Ethernet switch device driver model (switchdev)

Copyright © 2014 Jiri Pirko <jiri@resnulli.us>

Copyright © 2014-2015 Scott Feldman <sfeldma@gmail.com>

The Ethernet switch device driver model (switchdev) is an in-kernel driver model for switch devices which offload the forwarding (data) plane from the kernel.

Figure 1 is a block diagram showing the components of the switchdev model for an example setup using a data-center-class switch ASIC chip. Other setups with SR-IOV or soft switches, such as OVS, are possible.

```
                       User-space tools

 user space                   |
+-------------------------------------------------------------------+
 kernel                       | Netlink
                              |
               +--------------+-------------------------------+
               |         Network stack                        |
               |           (Linux)                            |
               |                                              |
               +----------------------------------------------+

                     sw1p2     sw1p4     sw1p6
                sw1p1  +  sw1p3  +  sw1p5  +          eth1
                  +    |    +    |    +    |            +
                  |    |    |    |    |    |            |
               +--+----+----+----+----+----+---+  +-----+-----+
               |         Switch driver         |  |    mgmt   |
               |        (this document)        |  |   driver  |
               |                               |  |           |
               +--------------+----------------+  +-----------+
                              |
 kernel                       | HW bus (eg PCI)
+-------------------------------------------------------------------+
 hardware                     |
               +--------------+----------------+
               |         Switch device (sw1)   |
               |  +----+                       +--------+
               |  |    v offloaded data path   | mgmt port
               |  |    |                       |
               +--|----|----+----+----+----+---+
                  |    |    |    |    |    |
                  +    +    +    +    +    +
                 p1   p2   p3   p4   p5   p6

                       front-panel ports


                              Fig 1.
```

Include Files

```
#include <linux/netdevice.h>
#include <net/switchdev.h>
```

Configuration

Use "depends NET_SWITCHDEV" in driver's Kconfig to ensure switchdev model support is built for driver.

Switch Ports

On switchdev driver initialization, the driver will allocate and register a struct net_device (using register_netdev()) for each enumerated physical switch port, called the port netdev. A port netdev is the software representation of the physical port and provides a conduit for control traffic to/from the controller (the kernel) and the network, as well as an anchor point for higher level constructs such as bridges, bonds, VLANs, tunnels, and L3 routers. Using standard netdev tools (iproute2, ethtool, etc), the port netdev can also provide to the user access to the physical properties of the switch port such as PHY link state and I/O statistics.

There is (currently) no higher-level kernel object for the switch beyond the port netdevs. All of the switchdev driver ops are netdev ops or switchdev ops.

A switch management port is outside the scope of the switchdev driver model. Typically, the management port is not participating in offloaded data plane and is loaded with a different driver, such as a NIC driver, on the management port device.

Switch ID

The switchdev driver must implement the net_device operation ndo_get_port_parent_id for each port netdev, returning the same physical ID for each port of a switch. The ID must be unique between switches on the same system. The ID does not need to be unique between switches on different systems.

The switch ID is used to locate ports on a switch and to know if aggregated ports belong to the same switch.

Port Netdev Naming

Udev rules should be used for port netdev naming, using some unique attribute of the port as a key, for example the port MAC address or the port PHYS name. Hard-coding of kernel netdev names within the driver is discouraged; let the kernel pick the default netdev name, and let udev set the final name based on a port attribute.

Using port PHYS name (ndo_get_phys_port_name) for the key is particularly useful for dynamically-named ports where the device names its ports based on external configuration. For example, if a physical 40G port is split logically into 4 10G ports, resulting in 4 port netdevs, the device can give a unique name for each port using port PHYS name. The udev rule would be:

```
SUBSYSTEM=="net", ACTION=="add", ATTR{phys_switch_id}=="<phys_switch_id>", \
        ATTR{phys_port_name}!="", NAME="swX$attr{phys_port_name}"
```

Suggested naming convention is "swXpYsZ", where X is the switch name or ID, Y is the port name or ID, and Z is the sub-port name or ID. For example, sw1p1s0 would be sub-port 0 on port 1 on switch 1.

Port Features

NETIF_F_NETNS_LOCAL

If the switchdev driver (and device) only supports offloading of the default network namespace (netns), the driver should set this feature flag to prevent the port netdev from being moved out of the default netns. A netns-aware driver/device would not set this flag and be responsible for partitioning hardware to preserve netns containment. This means hardware cannot forward traffic from a port in one namespace to another port in another namespace.

Port Topology

The port netdevs representing the physical switch ports can be organized into higher-level switching constructs. The default construct is a standalone router port, used to offload L3 forwarding. Two or more ports can be bonded together to form a LAG. Two or more ports (or LAGs) can be bridged to bridge L2 networks. VLANs can be applied to sub-divide L2 networks. L2-over-L3 tunnels can be built on ports. These constructs are built using standard Linux tools such as the bridge driver, the bonding/team drivers, and netlink-based tools such as iproute2.

The switchdev driver can know a particular port's position in the topology by monitoring NETDEV_CHANGEUPPER notifications. For example, a port moved into a bond will see it's upper master change. If that bond is moved into a bridge, the bond's upper master will change. And so on. The driver will track such movements to know what position a port is in in the overall topology by registering for netdevice events and acting on NETDEV_CHANGEUPPER.

L2 Forwarding Offload

The idea is to offload the L2 data forwarding (switching) path from the kernel to the switchdev device by mirroring bridge FDB entries down to the device. An FDB entry is the {port, MAC, VLAN} tuple forwarding destination.

To offloading L2 bridging, the switchdev driver/device should support:

- Static FDB entries installed on a bridge port
- Notification of learned/forgotten src mac/vlans from device
- STP state changes on the port
- VLAN flooding of multicast/broadcast and unknown unicast packets

Static FDB Entries

A driver which implements the `ndo_fdb_add`, `ndo_fdb_del` and `ndo_fdb_dump` operations is able to support the command below, which adds a static bridge FDB entry:

```
bridge fdb add dev DEV ADDRESS [vlan VID] [self] static
```

(the "static" keyword is non-optional: if not specified, the entry defaults to being "local", which means that it should not be forwarded)

The "self" keyword (optional because it is implicit) has the role of instructing the kernel to fulfill the operation through the `ndo_fdb_add` implementation of the `DEV` device itself. If `DEV` is a bridge port, this will bypass the bridge and therefore leave the software database out of sync with the hardware one.

To avoid this, the "master" keyword can be used:

```
bridge fdb add dev DEV ADDRESS [vlan VID] master static
```

The above command instructs the kernel to search for a master interface of `DEV` and fulfill the operation through the `ndo_fdb_add` method of that. This time, the bridge generates a `SWITCHDEV_FDB_ADD_TO_DEVICE` notification which the port driver can handle and use it to program its hardware table. This way, the software and the hardware database will both contain this static FDB entry.

Note: for new switchdev drivers that offload the Linux bridge, implementing the `ndo_fdb_add` and `ndo_fdb_del` bridge bypass methods is strongly discouraged: all static FDB entries should be added on a bridge port using the "master" flag. The `ndo_fdb_dump` is an exception and can be implemented to visualize the hardware tables, if the device does not have an interrupt for notifying the operating system of newly learned/forgotten dynamic FDB addresses. In that case, the hardware FDB might end up having entries that the software FDB does not, and implementing `ndo_fdb_dump` is the only way to see them.

Note: by default, the bridge does not filter on VLAN and only bridges untagged traffic. To enable VLAN support, turn on VLAN filtering:

```
echo 1 >/sys/class/net/<bridge>/bridge/vlan_filtering
```

Notification of Learned/Forgotten Source MAC/VLANs

The switch device will learn/forget source MAC address/VLAN on ingress packets and notify the switch driver of the mac/vlan/port tuples. The switch driver, in turn, will notify the bridge driver using the switchdev notifier call:

```
err = call_switchdev_notifiers(val, dev, info, extack);
```

Where val is SWITCHDEV_FDB_ADD when learning and SWITCHDEV_FDB_DEL when forgetting, and info points to a struct switchdev_notifier_fdb_info. On SWITCHDEV_FDB_ADD, the bridge driver will install the FDB entry into the bridge's FDB and mark the entry as NTF_EXT_LEARNED. The iproute2 bridge command will label these entries "offload":

```
$ bridge fdb
52:54:00:12:35:01 dev sw1p1 master br0 permanent
00:02:00:00:02:00 dev sw1p1 master br0 offload
00:02:00:00:02:00 dev sw1p1 self
52:54:00:12:35:02 dev sw1p2 master br0 permanent
00:02:00:00:03:00 dev sw1p2 master br0 offload
00:02:00:00:03:00 dev sw1p2 self
33:33:00:00:00:01 dev eth0 self permanent
01:00:5e:00:00:01 dev eth0 self permanent
33:33:ff:00:00:00 dev eth0 self permanent
01:80:c2:00:00:0e dev eth0 self permanent
33:33:00:00:00:01 dev br0 self permanent
01:00:5e:00:00:01 dev br0 self permanent
33:33:ff:12:35:01 dev br0 self permanent
```

Learning on the port should be disabled on the bridge using the bridge command:

```
bridge link set dev DEV learning off
```

Learning on the device port should be enabled, as well as learning_sync:

```
bridge link set dev DEV learning on self
bridge link set dev DEV learning_sync on self
```

Learning_sync attribute enables syncing of the learned/forgotten FDB entry to the bridge's FDB. It's possible, but not optimal, to enable learning on the device port and on the bridge port, and disable learning_sync.

To support learning, the driver implements switchdev op switchdev_port_attr_set for SWITCHDEV_ATTR_PORT_ID_{PRE}_BRIDGE_FLAGS.

FDB Ageing

The bridge will skip ageing FDB entries marked with NTF_EXT_LEARNED and it is the responsibility of the port driver/device to age out these entries. If the port device supports ageing, when the FDB entry expires, it will notify the driver which in turn will notify the bridge with SWITCHDEV_FDB_DEL. If the device does not support ageing, the driver can simulate ageing using a garbage collection timer to monitor FDB entries. Expired entries will be notified to the bridge using SWITCHDEV_FDB_DEL. See rocker driver for example of driver running ageing timer.

To keep an NTF_EXT_LEARNED entry "alive", the driver should refresh the FDB entry by calling call_switchdev_notifiers(SWITCHDEV_FDB_ADD, ...). The notification will reset the FDB entry's last-used time to now. The driver should rate limit refresh notifications, for example, no more than once a second. (The last-used time is visible using the bridge -s fdb option).

STP State Change on Port

Internally or with a third-party STP protocol implementation (e.g. mstpd), the bridge driver maintains the STP state for ports, and will notify the switch driver of STP state change on a port using the switchdev op switchdev_attr_port_set for SWITCHDEV_ATTR_PORT_ID_STP_UPDATE.

State is one of BR_STATE_*. The switch driver can use STP state updates to update ingress packet filter list for the port. For example, if port is DISABLED, no packets should pass, but if port moves to BLOCKED, then STP BPDUs and other IEEE 01:80:c2:xx:xx:xx link-local multicast packets can pass.

Note that STP BDPUs are untagged and STP state applies to all VLANs on the port so packet filters should be applied consistently across untagged and tagged VLANs on the port.

Flooding L2 domain

For a given L2 VLAN domain, the switch device should flood multicast/broadcast and unknown unicast packets to all ports in domain, if allowed by port's current STP state. The switch driver, knowing which ports are within which vlan L2 domain, can program the switch device for flooding. The packet may be sent to the port netdev for processing by the bridge driver. The bridge should not reflood the packet to the same ports the device flooded, otherwise there will be duplicate packets on the wire.

To avoid duplicate packets, the switch driver should mark a packet as already forwarded by setting the skb->offload_fwd_mark bit. The bridge driver will mark the skb using the ingress bridge port's mark and prevent it from being forwarded through any bridge port with the same mark.

It is possible for the switch device to not handle flooding and push the packets up to the bridge driver for flooding. This is not ideal as the number of ports scale in the L2 domain as the device is much more efficient at flooding packets that software.

If supported by the device, flood control can be offloaded to it, preventing certain netdevs from flooding unicast traffic for which there is no FDB entry.

IGMP Snooping

In order to support IGMP snooping, the port netdevs should trap to the bridge driver all IGMP join and leave messages. The bridge multicast module will notify port netdevs on every multicast group changed whether it is static configured or dynamically joined/leave. The hardware implementation should be forwarding all registered multicast traffic groups only to the configured ports.

L3 Routing Offload

Offloading L3 routing requires that device be programmed with FIB entries from the kernel, with the device doing the FIB lookup and forwarding. The device does a longest prefix match (LPM) on FIB entries matching route prefix and forwards the packet to the matching FIB entry's nexthop(s) egress ports.

To program the device, the driver has to register a FIB notifier handler using register_fib_notifier. The following events are available:

FIB_EVENT_ENTRY_ADD	used for both adding a new FIB entry to the device, or modifying an existing entry on the device.
FIB_EVENT_ENTRY_DEL	used for removing a FIB entry
FIB_EVENT_RULE_ADD,	
FIB_EVENT_RULE_DEL	used to propagate FIB rule changes

FIB_EVENT_ENTRY_ADD and FIB_EVENT_ENTRY_DEL events pass:

```
struct fib_entry_notifier_info {
        struct fib_notifier_info info; /* must be first */
        u32 dst;
        int dst_len;
        struct fib_info *fi;
        u8 tos;
        u8 type;
        u32 tb_id;
        u32 nlflags;
};
```

to add/modify/delete IPv4 dst/dest_len prefix on table tb_id. The `*fi` structure holds details on the route and route's nexthops. `*dev` is one of the port netdevs mentioned in the route's next hop list.

Routes offloaded to the device are labeled with "offload" in the ip route listing:

```
$ ip route show
default via 192.168.0.2 dev eth0
11.0.0.0/30 dev sw1p1  proto kernel  scope link  src 11.0.0.2 offload
11.0.0.4/30 via 11.0.0.1 dev sw1p1  proto zebra  metric 20 offload
11.0.0.8/30 dev sw1p2  proto kernel  scope link  src 11.0.0.10 offload
11.0.0.12/30 via 11.0.0.9 dev sw1p2  proto zebra  metric 20 offload
12.0.0.2  proto zebra  metric 30 offload
        nexthop via 11.0.0.1  dev sw1p1 weight 1
        nexthop via 11.0.0.9  dev sw1p2 weight 1
12.0.0.3 via 11.0.0.1 dev sw1p1  proto zebra  metric 20 offload
12.0.0.4 via 11.0.0.9 dev sw1p2  proto zebra  metric 20 offload
192.168.0.0/24 dev eth0  proto kernel  scope link  src 192.168.0.15
```

The "offload" flag is set in case at least one device offloads the FIB entry.

XXX: add/mod/del IPv6 FIB API

Nexthop Resolution

The FIB entry's nexthop list contains the nexthop tuple (gateway, dev), but for the switch device to forward the packet with the correct dst mac address, the nexthop gateways must be resolved to the neighbor's mac address. Neighbor mac address discovery comes via the ARP (or ND) process and is available via the arp_tbl neighbor table. To resolve the routes nexthop gateways, the driver should trigger the kernel's neighbor resolution process. See the rocker driver's rocker_port_ipv4_resolve() for an example.

The driver can monitor for updates to arp_tbl using the netevent notifier NETEVENT_NEIGH_UPDATE. The device can be programmed with resolved nexthops for the routes as arp_tbl updates. The driver implements ndo_neigh_destroy to know when arp_tbl neighbor entries are purged from the port.

Device driver expected behavior

Below is a set of defined behavior that switchdev enabled network devices must adhere to.

Configuration-less state

Upon driver bring up, the network devices must be fully operational, and the backing driver must configure the network device such that it is possible to send and receive traffic to this network device and it is properly separated from other network devices/ports (e.g.: as is frequent with a switch ASIC). How this is achieved is heavily hardware dependent, but a simple solution can be to use per-port VLAN identifiers unless a better mechanism is available (proprietary metadata for each network port for instance).

The network device must be capable of running a full IP protocol stack including multicast, DHCP, IPv4/6, etc. If necessary, it should program the appropriate filters for VLAN, multicast, unicast etc. The underlying device driver must effectively be configured in a similar fashion to what it would do when IGMP snooping is enabled for IP multicast over these switchdev network devices and unsolicited multicast must be filtered as early as possible in the hardware.

When configuring VLANs on top of the network device, all VLANs must be working, irrespective of the state of other network devices (e.g.: other ports being part of a VLAN-aware bridge doing ingress VID checking). See below for details.

If the device implements e.g.: VLAN filtering, putting the interface in promiscuous mode should allow the reception of all VLAN tags (including those not present in the filter(s)).

Bridged switch ports

When a switchdev enabled network device is added as a bridge member, it should not disrupt any functionality of non-bridged network devices and they should continue to behave as normal network devices. Depending on the bridge configuration knobs below, the expected behavior is documented.

Bridge VLAN filtering

The Linux bridge allows the configuration of a VLAN filtering mode (statically, at device creation time, and dynamically, during run time) which must be observed by the underlying switchdev network device/hardware:

- with VLAN filtering turned off: the bridge is strictly VLAN unaware and its data path will process all Ethernet frames as if they are VLAN-untagged. The bridge VLAN database can still be modified, but the modifications should have no effect while VLAN filtering is turned off. Frames ingressing the device with a VID that is not programmed into the bridge/switch's VLAN table must be forwarded and may be processed using a VLAN device (see below).
- with VLAN filtering turned on: the bridge is VLAN-aware and frames ingressing the device with a VID that is not programmed into the bridges/switch's VLAN table must be dropped (strict VID checking).

When there is a VLAN device (e.g: sw0p1.100) configured on top of a switchdev network device which is a bridge port member, the behavior of the software network stack must be preserved, or the configuration must be refused if that is not possible.

- with VLAN filtering turned off, the bridge will process all ingress traffic for the port, except for the traffic tagged with a VLAN ID destined for a VLAN upper. The VLAN upper interface (which consumes the VLAN tag) can even be added to a second bridge, which includes other switch ports or software interfaces. Some approaches to ensure that the forwarding domain for traffic belonging to the VLAN upper interfaces are managed properly:

    - If forwarding destinations can be managed per VLAN, the hardware could be configured to map all traffic, except the packets tagged with a VID belonging to a VLAN upper interface, to an internal VID corresponding to untagged packets. This internal VID spans all ports of the VLAN-unaware bridge. The VID corresponding to the VLAN upper interface spans the physical port of that VLAN interface, as well as the other ports that might be bridged with it.
    - Treat bridge ports with VLAN upper interfaces as standalone, and let forwarding be handled in the software data path.

- with VLAN filtering turned on, these VLAN devices can be created as long as the bridge does not have an existing VLAN entry with the same VID on any bridge port. These VLAN devices cannot be enslaved into the bridge since they duplicate functionality/use case with the bridge's VLAN data path processing.

Non-bridged network ports of the same switch fabric must not be disturbed in any way by the enabling of VLAN filtering on the bridge device(s). If the VLAN filtering setting is global to the entire chip, then the standalone ports should indicate to the network stack that VLAN filtering is required by setting 'rx-vlan-filter: on [fixed]' in the ethtool features.

Because VLAN filtering can be turned on/off at runtime, the switchdev driver must be able to reconfigure the underlying hardware on the fly to honor the toggling of that option and behave appropriately. If that is not possible, the switchdev driver can also refuse to support dynamic toggling of the VLAN filtering knob at runtime and require a destruction of the bridge device(s) and creation of new bridge device(s) with a different VLAN filtering value to ensure VLAN awareness is pushed down to the hardware.

Even when VLAN filtering in the bridge is turned off, the underlying switch hardware and driver may still configure itself in a VLAN-aware mode provided that the behavior described above is observed.

The VLAN protocol of the bridge plays a role in deciding whether a packet is treated as tagged or not: a bridge using the 802.1ad protocol must treat both VLAN-untagged packets, as well as packets tagged with 802.1Q headers, as untagged.

The 802.1p (VID 0) tagged packets must be treated in the same way by the device as untagged packets, since the bridge device does not allow the manipulation of VID 0 in its database.

When the bridge has VLAN filtering enabled and a PVID is not configured on the ingress port, untagged and 802.1p tagged packets must be dropped. When the bridge has VLAN filtering enabled and a PVID exists on the ingress port, untagged and priority-tagged packets must be accepted and forwarded according to the bridge's port membership of the PVID VLAN. When the bridge has VLAN filtering disabled, the presence/lack of a PVID should not influence the packet forwarding decision.

Bridge IGMP snooping

The Linux bridge allows the configuration of IGMP snooping (statically, at interface creation time, or dynamically, during runtime) which must be observed by the underlying switchdev network device/hardware in the following way:

- when IGMP snooping is turned off, multicast traffic must be flooded to all ports within the same bridge that have mcast_flood=true. The CPU/management port should ideally not be flooded (unless the ingress interface has IFF_ALLMULTI or IFF_PROMISC) and continue to learn multicast traffic through the network stack notifications. If the hardware is not capable of doing that then the CPU/management port must also be flooded and multicast filtering happens in software.
- when IGMP snooping is turned on, multicast traffic must selectively flow to the appropriate network ports (including CPU/management port). Flooding of unknown multicast should be only towards the ports connected to a multicast router (the local device may also act as a multicast router).

The switch must adhere to RFC 4541 and flood multicast traffic accordingly since that is what the Linux bridge implementation does.

Because IGMP snooping can be turned on/off at runtime, the switchdev driver must be able to reconfigure the underlying hardware on the fly to honor the toggling of that option and behave appropriately.

A switchdev driver can also refuse to support dynamic toggling of the multicast snooping knob at runtime and require the destruction of the bridge device(s) and creation of a new bridge device(s) with a different multicast snooping value.