real ingress port of a flow that needs decapsulation. Rather than doing a one-shot fib_lookup() to find the device when inserting the flow to the data path, the common code would be able to try to offload the flow again to a different device every time there is a routing change.

Reflect routing changes in OVS flows

We can avoid the difficulty of tracking routing changes by modifying OVS slightly. If the OVS data path had a match on the source netdev of incoming tunneled packets then a packet arriving to the tunnel VPORT through a different netdev would cause a miss in the data path and give us a chance to offload the new flow that would be inserted to handle the new packet.

Layer 2 information

Another issue with offloading tunneled flow is that those flows do not include layer 2 information. Consequently this information may change while OVS data path remains unchanged and the hardware need to be notified about those changes. In order to do that we hooked the eSwitch driver with a netevent notifier. We maintain a hash table that maps neighbor table entries to encapsulation headers. Whenever we receive a neighbor update we update the relevant encapsulation headers.

Challenges

Openstack security groups

Openstack currently uses Linux bridges with netfilter rules to implement security groups. In such a setup each VF representor is connected to a Linux bridge and that Linux bridge is connected to OVS. As a result the OVS sees the traffic as coming from the bridge, and not from a representor device. With our current implementation such flows will not be offloaded. Even if we could identify the real source of the traffic, we would have to offload all the netfilter rules to maintain correctness. We plan to work without security groups in the first stage. And we are hoping that in the future the security groups will be implemented with OVN and that we will be able to offload those rules.

Aging

One important challenge we havent tackled yet is the proper aging of OVS flows and Linux neighbor table entries. When the forwarding is done in software, OVS keeps a last-used timestamp on each flow, and updates for every packet that is forwarded. The vswitchd daemon iterates all data path flows once in every given number of seconds, and checks for flows whose last-used value is too far back in the past. Such flows are retired from the data path. Since offloaded flows do not have any of their packets forwarded by the data path module, their last-used field is not up-to-date. To avoid removing them and re-adding them over and over again we plan to have a hardware flow counter gather statistics on each flow. Once in a while, the driver will read all the flows counters, compare them with a previous read, and update the last-used value. With tunneling, the Linux neighbor table also needs to be updated to prevent the system from discarding neighbor entries that are only being used by an offloaded flow.

MTU

Tunneling protocols impose restrictions on the maximum packet size. The maximum size must account for the additional encapsulation headers. To facilitate the configuration of the correct MTU in the VM, we propose reflecting the MTU set to the representor in the VNIC the VF exposes to the VM. The administrator could then set the desired MTU in the hypervisor and thus forcing the VM to use that MTU for all packets.

Related work

The switchdev API [4] is based on having representor netdevs, and although the paper focuses on physical switches, it also presents the SR-IOV case. We follow that model with our VF representors and we hope our analogy to the TAP device makes our use of the model clearer. Flow API [1] is a proposed kernel and user-space API for hardware flow offloads. It allows a netdev driver to expose its flow processing pipeline as a set of tables and supported headers graph. Our design is still in early stages so we havent given much taught to the offload API but we will need to consider something along those lines. Netronome Agilio [3] offloads Open vSwitch flows to a network adaptor similarly to our work. Although a patch that exposes the necessary OVS hooks has been sent upstream [2], the rest of the work is not open source.

Conclusion

We have shown a design that allows keeping the best of both worlds using SR-IOV for improved VM networking performance, without giving up on the flexibility of using Open vSwitch to define and manage the virtual network. Our system supports offloading VXLAN tunneled flows that so far has not been supported with SR-IOV. We hope this work facilitate discussion and bring flow offload support into the upstream Linux.

References

[1] Fastabend, J. 2015. A flow api for linux hardware devices. http://people.netfilter.org/pablo/netdev0.1/papers/A-Flow-API-for-Linux-Hardware-Devices.pdf.

[2] Horman, S. 2014. datapath: offload hooks. http://lwn.net/Articles/615324/.

[3] netronome15. Agilio ovs software architecture. http://netronome.com/media/redactor_files/WP_Agilio_SW.pdf.

[4] Prko, J. 2015a. Hardware switches - the open-source approach. http://people.netfilter.org/pablo/netdev0.1/papers/Hardware-switches-the-open-source-approach.pdf.

[5] Prko, J. 2015b. Implementing open vswitch datapath using tc. http://people.netfilter.org/pablo/netdev0.1/papers/Implementing-Open-vSwitch-datapath-using-TC.pdf.