What is an L3 Master Device?

David Ahern
Cumulus Networks
Mountain View, CA, USA
dsa@cumulusnetworks.com

Abstract

The L3 Master Device (l3mdev) concept was introduced to the Linux networking stack in v4.4. While it was created for the VRF implementation, it is a separate API that can be leveraged by other drivers that want to influence FIB lookups or want to manipulate packets at layer 3. This paper discusses the l3mdev implementation, the hooks in the IPv4 and IPv6 networking stack and the driver API, why they are needed and what opportunities they provide to network drivers. The VRF driver is used as an example of what can be done in each of the driver hooks.

Keywords

l3mdev, VRF, IPv4, IPv6

Introduction

The L3 Master Device (l3mdev for short) idea evolved from the initial Virtual Routing and Forwarding (VRF) implementation for the Linux networking stack. The concept was created to generalize the changes made to the core IPv4 and IPv6 code into an API that can be leveraged by devices that operate at layer 3 (L3).

The primary motivation for l3mdev devices is to create L3 domains that correlate to a specific FIB table (Figure 1). Network interfaces can be enslaved to an l3mdev device uniquely associating those interfaces with an L3 domain. Packets going through devices enslaved to an l3mdev device use the FIB table configured for the device for routing, forwarding and addressing decisions. The key here is the enslavement only affects layer 3 decisions.

Drivers leveraging the l3mdev operations can get access to packets at layer 3 similar to the rx-handler available to layer 2 devices such as bridges and bonding. Drivers can use the hooks to implement device-based networking features that apply to the entire L3 domain.

Userspace programs can use well-known POSIX APIs to specify which domain to use when sending packets. This is required since layer 3 network addresses and routes are local to the L3 domains.

Finally, administration, monitoring and debugging for l3mdev devices follows the existing paradigms for Linux networking.

The code references in this paper are for net-next in what will be the 4.9 kernel. Older kernels operate similarly though the driver operations and hooks into the kernel are slightly different. As of this writing there are 2 drivers using the l3mdev API: VRF and IPvlan. This paper uses the VRF driver for examples of what can be done with the driver operations.

Layer 3 Master Devices

The l3mdev feature is controlled by the kernel configuration option CONFIG_NET_L3_MASTER_DEV under Networking support -> Networking options. It must be set to enable drivers that leverage the infrastructure (e.g., VRF and IPvlan).

L3 master devices are created like any other network device (e.g., RTM_NEWLINK and 'ip link add …'); required attributes are a function of the device type. For example, a VRF device requires a table id that is associated with the VRF device while IPvlan does not since it does not use the FIB table aspect of l3mdev.

l3mdev devices have the IFF_L3MDEV_MASTER flag set in priv_flags; devices enslaved to an l3mdev device have IFF_L3MDEV_SLAVE set. These flags are leveraged in the fast path to determine if l3mdev hooks are relevant for a particular flow using the helpers netif_is_l3_master and netif_is_l3_slave.

The l3mdev driver operations are defined in include/net/l3mdev.h, struct l3mdev_ops. As of v4.9 the handlers are:

l3mdev_fib_table – returns FIB table for L3 domain,

l3mdev_l3_rcv – Rx hook in network layer,

l3mdev_l3_out – Tx hook in network layer, and

l3mdev_link_scope_lookup – route lookup for IPv6 link local and multicast addresses.

Drivers using the l3mdev infrastructure only need to implement the handlers of interest. For example, IPvlan only implements the l3mdev_l3_rcv hook for its l3s mode, while the VRF driver implements all of them.

The l3mdev_fib_table and l3mdev_link_scope_lookup are discussed in the next section. The l3mdev_l3_rcv and

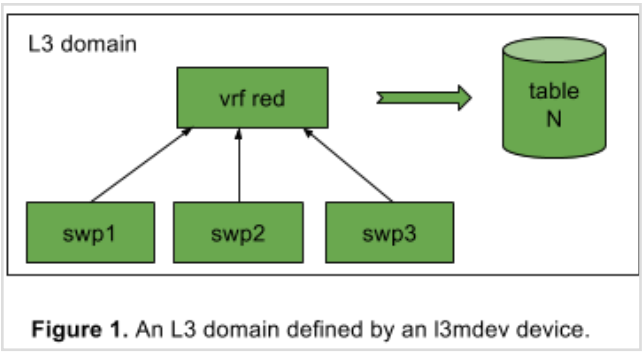

Figure 1. An L3 domain defined by an l3mdev device.

l3mdev_l3_out operations are discussed in the Layer 3 Packet Processing section.

Layer 3 Routing and Forwarding

The primary motivation for Layer 3 master devices is to create L3 domains represented by an associated FIB table (Figure 1). Network interfaces can be enslaved to the l3mdev device making them part of the domain for layer 3 routing and forwarding decisions. A key point is that the domains and enslaving an interface to those domains affect only layer 3 decisions. The association with an l3mdev device has no impact on Layer 2 applications such as lldpd sending and receiving packets over the enslaved network interfaces (e.g., for neighbor discovery).

Network Addresses

Network addresses for interfaces enslaved to an l3mdev device are local to the L3 domain. When selecting a source address for outgoing packets, only addresses associated with interfaces in the L3 domain are considered during the selection process. By extension this means applications need to specify which domain to use when communicating over IPv4 or IPv6. This is discussed in the 'Userspace API' section below.

Since the l3mdev device is also a network interface, it too can have network addresses (e.g, loopback addressing commonly used for routing protocols). Those addresses are also considered for source address selection on outgoing connections. The l3mdev device is treated as the loopback device for the domain, and and the IPv4 stack allows the loopback address (127.0.0.1) on an l3mdev device.

FIB Tables

The FIB table id for an l3mdev device is retrieved using the l3mdev_fib_table driver operation. Since l3mdev_fib_table is called in fast path, the operation should only return the table id for the device presumably stored in a private struct added to the net_device.

Local and connected routes for network addresses added to interfaces enslaved to an l3mdev device are automatically moved to the FIB table associated with the l3mdev device when the network interface is brought up. This means that the l3mdev FIB table has all routes for the domain – local, broadcast, and unicast. Additional routes can be added to the FIB table either statically (e.g, ip route add table N) or using protocol suites such as quagga.

Policy Routing and FIB Rules

The Linux networking stack has supported policy routing with FIB rules since v2.2. Rules can use the oif (outgoing interface index) or iif (incoming interface index) to direct lookups to a specific table. The l3mdev code leverages this capability to direct lookups to the table associated with the master device.

For this to work the flow structure, which contains parameters to use for FIB lookups, needs to have either oif or iif set to the interface index of the l3mdev device. This is accomplished in multiple ways: for locally originated traffic the oif is originally set based on either the socket (sk_bound_dev_if or uc_index) or cmsg and IP_PKTINFO. For responses and forwarded traffic, the original iif or oif are based on the ingress device.

l3mdev has several operations for updating the oif or iif from an enslaved interface to the L3 master device. Generically, this is done in the IPv4 and IPv6 stacks with calls to l3mdev_update_flow before calling fib_rules_lookup. Unfortunately, there are several special cases where the oif or iif is not set in the flow. These have to be handled directly with calls to l3mdev_master_ifindex and related helper functions.

FIB rules can be written per l3mdev device (e.g., an oif and/or iif rule per device) to direct lookups to a specific table:

```
$ ip rule add oif blue table 1001
$ ip rule add iif blue table 1001
```

Alternatively, a single l3mdev rule can be used to direct lookups to the table associated with the device:

```
$ ip rule add l3mdev pref 1000
```

The l3mdev rule was designed to address the scalability problems of having 1 or 2 rules per device since the rules are evaluated linearly on each FIB lookup. With the l3mdev rule, a single rule covers all l3mdev devices as the table id is retrieved from the device.

If an l3mdev rule exists, l3mdev_fib_rule_match is called to determine if the flow structure oif or iif references an l3mdev device. If so, the l3mdev_fib_table driver operation is used to retrieve the table id. It is saved to the fib_lookup_arg, and the lookup is directed to that table.

The l3mdev rule is specified by adding the FRA_L3MDEV attribute with a value of 1 in RTM_NEWRULE and RTM_DELRULE messages.

The VRF driver adds the l3mdev rule with a preference of 1000 when the first VRF device is created. That rule can be deleted and added with a different priority if desired.

IPv6 link scope

Special consideration is needed for IPv6 linklocal and multicast addresses. For these addresses, the flow struct can not be updated to the l3mdev device as the enslaved device index is needed for an exact match (e.g., the linklocal address for the specific interface is needed). In this case, the l3mdev device needs to do the lookup directly in the FIB table for the device. The VRF driver and its function vrf_link_scope_lookup is an example of how to do this.

Also, IPv6 linklocal addresses are not added to l3mdev devices by the kernel, and the stack does not insert IPv6 multicast routes for the devices. The VRF driver for example specifically fails route lookups for IPv6 linklocal or multicast addresses on a VRF device.

Layer 3 Packet Processing

Packets are passed to l3mdev devices on ingress and egress if the driver implements the l3mdev_l3_rcv and l3mdev_l3_out handlers.

Rx

Packets are passed to the l3mdev driver in the IPv4 or IPv6 receive handlers after the netfilter hook for NF_INET_PRE_ROUTING (Figure 2). At this point the IPv4 and IPv6 receive functions have done basic sanity checks on the skb, and the skb device (skb->dev) is set to the ingress device. The l3mdev driver can modify the skb or its metadata as needed based on relevant features. If it returns NULL, the skb is assumed to be consumed by the l3mdev driver and no further processing is done. The l3mdev operation is called before invoking the input function for the dst attached to the skb which means the l3mdev driver can set (or change) the dst if desired hence altering the next function called on it.

The l3mdev_l3_rcv hook is the layer 3 equivalent to the rx_handler commonly used for layer 2 devices such as bonds and bridges. By passing the skb to the l3mdev handler in the networking stack at layer 3, drivers do not need to duplicate network layer checks on skbs. Furthermore, it allows the IPv4 and IPv6 layers to save the original ingress device index to the skb control buffer prior to calling the l3mdev_rcv_out hook. This is essential for datagram applications that require the ingress device index and not the l3mdev index (the latter is easily derived from the former via the master attribute).

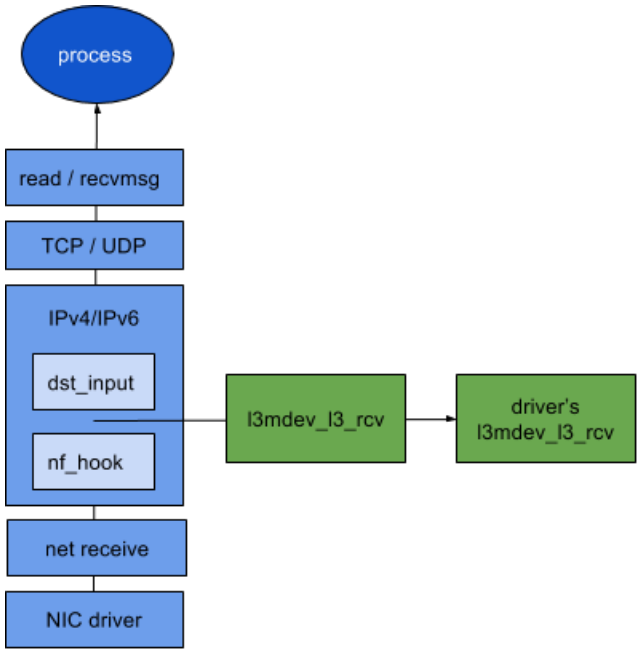

Figure 2. l3mdev receive hook.

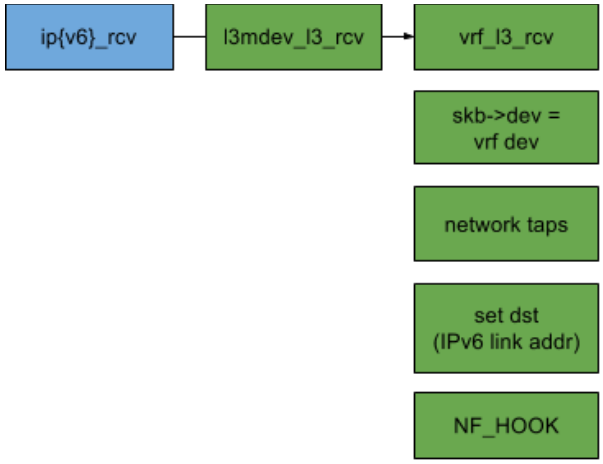

Figure 3. Receive path for VRF.

Prior to the l3mdev hooks, drivers relied on the rx-handler and duplicating network layer code. That design had other limitations such as preventing a device with a macvlan or ipvlan from also being placed in an L3 domain. With this l3mdev hook both are possible (e.g., eth2 can be assigned to a VRF and eth2 can be the parent device for macvlans).

Figure 3 shows the operations done by the VRF driver. It uses the l3mdev receive functions to switch the skb device to its device to influence socket lookups, and the skb is run through the network taps allowing tcpdump on a VRF device to see all packets for the domain.

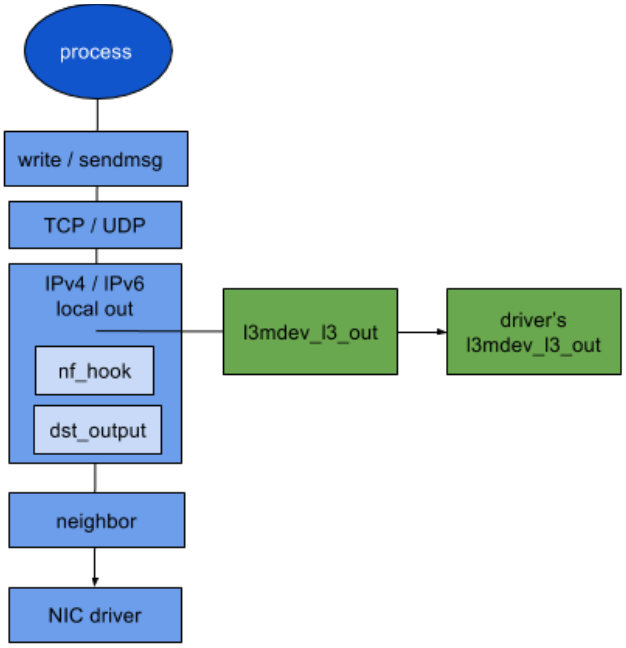

Figure 4. l3mdev transmit hook.

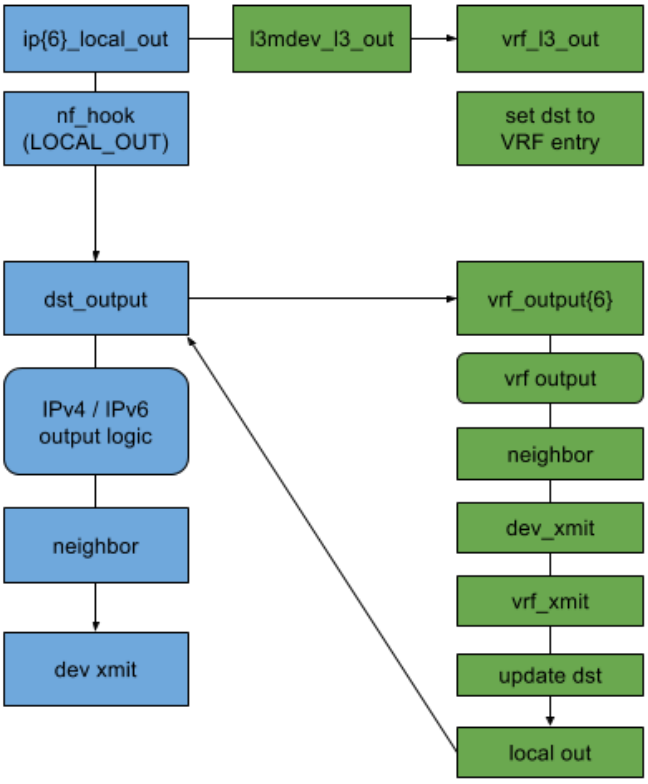

Figure 5. Transmit path for VRF.

As mentioned earlier, IPv6 linklocal and multicast addresses need special handling. VRF uses the l3mdev_ip6_rcv function to do the ingress lookup directly in the associated FIB table and set the dst on the skb.

The skb is then run through NF_HOOK for NF_INET_PRE_ROUTING. This allows netfilter rules that look at the VRF device. (Additional netfilter hooks may be added in the future.)

Tx

For transmit path, packets are passed to the l3mdev driver in the IPv4 or IPv6 layer in the local_out functions before the netfilter hook for NF_INET_LOCAL_OUT. As with the Rx path, the l3mdev driver can modify the skb or its metadata as needed based on relevant features. If it returns NULL, the skb is consumed by the l3mdev driver and no further processing is done. Since the l3_out hook is called before dst_output, an l3mdev driver can change the dst attached to the skb thereby impacting the next function (dst->output) invoked after the netfilter hook.

The VRF driver uses the l3mdev_l3_out handler to implement device based features for the L3 domain (Figure 5). It accomplishes this by using the vrf_l3_out handler to switch the skb dst to its per-VRF device dst and then returns. The VRF dst has the output function pointing back to the VRF driver.

The skb proceeds down the stack with dst->dev pointing to the VRF device. Netfilter, qdisc and tc rules and network taps are evaluated based on this device. Finally, the skb makes it to the vrf_xmit function which resets the dst based on a FIB lookup. It goes through the netfilter LOCAL_OUT hook again this time with the real Tx device and then back to dst_output for the real Tx path.

This additional processing comes with a performance penalty, but that is a design decision within the VRF driver and is separate topic from the l3mdev API. The relevant point here is to illustrate what can be done in an l3mdev driver.

Userspace API

As mentioned in the “Layer 3 Routing and Forwarding” section network addresses and routes are local to an L3 domain. Accordingly, userspace programs communicating over IPv4 and IPv6 need to specify which domain to use. If a device (L3 domain) is not specified, the default table is used for lookups and only addresses for interfaces not enslaved to an l3mdev device are considered.

Since the domains are defined by network devices, userspace can use the age old POSIX apis for sockets - SO_BINDTODEVICE or cmsg and IP_PKTINFO

(datagram sockets). The former binds the socket to the l3mdev device while the latter applies to a single sendmsg. In both cases, the scope of the send is limited to a specific L3 domain, affecting source address selection and route lookups as mentioned earlier.

On ingress, the skb device index is set to the l3mdev device index, so only unbound sockets (wildcard) or sockets bound to the l3mdev device will match on a socket lookup.

The tcp_l3mdev_accept sysctl allows a TCP server to bind to a port globally — i.e., across all L3 domains. Any connections accepted by it are bound to the L3 domain the connection originates. This enables users to have a choice: run a TCP-based daemon per L3 domain or run a single daemon across all domains with client connections bound to an L3 domain.

Performance Overhead

The l3mdev hooks into the core networking stack were written such that if a user does not care about L3 devices the feature completely compiles out. This by definition means the existence of the code has no affect on performance.

When the L3_MASTER_DEVICE config is enabled in the kernel the hooks have been written to minimize the overhead, leveraging device flags and organizing the checks in the most likely paths first. The overhead in this case is mostly extra device lookups on the oif or iif in the flow struct and checking the priv_flags of a device (IFF_L3MDEV_MASTER and IFF_L3MDEV_SLAVE) to determine if the device is l3mdev related.

When an l3mdev is enabled (e.g., a VRF device is created and an interface is enslaved to it) the performance overhead is dictated by the driver and what it chooses to do.

The intent of this performance comparison is to examine the overhead of the l3mdev hooks in the packet path. Latency tests such as netperf's UDP_RR with 1-byte payloads stress the overhead of l3mdev code and drivers such as VRF. This study compared three cases:

1. l3mdev disabled (kernel config option CONFIG_NET_L3_MASTER_DEV is not set).

2. l3mdev compiled in, but no device instances are created.

3. l3mdev compiled in with a minimal VRF device driver.

For case 3, the VRF driver was reduced to only influencing FIB lookups and switching the skb dev on ingress for socket lookups. This means all of the Rx and Tx processing discussed earlier (e.g, the nf_hooks and switching the dst in the output path) were removed. The overhead of VRF and options to improve it are a separate study.

Data were collected using a VM running on kvm with virtio+vhost. The vcpus, vhost threads and netperf command were restrained to specific cpus in the first numa node of the host using taskset. The kernel for the VM was net-next tree at commit 4c1fad64eff4.

Figure 6 shows the average UDP_RR transactions/sec for 3 30-sec runs. Comparing the result for cases 1 and 2 the overhead of enabling the L3_MASTER_DEVICE kernel configuration option is roughly 2.4% for IPv4 and 1.0% for IPv6.

The overhead of the l3mdev code in the fast path is the difference in UDP RR transactions per second between cases 1 and 3. With the minimal VRF driver, a single VRF

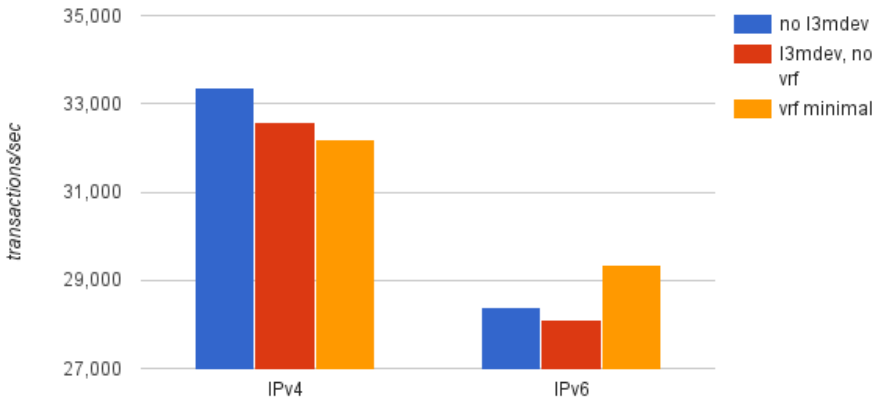

Figure 6. UDP_RR performance

with traffic through an enslaved interface has about a 3.6% performance hit for IPv4 compared to without l3mdev support at all, while IPv6 shows about a 3.2% gain. A cursory review of perf output suggests one reason IPv6 shows a gain is less time spent looking for a source address. Specifically, the l3mdev case spends 1/4 the time in ipv6_get_saddr_eval as only interfaces in the L3 domain are considered.

Despite the best efforts to control variability there is about a 0.5% difference between netperf runs. Figure 6 does show the general trend of what to expect with the current l3mdev code which is a worst case loss in performance around 2 to 3%.

Futhermore, UDP_RR is a worst case test as it exercises the l3mdev overhead in both fib lookups and the l3mdev hooks in the Rx and Tx paths for each packet. Other tests such as TCP_RR show less degradation in performance since connected sockets avoid fib lookups per packet.

Administration, Monitoring and Debugging

l3mdev devices follow the Linux networking paradigms established by devices such as bridging and bonding. Accordingly, l3mdev devices are created, configured, monitored and destroyed using the existing Linux networking APIs and tools such as iproute2.

As a master device, the rtnetlink features for MASTER_DEVICE apply to l3mdev devices as well. For example, filters can be passed in the link request to only show devices enslaved to the l3mdev device:

```
$ ip link show master red
```

or to only show addresses for devices enslaved to an L3 domain

```
$ ip address show master red
```

or neighbor entries for the L3 domain:

```
$ ip neighbor show master red
```

Furthermore, tools such as ss can use the inet diag API with a device filter to list sockets bound to an l3mdev device:

```
ss -ap 'dev == red'
```

Maintaining existing semantics is another key feature of l3mdev and by extension the VRF implementation.

Differences by Kernel Version

The l3mdev API is fairly new (about a year old at the time of this writing) and to date driven by what is needed for the VRF driver. A short summary by kernel version:

v4.4 Initial l3mdev api added

v4.5 tcp_l3mdev_accept sysctl added

v4.7 l3mdev_l3_rcv driver operation added.

v4.8 l3mdev FIB rule added

v4.9 Overhaul of changes for FIB lookups. l3mdev_l3_out operation added.

Conclusions

The L3 Master Device (l3mdev) is a layer 3 API that can be leveraged by network drivers that want to influence FIB lookups or manipulate packets at L3. The concept has been driven by the VRF implementation but is by no means limited to it. The API will continue to evolve for VRF as well as any future drivers that wish to take advantage of the capabilities.

Author Biography

David Ahern is a Member of Technical Staff at Cumulus Networks.