Network Interface Configuration on a Linux NOS

Roopa Prabhu

Cumulus Networks, Mountain View, CA, USA, roopa@cumulusnetworks.com

Abstract

Network interface configuration is a key component of provisioning operating systems. A network interface manager carries the responsibility of configuring network interfaces. In the most basic case its job is to bring a single network port up. Network Interface configuration is tailored to an operation system distributions needs. Most Linux network interface managers use the Linux networking API and native networking tools. Yet, every distribution maintains its own Network Interface Manager optimized for its user-base. We believe there should be a single universal way to configure networking across all Linux distributions. In this talk we will see an example of a Linux server network interface manager put to use on a Linux network operating system running on a switch and router. We will show examples of network interface configuration on a Linux network switch with standard Linux networking API and linux native tools. We use ifupdown2 [3] a variant of Debian's [1] network interface manager ifupdown [2] for examples. With ifupdown2 we have tried to unify network interface configuration on Linux server and network operating system distributions.

Keywords

Netlink, Rtnetlink, iproute2

Introduction

The goal of a Network interface manager is to make network interface management painless and easy. Examples of network interface configuration include dhcp, address, link settings, bridge, stp, tunnels, vlans.

The Linux kernel understands two types of network interfaces: physical and logical. Physical interfaces represent real hardware and are owned by the device driver that manages the device. Example of physical interfaces include switch ports. Logical or virtual interfaces are created and managed by the kernel. Examples of logical interfaces include bonds, bridges, vlan, vxlan interfaces etc. Linux network interfaces are often stacked i.e they exhibit a master slave dependency relationship. Example of stacked network interfaces includes bridge and its ports. Understanding dependency relationships of Linux stacked devices is critical to a Linux network interface manager.

The Linux kernel provides netlink, ioctl and sysfs based APIs to configure networking [15, 16]. Existing native Linux tools like iproute2, brctl use one or more of these kernel APIs to configure network interfaces.

Most Linux distributions (such as Debian) provide their own network interface managers. Network interface managers operate either above the native Linux tools or use the kernel networking API directly to provide a unified way to configure all types of network interfaces.

Characteristics of Network Interface configuration: Desktop and mobile Linux distributions optimize their network interface manager for wireless and changing networks. Optimizations include automatically connecting to networks, connection retries and in most cases switch networks seamlessly.

Hypervisor and Container Linux distributions are optimized for dynamic network configurations with tenants coming and going away. Networking parameters and attributes are attached to the container or vm configuration and provisioned along with the vm or containers.

Linux Network operating systems are optimized mostly for static network interface configuration, scale and ability to make changes to parts of the network interface deployment without disrupting the state of the rest of the network.

Linux as a Network Operating System:

Network devices like switches and routers are now seeing the Linux revolution the compute world saw a few decades ago.

Not so long ago these network devices were black boxes running some version of Linux with closed vendor modifications. Automation was difficult or almost impossible. With the datacenter scale today, automation of network devices has become a necessity. Linux with its management and programability and openness has proven to be an outstanding Linux distribution for compute scale. Linux natively supports all of the networking features that an enterprise grade data-center networking gear needs and has a rich networking API used on compute nodes for a long time.

Network Interface configuration on Linux network operating system:

With network switches running Linux server distributions [17], Linux network interface managers are now subject to specific requirements of a network operating system. As data center have grown, the complexity of configuration of network switches has grown. There is a growing demand for network automation and automated provisioning of network devices in the data center.
Requirements of a Network interface manager on a Network operating system:

• Network Interface Configuration dependency handling: Linux network interfaces are often stacked which causes dependency relationships between the stacked network interfaces. A network interface manager should handle the dependencies seamlessly.
• Least service disruption: Network switches often need incremental updates to interface attributes with limited disruption.
• Scale: With the advent of server virtualization, endpoints that a network device sees has multiplied. Network switch ASICs can do more today than yesterday. As network interface configuration complexity and scale increases, the configuration specification increases and there is often repetition of configuration, resulting in configurations that are hard to manage easily. The format would benefit from a concise, repeatable interface definition standard. In addition, most automation/orchestration tools prefer a standard format (XML/JSON) for network interface specification.
• Network interface configuration validation: With interface configuration at scale comes the requirement for querying and validating running interface configuration for troubleshooting and operational efficiency.
• Network interface configuration policies: provide a way to customize configuration.

**ifupdown2**

ifupdown2 [3] is a rewrite of Debian's network interface manager ifupdown [2]. It solves some of the limitations of ifupdown but provides the same user-interface for network interface configuration on a Debian based Linux server and network switch operating system distributions. Like ifupdown, Ifupdown2 allows for pluggable network interface configuration modules which keeps it extensible and makes it easier to customize it to a specific Linux distributions needs.

Ifupdown2 was designed for data center network operating systems scale by providing simple ways to templatize repetitive config.

```bash
# ifupdown2 template example
# configure swp1 as a trunk port carrying 200 vlans
%for v in range(1, 101):
  auto swp1.$v
  iface swp1.$v
%endfor
```

Throughout the rest of this paper we provide ifupdown2 configuration examples. All examples used are from a network operating system on a data center network switch. But they do work on any Linux distribution with the right kernel, netlink [7] and iproute2 [12] support. We also provide example policies where applicable.

**Physical port and L2 link attributes**

We start at the lowest level applying physical port attributes on a network switch port: Just like on a server network port, physical port configuration on a network switch port includes speed, duplex, autoneg, mtu settings.


```bash
# ifupdown2 example for physical attribute settings on a network switch port
auto swp1
iface swp1
  link-speed 10000
  link-duplex full
  link-autoneg off
  mtu 9000
  hwaddress 00:02:0a:0b:0c:0d
```

**Attributes controlled by policy**: policy attributes can be used to force a port to a default speed, duplex, autoneg settings. Some policy decisions can also depend on the type of cable plugged into the physical port.

**L3 interface attributes**

Layer3 interface attributes on a network interface include addresses and static routes.

Configuration using Ifupdown2: ifupdown2 supports configuring addresses on a port using standard kernel netlink api or tools such as iproute2. Static routes can be added using pre and post hooks to any interface.

```bash
# iproute2 example showing l3 attributes on an network switch port
auto swp1
iface swp1
  address 10.99.1.1/30
  post-up ip route add 10.1.2.0/24 via 10.99.1.2
```
Bonding or Link Aggregation

The Linux bonding driver provides a method for aggregating multiple network interfaces. Link aggregation allows one or more links to be aggregated together to form a link aggregation group (LAG). Benefits of link aggregation include: linear scaling of bandwidth as links are added to LAG, load balancing and Failover protection.

Configuration using Ifupdown2: ifupdown2 uses a mix of netlink, iproute2 and sysfs API to configure link aggregation.

```
auto bond1
iface bond1
  bond-slaves swp1 swp2
  bond-mode 802.3ad
  bond-lacp-rate 1
  bond-mimon 100
  bond-min-links 1
  bond-use-carrier 1
  bond-xmit_hash_policy layer3+4
```

Attributes controlled by policy: restrict bond modes to network switch hardware link aggregation modes

Bridging

Ethernet bridges provide a means for hosts to communicate at layer 2. The Linux bridge driver can function in vlan filtering and non-vlan filtering modes. In this paper we only talk about the vlan filtering bridge. In a vlan filtering bridge config, one configures vlans directly on the bridge. A vlan participating in l3 is configured by a vlan device on top of the bridge.

Configuration using Ifupdown2: ifupdown2 uses a mix of netlink, iproute2 and sysfs API to configure a Linux bridge.

```
# ifupdown2 example for bridging on a network switch:
auto bridge.10
iface bridge.10
```

STP

Spanning tree is always recommended in l2 networks to prevent bridge loops and broadcast radiation. Here, we use mstpd [13] a user-space STP implementation. Mstpd is controlled by the mstpctl tool to configure stp. Mstpd works with the kernel bridge driver via netlink to run the STP protocol in user-space. Mstpd propagates bridge port stp states to the kernel bridge driver using netlink and it also reacts to kernel bridge and link netlink notifications.

Configuration using Ifupdown2: ifupdown2 talks to mstpd using mstpctl.

```
# ifupdown2 example for stp configuration on a bridge
auto bridge
iface bridge
  bridge-vlan-aware yes
  bridge-ports swp1 swp2 swp3
  bridge-stp on
```

IGMP snooping

Linux kernel bridge driver supports IGMP and MLD snooping [14]. The bridge driver snoops IGMP v1/v2/v3 reports received on a bridge port to identify hosts interested in the multicast traffic destined to a group. An IGMP query message received on a port
is used to identify the port that is connected to a router and is interested in receiving multicast traffic. The bridge driver stores all the information learnt via snooping in a mdb database which is available to user-space using netlink.

**Configuration using Ifupdown2:** IGMP snooping attributes are configured under the bridge stanza. Ifupdown2 uses iproute2 or netlink directly to send the igmp snooping attributes to the bridge driver.

```bash
# ifupdown2 example for igmp snooping attributes
auto br0
iface br0 inet static
     bridge-ports swp1 swp2 swp3
     bridge-mcrouter 1
     bridge-mcsnoop 1

auto bridge
iface bridge
     bridge-vlan-aware yes
     bridge-ports swp1 vxlan1000
     bridge-vids 1000
```

**VXLAN Tunnel Endpoints**

Vxlan tunnel endpoints (VTEP) are entities that originate and/or terminate Vxlan tunnels [4]. Linux kernel has Vxlan support and can be configured as a VTEP [5]. A first hop switch configured as a VTEP maps connected end devices to vxlan segments.

We use the Linux bridge to map traffic between the local endpoint vlan to a vxlan segment. As shown in the fig below, this is achieved by two interfaces: a local interface connecting the end device and a vxlan device enslaved in a bridge. The vlan to vxlan mapping is achieved by configuring the vlans on the local port and the vxlan device.

**Configuration using Ifupdown2:** Like any other logical devices created via ifupdown2, vxlan devices are identified by the key attribute vxlan-id. In the below ifupdown2 example, the bridge maps vlan 1000 on swp1 to vxlan1000. swp1 is configured as a trunk port possibly carrying other vlans and vxlan1000 is configured with pvid 1000.

```bash
# ifupdown2 example
auto vxlan1000
iface vxlan1000
     vxlan-local-tunnelip 10.0.0.1
     vxlan-id 1000
     bridge-access 1000

auto bridge
iface bridge
     bridge-vlan-aware yes
     bridge-ports swp1 vxlan1000
     bridge-vids 1000
```

**Attributes controlled by policy:** policy attributes enforce vlan to vxlan mapping via the pvid configuration

**Virtual Redundant routers**

VRR provides virtualized router redundancy. It enables hosts to communicate with any redundant router without needing to be reconfigured. The Figure below shows a basic VRR deployment. A bridge connects all the local end-point devices. A vlan sub interface on the bridge acts as a switched virtual interface or a layer3 interface for that vlan. This bridge vlan interface carries the original mac and ip for that vlan. A Linux macvlan interface on top of the bridge vlan interface carries the virtual mac and ip. The virtual mac and ip are common on both routers of a virtual redundant router pair.
Configuration using ifupdown2: A virtual mac and ip are represented by a address-virtual attribute as shown in table. The address-virtual attribute internally maps to a Linux macvlan device carrying the virtual mac and ip address.

```
# ifupdown2 example for a VRR config on a network switch
auto bridge.100
iface bridge.100
    address 192.168.0.252/24
    address-virtual 00:00:5e:00:01:01 192.168.0.254/24
auto bridge
iface bridge
    bridge-vlan-aware yes
    bridge-ports glob swp1-3
```

Virtual Routing and Forwarding
VRF allows for the presence of multiple independent routing tables working simultaneously on the same router or switch. This allows multiple network paths without the need for multiple switches. The Linux kernel natively supports VRF today [8]. The VRF is represented as a layer3 master network device with its own associated routing table. Configuring a VRF involves creating a VRF master interface, allocating a routing table and enslaving interfaces to the VRF master device [10]

Configuration using Ifupdown2: A VRF device is indicated by the key vrf-table attribute. VRF member dependencies are indicated by a VRF attribute under the slave. For easier allocation and management of routing tables for VRF’s, ifupdown2 uses iproute2 /etc/iproute2/rt tables.d/ files [9]. ifupdown2 maintains a VRF name and routing table id in /etc/iproute2/rt_tables.d/ifupdown2_vrf_map.conf file.
This enables usage of the VRF name for routing tables and VRF devices for easier queries and troubleshooting. Ifupdown2 also supports creation of management VRF to isolate management and data plane traffic. It can be created like any other VRF. We recommend creating it with the name ‘mgmt’ for special handling of management VRF connections like cleaning up active ssh sessions on tear-down.

Attributes controlled by policy:
- vrf table id reserved range: Reserving table id ranges helps a system administrator allocate kernel routing tables for various functions in the system.
- vrf max count: maps to hardware vrf limits
- Vrf helper hook scripts: user provided scripts run at creation and deletion of a vrf
- vrf close sockets on down: close active sockets bound to the vrf device
Conclusions

In this paper we have shown that network interface configuration on a network operating system distribution need not be different from that on a server running the same Linux kernel and native tools. We see huge value in unifying network configuration across Linux distributions and little value in configuring a bond differently on every Linux distribution. This will in-turn help standardize network automation modules across all devices running Linux in a data center. We have also shown that policies and dynamic configuration modules can help customize a network interface manager for a specific user-case.

References

1. Debian: https://www.debian.org/
2. Debian ifupdown: https://packages.debian.org/wheezy/ifupdown
3. Debian ifupdown2: https://packages.debian.org/sid/ifupdown2
5. Linux kernel Vxlan support: https://www.kernel.org/doc/Documentation/networking/vxlan.txt
6. Your next operating system is Linux: http://www.infoworld.com/article/2612738/networking/your-next-network-operating-system-is-linux.html
9. iproute2 support for rt_tables.d: https://patchwork.ozlabs.org/patch/548280/
11. ethtool: https://www.kernel.org/pub/software/network/ethtool/
12. iproute: https://wiki.linuxfoundation.org/networking/iproute2
13. mstpd: https://sourceforge.net/p/mstpd/wiki/Home/
Author Biography
Roopa Prabhu is part of the kernel networking group at Cumulus Networks. At Cumulus she works on networking in the Linux kernel and user-space, Network interface management and other system infrastructure areas. Her previous experience includes Linux clusters, ethernet drivers and Linux KVM virtualization platforms. She has an MS in Computer Science from the University of Southern California.