Rocker: switchdev prototyping vehicle

Scott Feldman

Somewhere in Oregon, USA
sfeldma@gmail.com

Abstract

Rocker is an emulated network switch platform created to accelerate development of an in-kernel network switch driver model. Rocker has two parts: an Qemu emulation of a 62-port switch chip with PCI host interface and a Linux device driver. The goal is to emulate capabilities of contemporary network switch ASICs used in data-center/enterprise so the community can develop the switch device driver interfaces in the kernel. The initial capabilities targeted are L2 bridging function offload and L3 routing function offload. In both cases, the forwarding (data) plane is offloaded to the switch device but the control and management planes remain in Linux. Additional capabilities such as L2-over-L3 tunneling, L2 bonding, ACL support, and flow-based networking are planned or in-progress. This paper/talk will cover an overview of, current status of, and future work on Rocker.

Keywords

Device driver, Linux networking, Qemu, switchdev, network switch ASIC, Motorhead

Introduction

The motivation behind Rocker is to accelerate the development of a Linux in-kernel device driver model for network switches, called switchdev[1]. Vendors of network switch ASICs used in data-center/enterprise class switches typically supply an SDK for programing the switch functions, but these SDKs are provided under NDA making the SDK not suitable for inclusion in an Open Source in-kernel device driver, where a GPL-compatible license is needed. In the absence of vendor-supplied Open Source drivers, Rocker was created as an emulation of a network switch device with a feature set approximating the real-world vendor switch ASICs. With the Rocker device, we can create a device driver to develop and test the switchdev driver model without depending on vendor SDKs. The expectation is once switchdev reaches a certain level of maturity, vender-supplied or community-supplied device drivers for real-world ASICs will appear, and the need for Rocker will diminish over time. Rocker was developed by Scott Feldman and Jiri Pirko, along with support from the Linux netdev community.

Rocker the Device

The Rocker device is a Qemu emulated virtual hardware device that appears to a virtual machine guest as a PCI device with PCI class network:other. As a PCI device, it supports a memory-mappable device register set, DMA support, and MSI-X interrupts. One or more instances of rocker devices can be specified on the qemu command line using the -device parameter. Each instance will be a PCI device with it's own resources. Each instance can have a name, an ID, and a list of front-panel ports. Up to 62 front-panel ports can be specified, with each front-panel port realizing a qemu "nic". An example qemu command line to instantiate a 4-port switch would be:

```
-device rocker,name=sw1,len-ports=4,ports[0]=dev1,ports[1]=dev2, \
   ports[2]=dev3,ports[3]=dev4
```

Four qemu nics dev1-4 can be connected to qemu netdevs to "wire" the ports to the external world using the -netdev parameter. For example, to wire ports 1-4 to host taps, as shown in figure 1, use:

```
-netdev tap,ifname=tap1,dev=dev1 -netdev tap,ifname=tap2,dev=dev2 -netdev
     tap,ifname=tap3,dev=dev3 -netdev tap,ifname=tap4,dev=dev4
```

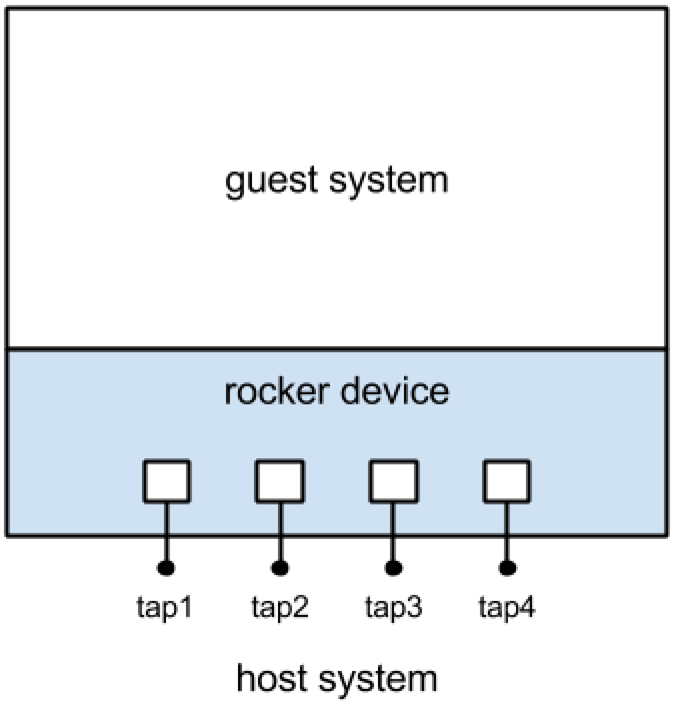

Figure 1. Single 4-port rocker switch wired to 4 host taps.

DMA and Interrupts

A rocker device uses multiple DMA descriptor rings to transfer data to/from the guest system and the device. The registers controlling each DMA ring are detailed in the Rocker Programmer's Reference Guide [2]. The device has a command ring, and event ring, and a per-port Tx/Rx ring pair. The command ring is for requests from the guest system to the device. The even ring is for events from the device to the guest. The Tx/Rx ring pair are used for I/O from the device port to the guest. I/O is typically network packet data for non-forwarded traffic.

All descriptor rings use conventional head and tail pointers to track progress of the producer and consume of the ring. The ring semantics are the same for each ring type: software is the producer of new ring buffers and attaches the buffers starting at current head position, and advancing head position as each buffer is added. Hardware (the device) is the consumer and advances the tail pointer as it uses buffers from the ring. A ring is empty when tail == head. A full ring is when head is one descriptor behind tail. A MSI-X interrupt is associated with each ring, and is fired when the device consumes one or more descriptors on the ring. An interrupt credit reference count is used to track producer and consumer progress and avoid unnecessary interrupts or missing interrupts. More details can be found in the Programmer's Reference Guide.

The descriptor rings use a TLV packing scheme for each descriptor buffer, similar to netlink messages. Using TLVs allows forward- and backward-compatibility of the device and guest software. There is some per-buffer overhead associated with the TLV scheme, but descriptor rings are used for control and event processing, and non-performance-path I/O, so the overhead is inconsequential.

The command ring is for getting and setting device and port settings. Examples include getting switch ID or setting port speed/duplex. The event ring is for asynchronous events from the device such as port link status change or notification of a learned MAC/vlan on a port.

Network Packet Processing

Being a network switch, the primary function of Rocker is to perform some action on ingressing network packets. Actions on the ingressing packet include one or more of counting, dropping, modifying, replicating, and forwarding to an egress port. Rocker uses a plug-in architecture for the packet processing module, called a world. An instance of Rocker may have multiple worlds active, but a given port can only belong to one world at a time, and there is no cross-traffic between worlds. Figure 2 shows a two-world setup.

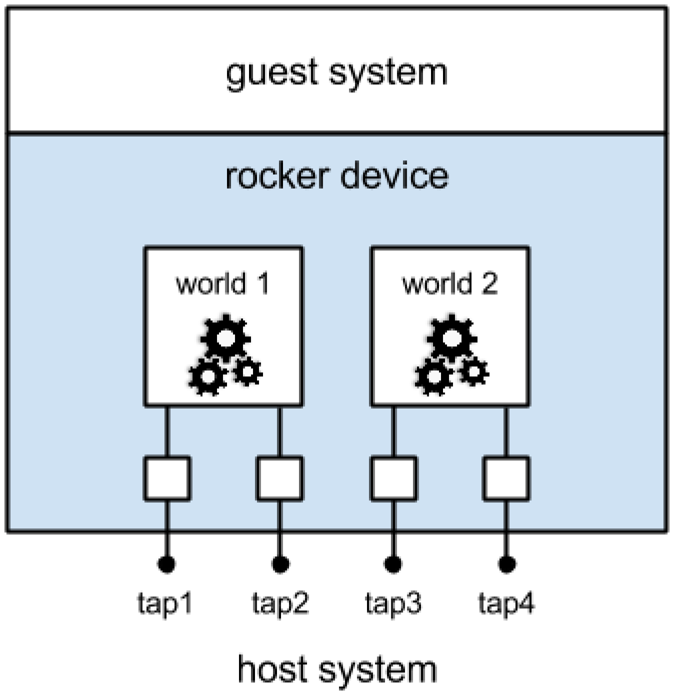

Figure 2. Two-world setup.

OF-DPA World

As of this writing, Rocker implements a single world based on Broadcom's OF-DPA abstract switch specification, version 1.0 [3]. The OF-DPA abstract switch is a specialization of the OpenFlow 1.3.1 abstract switch, and uses OpenFlow objects suchs as flow tables with actions, groups, and ports as basic components. The abstraction also defines a single-pass pipeline, shown in figure 3, which dictates the processing flow. Each flow table matches packet header L2-L4 fields to determine action and next table. Refer to the OF-DPA spec [3] for details on match fields and actions for each table.

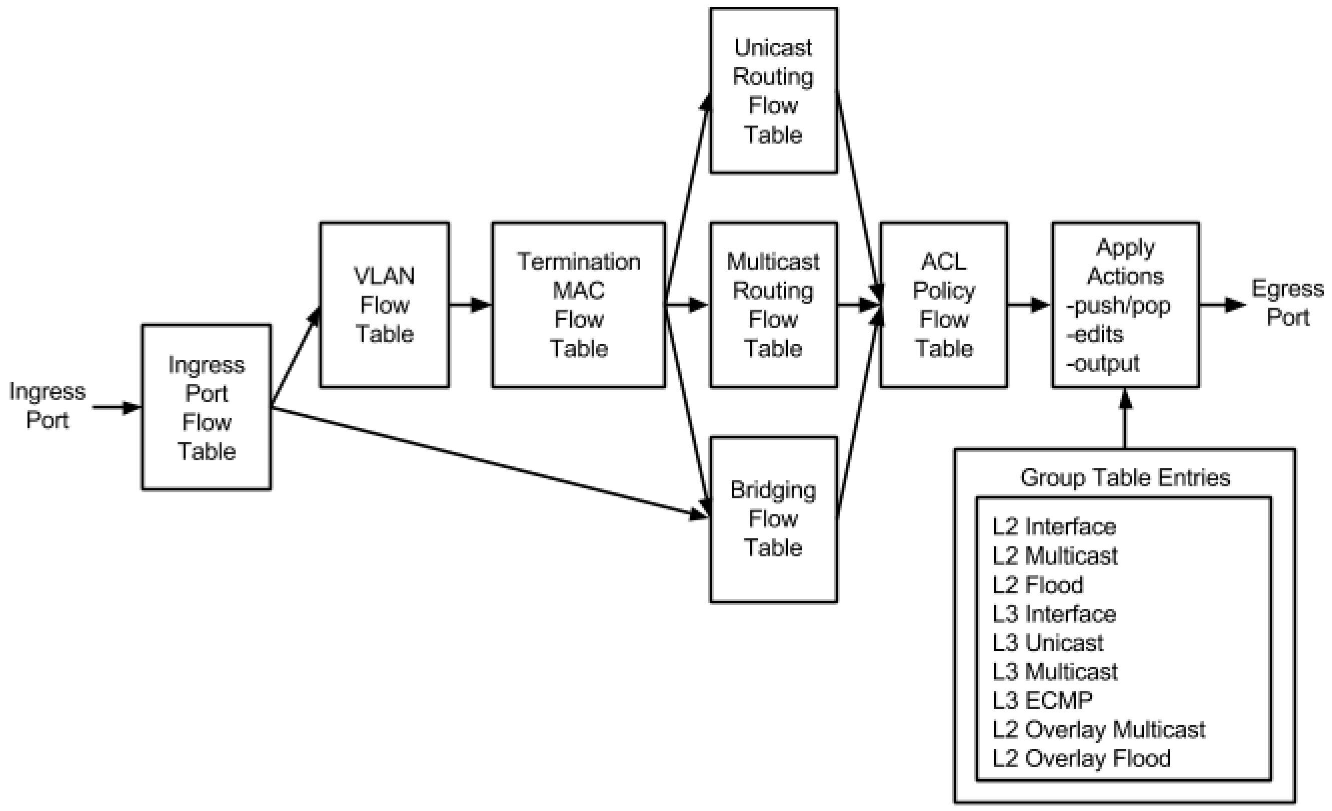

Figure 3. OF-DPA flow pipeline, copied from OF-DPA 1.0 spec.

Although OF-DPA uses OpenFlow objects, we'll see that it can be used in non-OpenFlow setups, in particular when offloading traditional L2/L3 forwarding processing from the Linux kernel.

Rocker the Driver

The Rocker driver is a standard GPL-licensed Linux kernel driver module with PCI lower edge and switchdev upper edge. The driver is available in 3.18 or higher Linux kernels, and it named rocker.ko. The driver source is at drivers/net/ethernet/rocker/ in the Linux kernel tree. An instance of the driver is created for each rocker device probed.

As a switchdev driver, rocker instantiates a netdev interface for each device front-panel port, as shown in figure 4. The netdev interface provides I/O for control traffic processed by the guest CPU for that port. The netdev interface also provides an anchor for other standard Linux tools such as ethtool and iproute2. Physical properties of the port such as PHY settings and link status are also represented by the netdev interface.

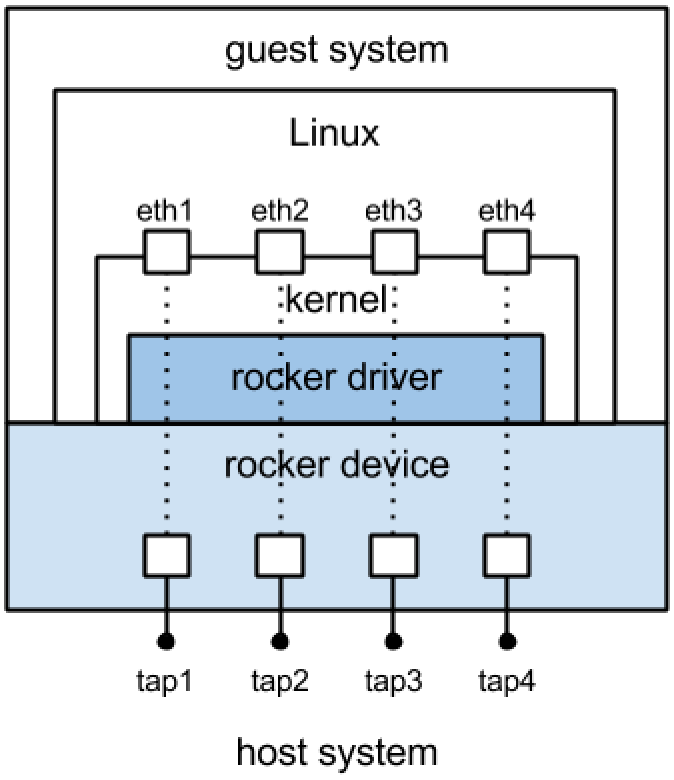

Figure 4. Rocker driver creates netdev interfaces for each front-panel port

The netdev interface is used also to construct L2 bridge and L3 routing functions, offloading the data forwarding path from the Linux kernel. Several new netdev .ndo ops are defined by switchdev to support these functions.

Offloading L2 Bridge Forwarding

Rocker can be used to offload L2 forwarding from the bridge driver, as shown in figure 5. A VLAN bridge is created using standard iproute2 tools and rocker netdev interfaces are added to the bridge. Traffic on the representative ports will now be bridged into one VLAN network, but instead of the Linux bridging driver forwarding traffic from one port to another, the rocker device will forward the traffic. In fact, the forwarded traffic is never seen by the Linux kernel once the device takes over the forwarding function.

Bridge STP control processing remains a function of the Linux bridge driver, and the device passes up STP BPDUs to the bridge driver for processing. STP state transitions on a port are communicated to the device via the switchdev op ndo_switch_port_stp_update. The device driver's ndo_switch_port_stp_update handler will keep the device updated with respect to STP state.

Learning of new MAC addresses on the VLAN on the port is best done by the device, where the device does a look up by source MAC address and VLAN in its FDB. A lookup miss results in a new learned MAC address/VLAN being registered with the Linux bridge driver via a switchdev notifier NETDEV_SWITCH_FDB_ADD call from the driver. FDB entries that expire (age timeout) can be removed with a notifier call NETDEV_SWITCH_FDB_DEL from the driver. The FDB entries learned by the device will appear in the Linux bridge's FDB table with a mark "external" to indicate they have been learned outside the bridge driver.

Static FDB entries can also be programmed into the device using the standard iproute2 command bridge fdb add, and specifying the target "self" so the entry is added to the device FBD rather than the bridge's FDB. Entries can be removed with bridge fdb del command.

Flooding of broadcast/multicast packets and unknown unicast packets will happen at the device level with flooding turned off on the bridge. The device will replicate the packet and flood out to the member ports. A copy of the packet is also sent to the bridge driver so control protocols running on the bridge interface can repond. For example, an ARP request to resolve br0's IP address.

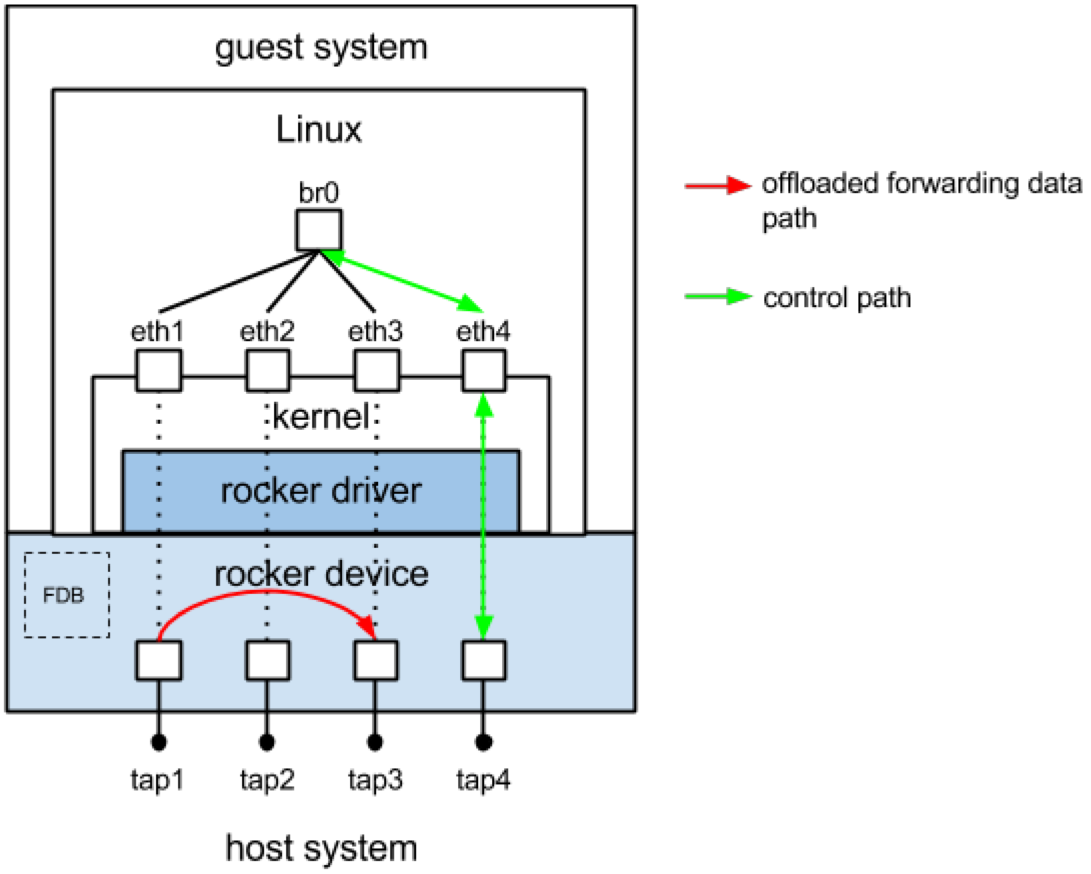

Figure 5. Bridge data paths. Normal forwarding path is port-to-port in device. Control packets pass through device to/from the bridge.

Offloading L3 Routing Function

Rocker can be used to offload L3 routing function from the Linux kernel, as seen in figure 6. The system is configured as a router with IP (and IPv6) forwarding enabled, and connected to other routers or end points within a larger IP network. The routing control processing remains in Linux as standard routing control protocol packets are passed through the device to the routing protocol daemons, such as OSPF or BGP. The kernel's forwarding information base (FIB) is replicated to the rocker device FIB using the switchdev ops ndo_switch_fib_ipv4_add/del[1].

The device will perform LPM (longest prefix match) lookup in the device FIB on incoming unicast[2] IP packets. A hit will result in the IP packet being forwarded by the device to the egress port, bypassing the Linux kernel L3 forwarding function. Since the device FIB and the kernel FIB are the same, the user can continue to use normal iproute2 tools to examine and manipulate the kernel FIB, and changes are synchronized to the device FIB.

[1] New switchdev ops ndo_switch_fib_ipv4_add/del are under discussion on the netdev mailing list and have not been officially accepted into the Linux kernel at this writing time. In the future, similiar ops will be proposed for IPv6 FIB replication.
[2] Multicast routing is planned but not support by device/driver at this writing time.

The ndo_switch_fib_ipv4_add op expects modify semantics when the route exists in the device FIB. The driver and device will update the FIB entry atomically, rather than doing a remove followed by an add. This ensures no interruption of service during the route change.

For the device to properly forward a packet, it must resolve the route's nexthop(s)[3] gateway IP address to a valid destination MAC address. So in addition to the FIB, the device must maintain a neighbor table to lookup a neighbor MAC address given a gateway IP address. To populate the device's neighbor table, the driver monitors the kernel's neighbor table. For gateway IP address not resolved, the driver will proactively send a gratuitous ARP request on a scheduled basis until the neighbor table entry is resolved.

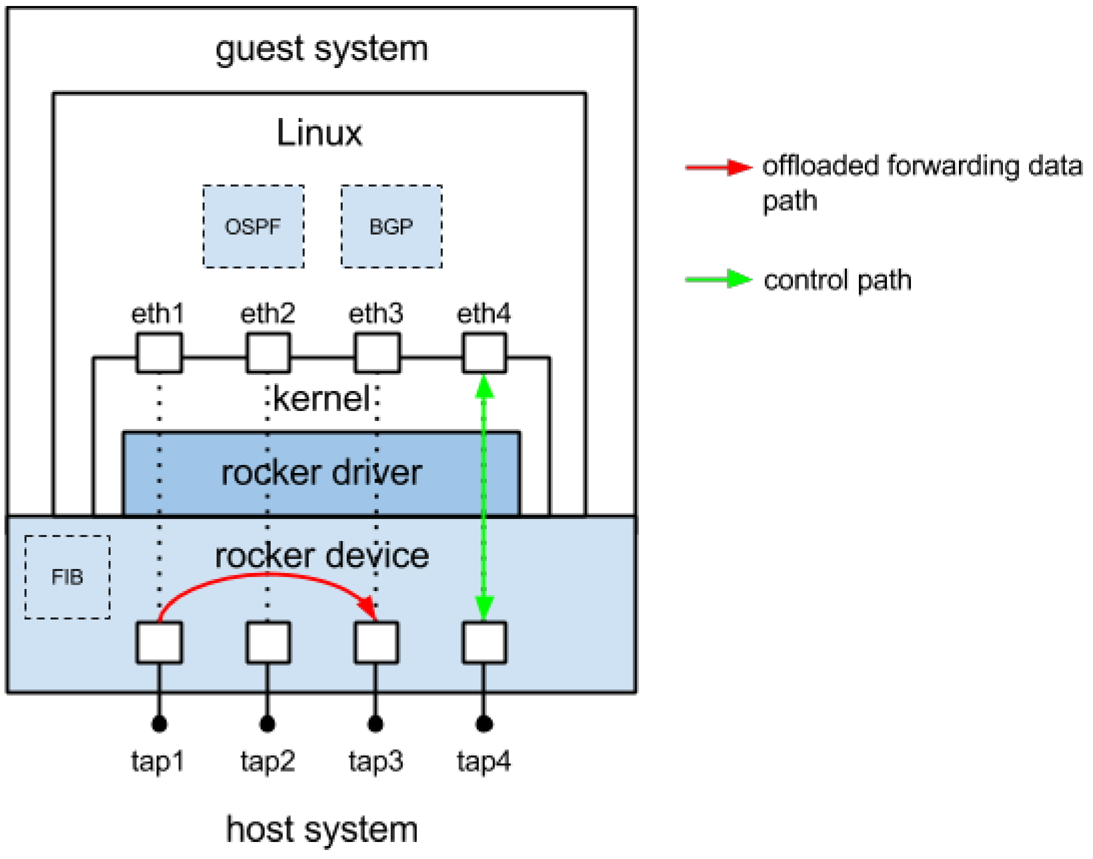

Figure 6. L3 data paths. Normal forwarding path is port-to-port in device. Control packets pass through device to/from the control protocols.

References

1. Switchdev specification in Linux kernel source tree: Documentation/networking/switchdev.txt
2. Rocker Programmer's Reference Guide, docs/specs/rocker.txt in qemu source tree.
3. OpenFlow Data Plane Abstraction (OF-DPA) Abstract Switch Specification, Version 1.0, Broadcom Corporation, February 21, 2014.
4. OpenFlow Data Plane Abstraction (OF-DPA) Abstract Switch Specification, Version 2.0, Broadcom Corporation, November 20, 2014.

[3] ECMP support is planned, but not included in intial patchset.