VRF for Linux — a contribution to the Linux Kernel

If you're familiar with Linux, you know how important and exciting it can be to submit new technology that is accepted into the kernel. If you're not familiar with Linux, you can take my word for it (and I highly suggest you attend one of our bootcamps). Many networking features are motivated by an OS for switches and routers, but most if not all of those features prove useful for other use cases as well. Cumulus Networks strives for a uniform operating model across switches and servers, so it makes sense for us to spend the time and effort getting these features into upstream code bases. An example of this effort is Virtual Routing & Forwarding (VRF) for Linux.

I joined Cumulus Networks in June 2015 to work on a VRF solution for Linux —to create an implementation that met the goals we wanted for Cumulus Linux and was acceptable to upstream maintainers for Linux as a whole. That solution was first available last year with Cumulus Linux 3.0 and because of the upstream push that solution is rolling out in general OS distributions such as Debian Stretch and Ubuntu 16.

This post is a bit long, so I start with a high level overview — key points that every reader should take away from this article. I hope you get at

Open Networking Topics

› Technology

› Industry & Market Trends

› Ecosystem

Quick Nav

+ **Blog Authors**

+ **Resources**

of pre-existing Linux routing facilities and the solution that Cumulus Networks spearheaded.

A bird's eye view

The concept of VRF was first introduced around 1999 for L3 VPNs, but it has become a fundamental feature for a networking OS. VRF provides traffic isolation at layer 3 for routing, similar to how you use a VLAN to isolate traffic at layer 2. Think multiple routing tables. Most networking operating systems can support 1000's of VRF instances, giving customers flexibility in deploying their networks.

Over the years, there have been multiple attempts to add proper support for VRF to the Linux kernel, but those attempts were rejected by the community. The Linux kernel has supported policy routing and multiple FIB tables going back to version 2.2 (also released in 1999), but, as I discuss below, multiple FIB tables is but one part of a complete VRF for Linux solution.

Another option that emerged around 2009 is using a network namespace as a VRF. Again, I'll get into this in more detail later, but network namespaces provide a complete isolation of the networking stack which is overkill (i.e. overhead) for VRF (a Layer 3 separation) and the choice of a namespace has significant consequences on the architecture and operational model of the OS.

Cumulus Networks tried all these options and even a custom kernel patch to implement VRF for Linux, but in the end, all of them fell a bit flat. We needed a better solution and decided to spearhead the development of the feature. After many months of hard work, blood, sweat and tears, we developed a solution that works seamlessly with Linux and was accepted into the Linux Kernel. Our solution for VRF is both resource efficient and operationally efficient. It does not require an overhaul in how programs are written to add VRF support or in how the system is configured, monitored and debugged. Everything maintains a logical and consistent view while providing separation of the routing tables.

Because of our commitment to open networking, the VRF for Linux solution is now rolling out in OS distributions allowing it to be used for

the end result appears to be a hit for both networking vendors and Linux users.

View from the ground

So now that you know the high level summary, let's zoom in and look at the details from a Linux perspective. Until the recent work done by Cumulus Networks, Linux users had three choices for VRFs: policy routing and multiple FIB tables, network namespace or custom kernel patches.

Multiple FIB tables and FIB rules fall short of a VRF for Linux

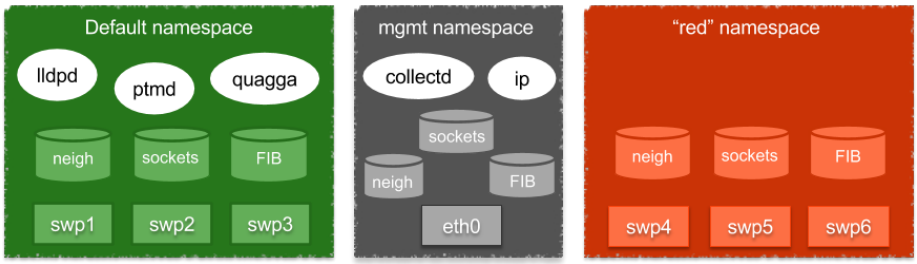
Linux has supported policy routing with multiple routing tables and FIB rules to direct lookups to a table since kernel version 2.2 released in January 1999. As many people have noted, you can kind of, sort of, get a VRF-like capability with them, after all VRF is essentially multiple routing tables, but it is an ad hoc solution at best and does not enforce the kind of isolation one expects for a VRF.

A major shortcoming with this approach is the lack of an API for a program to specify which VRF to use for an outgoing connection or an API for a program to learn the table (VRF) for incoming connections and messages. Further, this approach lacks any strong binding between interfaces and FIB tables. Sure, FIB rules can be installed to direct packets received on an interface to a specific table, but that does not work for locally originated traffic. And, FIB rules per interface do not scale as the number of interfaces increases (physical network interfaces, vlan sub-interfaces, bridges, bonds, etc). The rules are evaluated linearly for each FIB lookup, so an increasing rule set has a significant impact to performance.

Other shortcomings include lack of support for overlapping or duplicate network addresses across VRFs; it is more difficult (near impossible) to program hardware for hardware offload of forwarding and ensure consistency between software and hardware programming; and there is no way to have proper source address selection on egress traffic especially with the common practice of VRF-local loopback addresses.

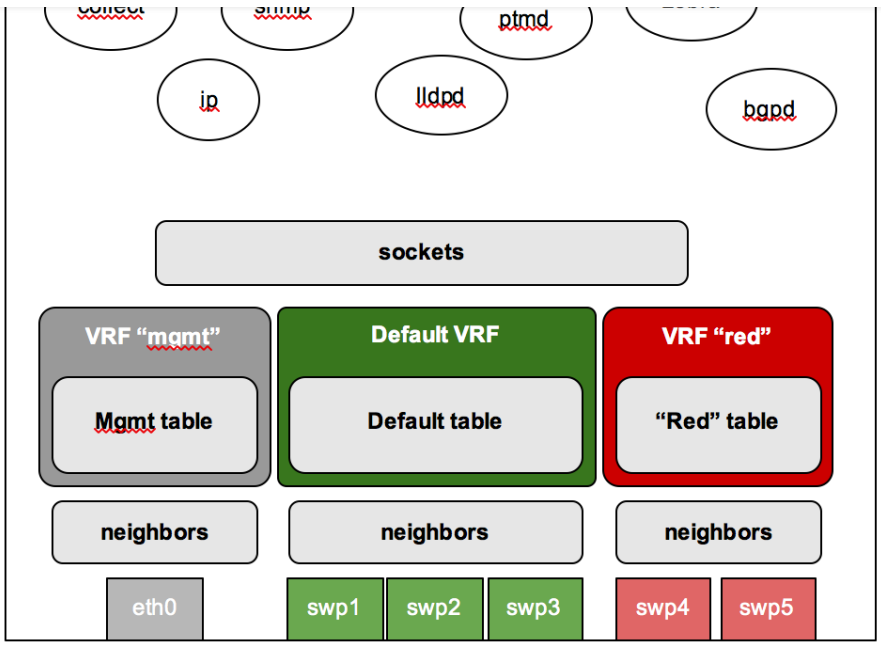
Fundamentally, this approach is missing the ability to tie network interfaces into L3 domains with a programmatic API.

2008 and "matured" in 2009. Since then the response to queries about VRF for Linux was to use a network namespace. While a network namespace does provide some of the properties needed for VRF, a namespace is the wrong construct for a VRF. Network namespaces were designed to provide a complete isolation of the entire network stack — devices, network addresses, neighbor and routing tables, protocol ports and sockets. Everything networking related is local to a network namespace, and tasks within Linux can be attached to only one network namespace at a time.



Network Namespaces Provide Total Stack Segmentation.

VRF on the other hand is a network layer feature and as such should really only impact FIB lookups. While the isolation provided by a namespace includes the route tables, a network namespace is much more than that, and the 'more than that' is the overhead that has significant impact on the software architecture and the usability and scalability of deploying VRF as a network namespace.



VRF is a Layer 3 Segmentation

Let's walk through a few a simple examples to highlight what I mean about impact on the software architecture and operational model of a network namespace as VRF.

Because a process in Linux is limited to a single network namespace, it will only see network devices, addresses, routes, even networking related data under /proc and /sys local to that namespace. For example, if you run 'ip link list' to list network devices, it will only show the devices in the namespace in which it is run. This command is just listing network interfaces, not transmitting packets via the network layer or using network layer sockets. Yet, using a namespace as a VRF impacts the view of all network resources by all processes.

A NOS is more than just routing protocols and programming hardware. You need supporting programs such as lldpd for verifying network connectivity, system monitoring tools such as snmpd, tcollector and collectd and debugging tools like 'netstat', 'ip', 'ss'. Using a namespace as a VRF has an adverse impact on all of them. An lldpd instance can only use the network devices in the namespace in which lldpd runs. Therefore, if you use a network namespace for a VRF, you have to run a separate instance of lldpd for each VRF. Deploying N-VRFs means starting N-instances of lldpd. VRF is a layer 3 feature, yet the network

That limitation applies to system monitoring applications such as snmpd, tcollector and collectd as well. For these tools to list and provide data about the networking configuration, statistics or sockets in a VRF they need a separate instance per VRF. N-VRFs means N-instances of the applications with an additional complication of how the data from the instances are aggregated and made available via the management interface.

And these examples are just infrastructure for a network OS. How a VRF is implemented is expected to impact network layer routing protocols such as quagga/bgp. Modifying them to handle the implementation of a VRF is required. But even here the choice is either running N-versions of bgpd or modifying bgpd to open a listen socket for each VRF which means N-listen sockets. As N scales into the 100's or 1000's this is wasted resources spinning up all of these processes or opening listen sockets for each VRF.

Yes, you could modify each of the code bases to be namespace aware. For example, as a VRF (network namespaces) is created and destroyed the applications either open socket local to the namespace, spawn a thread for the namespace or just add it to a list of namespaces to poll. But there are practical limitations with this approach – for example the need to modify each code base and work with each of the communities to get those changes accepted. In addition, it still does not resolve the N-VRF scalability issue as there is still 1 socket or 1 thread per VRF or the complexity of switching in and out namespaces. Furthermore, what if a network namespace is created for something other than a VRF? For example, a container is created to run an application in isolation, or you want to create a virtual switch with a subset of network devices AND within the virtual switch you want to deploy VRFs? Now you need a mechanism to discriminate which namespaces are VRFs. This option gets complicated quick.

And then there is consideration of how the VRF implementation maps to hardware for offload of packet forwarding.

I could go on, but hopefully you get my point: The devil is in the details. Many people and many companies have tried using network

tears, but it really is forcing a design that was not meant to be. Network namespaces are great for containers where you want strict isolation in the networking stack. Namespaces are also a great way to create virtual switches, carving up a single physical switch into multiple smaller and distinct logical switches. But network namespaces are a horrible solution when you just want layer 3 isolation.

Networking vendors and proprietary solutions

To date, traditional networking vendors have solved the VRF challenge in their own way, most notably by leveraging user space networking stacks and/or custom kernel patches. These proprietary solutions may work for their closed systems, but the design choice does not align with open networking and the ability to use the variety of open source tools. Even though Linux is the primary OS used by many of these vendors, they have to release SDKs for third party applications. Software that otherwise runs on millions of Linux devices and is written to standard POSIX/Linux APIs has to be modified or hooked with preloaded libraries to run in these proprietary networking environments.

As a company committed to Open Networking, Cumulus Networks wanted a proper solution for VRF not just for Cumulus Linux but for Linux in general. We wanted a common design and implementation across all devices running Linux, including network switches and servers running Routing on the Host.

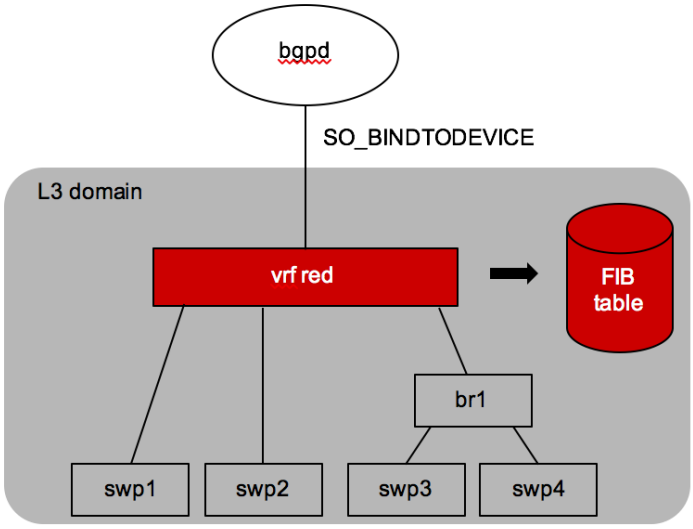
The missing piece

As mentioned earlier, the Linux networking stack supports multiple FIB tables, and it has most of what is needed to create a VRF. What it lacked (until the recent work) was a formal, programmatic construct to make it complete — some glue to bring the existing capabilities together with a consistent API and in a way that only impacts applications that use Layer 3 networking.

In early 2015, our VP of engineering, Shrijeet Mukherjee, had an idea: why not model a VRF using a netdevice that correlates to a FIB table? Netdevices (or netdevs) are a core construct in the Linux networking stack, serving as an anchor for many features — firewall rules, traffic shaping, packet captures and of course network addresses and neighbor entries. Enslaving network interfaces to the VRF device makes

enslaving devices to make them a part of the bridge domain, so enslaving interfaces to a VRF device has a similar operational model. In addition, the VRF device can be assigned VRF-local loopback addresses for routing protocols, and as a netdevice applications can use well known and established APIs (SO_BINDTODEVICE or IP_PKTINFO) to bind sockets to the VRF domain or to learn the VRF association of an incoming connection.

In short, the VRF device provides the glue for the existing capabilities to create L3 domains and without impacting the rest of the networking stack and without the need to introduce new APIs.



Getting it done

Cumulus engineers worked with the Linux networking community to get the design and code accepted into the kernel and support added to administration tools such as iproute2, libnl3 and the ifupdown2 interface manager. The result of this effort is a design and implementation that is both resource efficient and operationally efficient. It follows existing Linux networking paradigms from an operational perspective (enslaving devices to the VRF) and for administration, monitoring and troubleshooting (e.g., use of iproute2 commands).

A feature was added to allow listen sockets not bound to a specific VRF device (ie., it has global scope within the namespace) to take incoming

connection occurs. Combined with existing POSIX APIs (SO_BINDTODEVICE and IP_PKTINFO) applications can learn the L3 domain (VRF) of connected sockets. This gives users and architects a choice: bind a socket per VRF or use a 'VRF any' socket. Either way it allows a single process instance to efficiently provide service across all VRFs.

With the VRF as a device approach, enslaving a network interwork to a VRF domain only affects layer 3 FIB lookups, so there is no impact to layer 2 applications or the ability for system monitoring tools to work across all VRFs within a network namespace. Finally, by using existing APIs, some commands (e.g., ping -I and traceroute -i) already have VRF support. And of course as a device VRFs nest within network namespaces allowing VRFs within containers and virtual switches.

Open networking and Linux

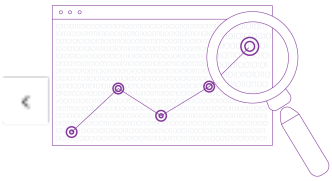
Now, 20 months later, OS distributions such as Ubuntu 16.04 and 16.10, and Debian Stretch are rolling out VRF support in their kernels and user space components. With this built-in implementation, deploying VRF is standardized across devices using Linux as the OS. You can use the same commands to configure VRFs on your host OS as you do on your switches running Cumulus Linux. Software that supports the bind-to-device API is VRF aware, and the same software can run on hosts, servers and switches.

That is the power of Open Networking and Linux.

"For technical documentation on how to configure VRF in user-space on Cumulus Linux see our technical documentation.

By David Ahern | Technology | 0 Comments

About the author: David Ahern



:ack ar

Data center network monitoring best practices part 3: Modernizing tooling

05.16.2017 | 0 Comments

Announcing Cumulus Linux 3.3: Buffer monitoring, PIM-SSM + more!

05.02.2017 | 0 Comments

Shari upstr

04.22.2

Leave A Comment

Comment...

*Please note that by leaving a comment on our blog, you are agreeing to our blog comment policy.

Name (required) Email (required) Website

POST COMMENT