Project Calico Documentation

Release 1.4.0

Tigera, Inc.

July 25, 2016

Contents

1 What is Calico? 3
- 1.1 The Calico Data Path: IP Routing and iptables . . . 3
- 1.2 Addressing and Connectivity Overview . . . 5
- 1.3 Security Policy Model . . . 6

2 Using Calico 9
- 2.1 Calico with OpenStack . . . 9
- 2.2 Calico with Docker . . . 9
- 2.3 Alternative Felix Install with PyInstaller Bundle . . . 10
- 2.4 Using Calico to Secure Host Interfaces . . . 12
- 2.5 Next Steps . . . 18
- 2.6 Support . . . 37

3 Understanding Calico in more detail 39
- 3.1 Calico over an Ethernet interconnect fabric . . . 39
- 3.2 IP Interconnect Fabrics in Calico . . . 44
- 3.3 Calico Architecture . . . 55
- 3.4 How Calico Interprets Neutron API Calls . . . 58
- 3.5 Calico etcd Data Model . . . 61

4 Getting Involved 69
- 4.1 Mailing Lists . . . 69
- 4.2 Read the Source, Luke! . . . 69
- 4.3 Contributing . . . 70

5 Contribution Guidelines 71
- 5.1 Contributor Agreements . . . 72

6 Frequently Asked Questions 73
- 6.1 "Why use Calico?" . . . 73
- 6.2 "Does Calico work with IPv6?" . . . 73
- 6.3 "Is Calico compliant with PCI/DSS requirements?" . . . 73
- 6.4 "How does Calico maintain saved state?" . . . 74
- 6.5 "I heard Calico is suggesting layer 2: I thought you were layer 3! What's happening?" . . . 74
- 6.6 "I need to use hard-coded private IP addresses: how do I do that?" . . . 75
- 6.7 "How do I control policy/connectivity without virtual/physical firewalls?" . . . 75
- 6.8 "How does Calico interact with the Neutron API?" . . . 75

7 Future Plans 77

7.1 Overlapping IPv4 address ranges 77

8 License and Acknowledgements **81**

Note: This is the documentation for version 1.4.0 of Calico.

Calico is an open source solution for virtual networking in cloud data centers, developed by Tigera, Inc. and released under the Apache 2.0 License. You can find high level information about it on our website, and more detailed documentation here.

For the basic idea of what Calico is, and how a Calico network operates, please see here:

Note: This is the documentation for version 1.4.0 of Calico.

CHAPTER 1

What is Calico?

Calico is a new approach to virtual networking, based on the same scalable IP networking principles as the Internet. It targets data centers where most of the workloads (VMs, containers or bare metal servers) only require IP connectivity, and provides that using standard IP routing. Isolation between workloads - whether according to tenant ownership, or any finer grained policy - is achieved by iptables programming at the servers hosting the source and destination workloads.

In comparison with the common solutions that provide simulated layer 2 networks, Calico is a lot simpler, specifically in the following ways.

- Packets flowing through a Calico network do not require additional encapsulation and decapsulation anywhere. In contrast, layer 2 solutions typically require packets to be encapsulated in a tunneling protocol when travelling between host servers.
- Where permitted by policy, Calico packets can be routed between different tenants' workloads, or out to or in from the Internet, in exactly the same way as between the workloads of a single tenant. There is no need for on- and off-ramps as in overlay solutions, and hence for passing through special 'networking' or 'router' nodes that provide those ramps.
- As a consequence of those two points, Calico networks are easier to understand and to troubleshoot. Standard tools like ping and traceroute work for probing connectivity, and tcpdump and Wireshark for looking at flows - because Calico packets are just IP packets, and the same throughout the network.
- Security policy is specified (using ACLs) and implemented (iptables) in a single uniform way - making it more likely that it will actually be correct and robust. In contrast, in layer 2 solutions, effective security policy is a more complex (but also less expressive) product of the networks and security groups that are defined for each tenant, and of any virtual 'routers' that have been defined to allow passage between tenant networks.

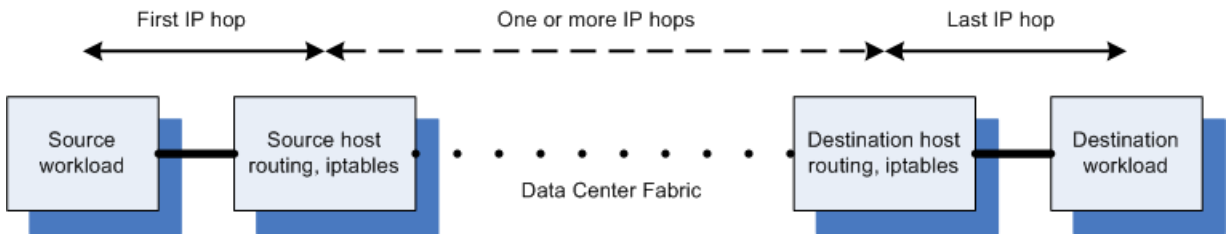
For more detail please check out the following pages.

Note: This is the documentation for version 1.4.0 of Calico.

1.1 The Calico Data Path: IP Routing and iptables

One of Calico's key features is how packets flow between workloads in a data center, or between a workload and the Internet, without additional encapsulation.

In the Calico approach, IP packets to or from a workload are routed and firewalled by the Linux routing table and iptables infrastructure on the workload's host. For a workload that is sending packets, Calico ensures that the host is always returned as the next hop MAC address regardless of whatever routing the workload itself might configure. For packets addressed to a workload, the last IP hop is that from the destination workload's host to the workload itself.



Suppose that IPv4 addresses for the workloads are allocated from a datacenter-private subnet of 10.65/16, and that the hosts have IP addresses from 172.18.203/24. If you look at the routing table on a host you will see something like this:

```
ubuntu@calico-ci02:~$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         172.18.203.1    0.0.0.0         UG    0      0        0 eth0
10.65.0.0       0.0.0.0         255.255.0.0     U     0      0        0 ns-db03ab89-b4
10.65.0.21      172.18.203.126  255.255.255.255 UGH   0      0        0 eth0
10.65.0.22      172.18.203.129  255.255.255.255 UGH   0      0        0 eth0
10.65.0.23      172.18.203.129  255.255.255.255 UGH   0      0        0 eth0
10.65.0.24      0.0.0.0         255.255.255.255 UH    0      0        0 tapa429fb36-04
172.18.203.0    0.0.0.0         255.255.255.0   U     0      0        0 eth0
```

There is one workload on this host with IP address 10.65.0.24, and accessible from the host via a TAP (or veth, etc.) interface named tapa429fb36-04. Hence there is a direct route for 10.65.0.24, through tapa429fb36-04. Other workloads, with the .21, .22 and .23 addresses, are hosted on two other hosts (172.18.203.126 and .129), so the routes for those workload addresses are via those hosts.

The direct routes are set up by a Calico agent named Felix, when it is asked to provision connectivity for a particular workload. A BGP client (such as BIRD) then notices those and distributes them – perhaps via a route reflector – to BGP clients running on other hosts, and hence the indirect routes appear also.

1.1.1 Bookended security

The routing above in principle allows any workload in a data center to communicate with any other – but in general an operator will want to restrict that; for example, so as to isolate customer A's workloads from those of customer B. Therefore Calico also programs iptables on each host, to specify the IP addresses (and optionally ports etc.) that each workload is allowed to send to or receive from. This programming is 'bookended' in that the traffic between workloads X and Y will be firewalled by both X's host and Y's host – this helps to keep unwanted traffic off the data center's core network, and as a secondary defence in case it is possible for a rogue workload to compromise its local host.

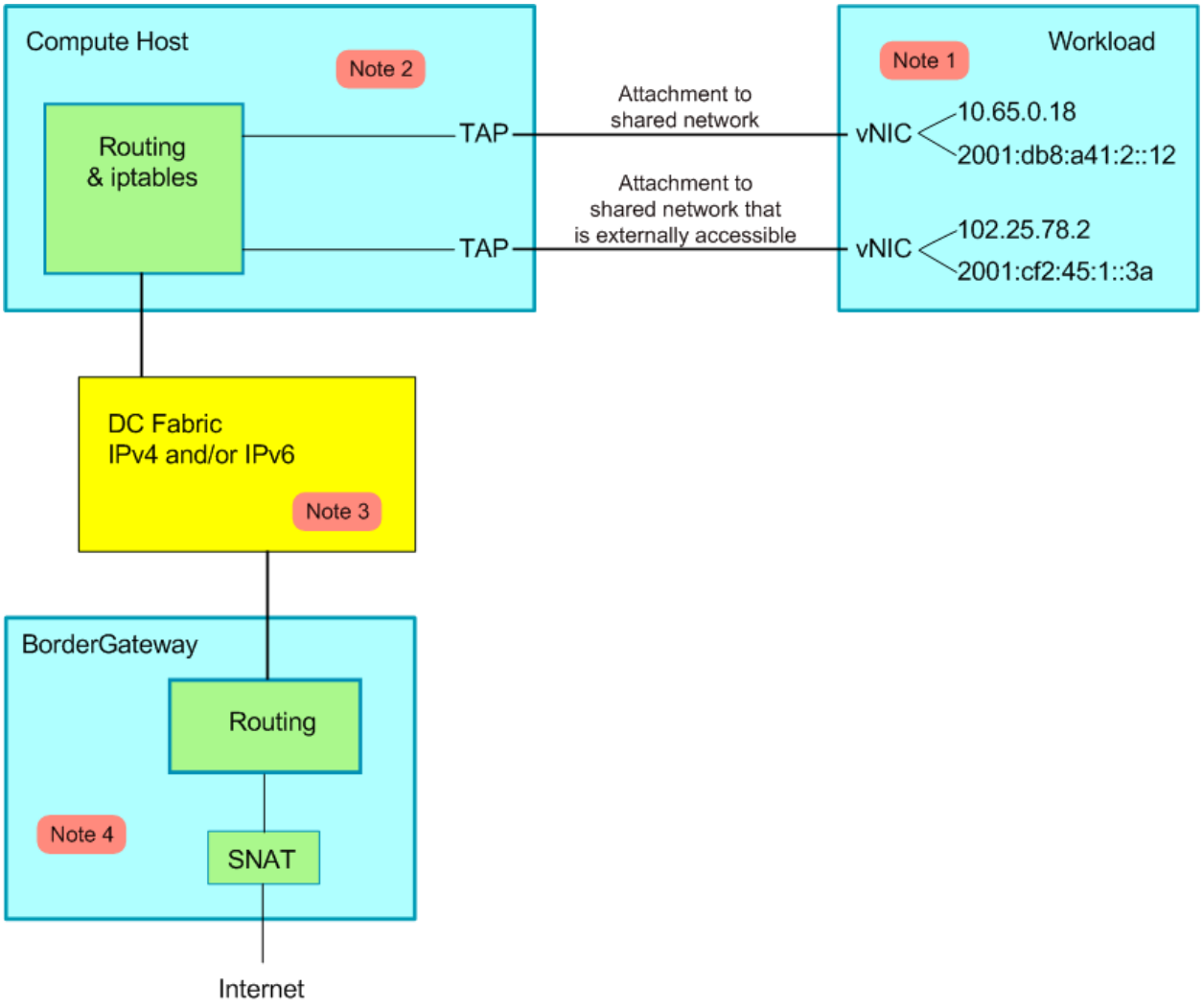
1.1.2 Is that all?

As far as the static data path is concerned, yes. It's just a combination of responding to workload ARP requests with the host MAC, IP routing and iptables. There's a great deal more to Calico in terms of how the required routing and security information is managed, and for handling dynamic things such as workload migration – but the basic data path really is that simple.

Note: This is the documentation for version 1.4.0 of Calico.

1.2 Addressing and Connectivity Overview

This diagram shows elements of a Calico network in a simplified way that allows us to focus on how IP addresses are assigned to workloads, and whether and how workloads have connectivity to and from the Internet (as well as to other workloads within the data center).



1.2.1 Network provisioning and IP addresses

In current Calico, the ranges from which IP addresses may be assigned to workloads are all provisioned by the data center operator. These ranges are all shared, in the sense that IP addresses from them might be allocated to workloads for different tenants, and the overall IPv4 and IPv6 address spaces are 'flat', in that any IPv4 address can in principle - meaning subject to security policy - communicate with any other IPv4 address, and similarly for IPv6.

Note: A future Calico release will add private address spaces with IPv4 address ranges that can be provisioned by tenants, and potentially overlap with each other and with the shared IPv4 address space. See Overlapping IPv4 address ranges for our thoughts on how that will work. (Private networks do not need IPv6 address ranges, because the shared

IPv6 address space is already sufficient for all needs.)

Within the shared address spaces there may be ranges of IP addresses (both v4 and v6) that are routable from outside the data center. Other shared network IP address ranges will not be routable from outside: they are potentially accessible from all other workloads within the data center, but not from the Internet. This is all under the control of the data center operators, as it is they who provision the shared network.

1.2.2 Outbound and inbound connectivity to and from the Internet

Subject to security configuration, *all* forms of IP addressing can initiate *outbound* connections to outside the data center. In the IPv4 case, the border gateway must NAT the source address of outbound packets, so that responses to them are routed back to the data center.

In the diagram above, 10.65/16 and 2001:db8:a41:2/64 are IPv4 and IPv6 subnets that are not routable from outside the data center, and 102.25.78/24 and 2001:cf2:45:1/64 are subnets that *are* routable from outside. For a workload to be accessible from outside the data center, it simply needs one of its vNICs to be given an IP address from one of the externally routable ranges.

The following table summarizes the properties of the IP addresses in the diagram.

IP address	Routable from Internet?	Can access Internet?
10.65.0.18	No	Yes
2001:db8:a41:2::12	No	Yes
102.25.78.2	Yes	Yes
2001:cf2:45:1::3a	Yes	Yes

Note: This is the documentation for version 1.4.0 of Calico.

1.3 Security Policy Model

Calico applies security policy to **endpoints**. Calico policy is defined in terms of **security profiles**, which contain lists of **rules** to apply as well as sets of **tags**.

1.3.1 Endpoints

Endpoints are the TAPs, veths or other interfaces, which are attached to virtual machines or containers.

Calico applies one or more security profiles to each endpoint.

Calico always tries to fail safe: if the configuration for an endpoint is missing, or no profiles are configured, Calico will drop traffic to/from that endpoint. There is always an implicit default deny rule at the end of the list of profiles.

1.3.2 Security profiles: rules

Endpoints are configured to belong to one or more security profiles. Profiles encode the policy (i.e. which packets to allow or deny) to apply to an endpoint.

Policy is encoded as two lists of rules:

- an **inbound** list, which is traversed for packets that are going *to* the endpoint
- an **outbound** list, for packets coming *from* the endpoint

In the lists, each rule consists of a set of match criteria and an action. The match criteria include:

- protocol,
- source/dest CIDR,
- source/dest tag (see below),
- source/dest port,
- ICMP type and code.

Calico supports actions, "allow" and "deny", which immediately accept or reject the packet. Once a packet is accepted or rejected further rules are not processed.

If a packet does not match any of the rules in any of the profiles attached to an endpoint then the default is to deny traffic.

If a workload (such as a virtual machine) has multiple endpoints (for example, multiple vNICs) then each of those endpoints may belong to a different set of security profiles.

1.3.3 Security profiles: tags

Each profile also has a set of (opaque) tags attached to it. An endpoint is considered a **member** of a tag if one of its profiles contains that tag.

Profile rules may reference tags in the source and destination match criteria. Calico calculates the tag memberships dynamically, updating them as endpoints come and go and as profiles are updated. This allows for very fine-grained but also maintainable policy.

For example, an operator could add the "db-user" tag to all endpoints that are to use the database. Then, they can use a single "allow" rule in the database's inbound chain to allow connections from all current members of the "db-user" tag.

1.3.4 Differences from OpenStack

Calico represents OpenStack security groups as profiles (with a single tag containing the name of the security group). While this is a simple 1-to-1 mapping at the rule level, there are some differences between Calico and OpenStack's security models to consider:

Effective security in OpenStack is a product of the interaction between three kinds of objects: networks, routers and security groups. Calico, on the other hand, **only** uses security groups for security configuration; and networks and routers have no impact. The following subsections go into this in more detail, and discuss how these concepts map onto the Calico data model.

Networks and Routers

As discussed in *Connectivity in OpenStack*, networks and routers are not used in Calico for connectivity purposes. Similarly, they serve no security purpose in a Calico environment.

Calico can provide equivalent functionality to networks and routers using security groups. To achieve it, rather than placing all ports that need to communicate into a single network, place them all in a security group that allows ingress from and egress to the same security group.

1.3.5 Architecture

At present, the flow of security information proceeds as follows:

```
[Configuration in OpenStack or other orchestrator] -(Plugin)-> [etcd] -(Felix)-> [Programmed iptables
```

When a security group is configured, the Calico orchestrator plugin discovers the new configuration. This configuration is translated into the Calico data model and written to etcd. The Felix agent watches etcd for changes and applies the policy using the kernel's iptables and ipsets.

Keen to try out Calico for yourself? The following pages describe various options for how you can deploy and run Calico.

Note: This is the documentation for version 1.4.0 of Calico.

CHAPTER 2

Using Calico

Calico as a networking model is applicable to a multitude of cloud systems, including those that use either containers or virtual machines as their basic workload unit. Correspondingly, if you want to try out Calico, there are several options for how you can easily do that, with different systems.

Note: This is the documentation for version 1.4.0 of Calico.

2.1 Calico with OpenStack

There are many ways to try out Calico with OpenStack, because OpenStack is a sufficiently complex system that there is a small industry concerned with deploying it correctly and successfully.

You can install Calico via any of the following methods:

- the packaged install for Ubuntu 14.04 - see `ubuntu-opens-install`
- an RPM install for Red Hat Enterprise Linux 7 (RHEL 7) - see `redhat-opens-install`
- our integration of Calico with Mirantis Fuel 6.1 or 7.0 - see `fuel-integration`
- our integration with Canonical's Juju Charms - see `juju-opens-install`

Warning: The `opens-chef-install`, which we used to recommend, is now very old and it only supports Icehouse. For now, we recommend using one of the above integrations.

In all cases, you just need at least two to three servers to get going (one OpenStack controller, one OpenStack compute node and, for Mirantis Fuel, a third node to serve as the Fuel master).

Note: This is the documentation for version 1.4.0 of Calico.

2.2 Calico with Docker

Calico networking for Docker and container environments is an active area of current development. Please look at calico-containers for the latest in this area, including instructions for how to set up Calico networking for a cluster of Docker container servers.

Note: This is the documentation for version 1.4.0 of Calico.

2.3 Alternative Felix Install with PyInstaller Bundle

These instructions will take you through a first-time install of Calico's per-host daemon, Felix, using the packaged PyInstaller bundle. In contrast to the `.rpm` and `.deb` installations, the bundle has minimal dependencies on distribution-provided packages. This allows it to be installed on systems where the packaged version of Python would be too old or where some of its Python dependencies are not available.

Note: This install process is most suited to bare-metal-only installations where Felix is to be used to control policy for the host's interfaces. For OpenStack and containers there are additional daemons that need to be installed, which are not covered here.

However, since the bundle doesn't take part in the distribution's package management, the dependencies that it does have must be installed manually.

2.3.1 Prerequisites

The bundle has the following pre-requisites:

- For IPv4 support, Linux kernel v2.6.32 is required. We have tested against v2.6.32-573+. Note: if you intend to run containers, Docker requires kernel v3.10+. The kernel's version can be checked with `uname -a`.
- For IPv6 support, Linux kernel 3.10+ is required (due to the lack of reverse path filtering for IPv6 in older versions).
- glibc v2.12+
- conntrack-tools; in particular, the `conntrack` command must be available. We test against v1.4.1+. To check the version, run `conntrack --version`.
- iptables; for IPv6 support, the `ip6tables` command must be available. We test against v1.4.7+. To check the version, run `iptables --version`.
- ipset; we test against v6.11+. To check the version, run `ipset --version`.
- The conntrack, iptables and ipsets kernel modules must be available (or compiled-in).
- An etcd; v2+ cluster. We recommend running the latest stable release of etcd v2.x. To check the version, run `etcd --version`

Note: If any of the commands above fail when run with the `--version` flag then you have an old version that doesn't support reporting its version.

2.3.2 Unpack the bundle

Once you have a system with the prerequisites above, the next step is to unpack the bundle, which is distributed as a `.tgz`. We recommend installing the bundle to `/opt/`:

```
cd <directory containing downloaded bundle>
# Then, as root:
tar -xzf calico-felix.tgz -C /opt/
```

After unpacking the bundle, you should have a directory `/opt/calico-felix`, containing a binary `/opt/calico-felix/calico-felix`.

2.3.3 Create a start-up script

Felix should be started at boot by your init system and the init system **must** be configured to restart Felix if it stops. Felix relies on that behaviour for certain configuration changes.

If your distribution uses systemd, then you could use the following unit file:

```
[Unit]
Description=Calico Felix agent
After=syslog.target network.target

[Service]
User=root
ExecStartPre=/usr/bin/mkdir -p /var/run/calico
ExecStart=/opt/calico-felix/calico-felix
KillMode=process
Restart=on-failure
LimitNOFILE=32000

[Install]
WantedBy=multi-user.target
```

Or, for upstart:

```
description "Felix (Calico agent)"
author "Project Calico Maintainers <maintainers@projectcalico.org>"

start on stopped rc RUNLEVEL=[2345]
stop on runlevel [!2345]

limit nofile 32000 32000

respawn
respawn limit 5 10

chdir /var/run

pre-start script
  mkdir -p /var/run/calico
  chown root:root /var/run/calico
end script

exec /opt/calico-felix/calico-felix
```

2.3.4 Configure Felix

Optionally, you can create a file at `/etc/calico/felix.cfg` to configure Felix. The configuration file as well as other options for configuring felix (including environment variables) are described in Configuring Calico.

If etcd is not running on the local machine, it's essential to configure the `EtcdAddr` or `EtcdEndpoints` setting to tell Felix how to reach etcd.

2.3.5 Start Felix

Once you've configured Felix, start it up via your init system.

For systemd, with the above unit file installed, you could run:

```
systemctl start calico-felix
```

For upstart:

```
start calico-felix
```

2.3.6 Running Felix manually

For debugging, it's sometimes useful to run Felix manually and tell it to emit its logs to screen. You can do that with the following command:

```
FELIX_LOGSEVERITYSCREEN=INFO /opt/calico-felix/calico-felix
```

Note: This is the documentation for version 1.4.0 of Calico.

2.4 Using Calico to Secure Host Interfaces

This guide describes how to use Calico to secure the network interfaces of the host itself (as opposed to those of any container/VM workloads that are present on the host). We call such interfaces "host endpoints", to distinguish them from "workload endpoints".

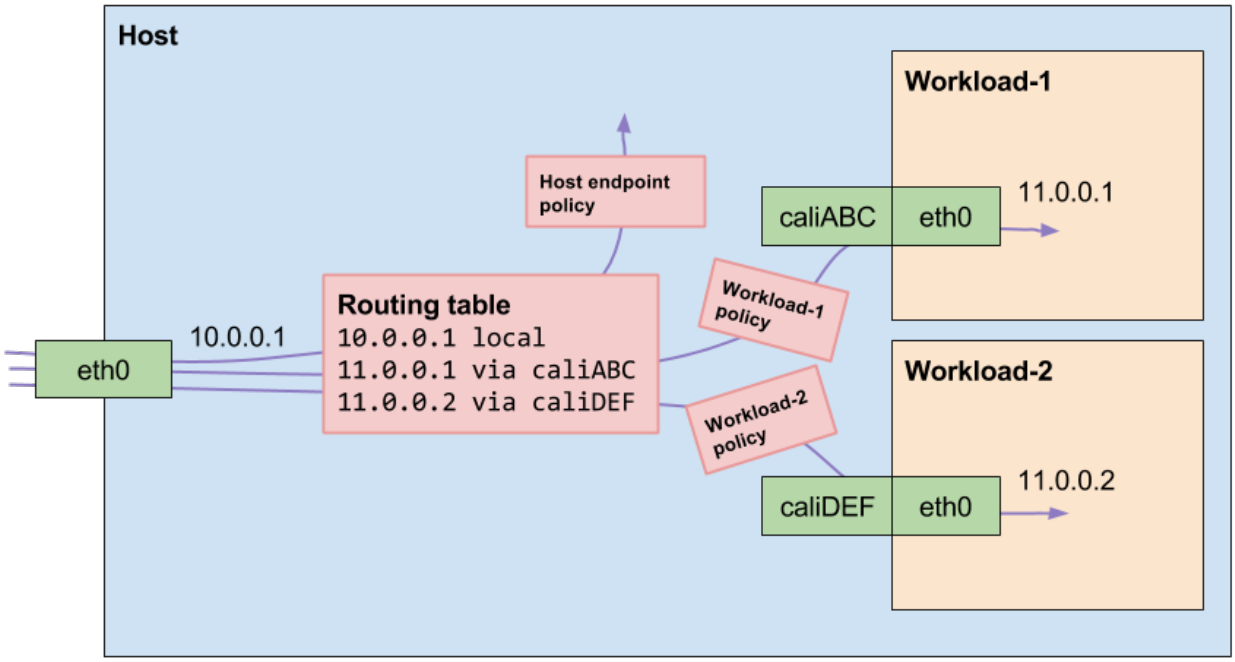
Calico supports the same rich security policy model for host endpoints that it supports for workload endpoints. Host endpoints can have labels and tags, and their labels and tags are in the same "namespace" as those of workload endpoints. This allows security rules for either type of endpoint to refer to the other type (or a mix of the two) using labels and selectors.

Calico does not support setting IPs or policing MAC addresses for host interfaces, it assumes that the interfaces are configured by the underlying network fabric.

Calico distinguishes workload endpoints from host endpoints by a configurable prefix controlled by the `InterfacePrefix` configuration value, (see: Configuring Calico). Interfaces that start with the value of `InterfacePrefix` are assumed to be workload interfaces. Others are treated as host interfaces.

Calico blocks all traffic to/from workload interfaces by default; allowing traffic only if the interface is known and policy is in place. However, for host endpoints, Calico is more lenient; it only polices traffic to/from interfaces that it's been explicitly told about. Traffic to/from other interfaces is left alone.

Note: If you have a host with workloads on it then traffic that is forwarded to workloads bypasses the policy applied to host endpoints. If that weren't the case, the host endpoint policy would need to be very broad to allow all traffic destined for any possible workload.



2.4.1 Overview

To make use of Calico's host endpoint support, you will need to follow these steps, described in more detail below:

- create an etcd cluster, if you haven't already
- install Calico's Felix daemon on each host
- initialise the etcd database
- add policy to allow basic connectivity and Calico function
- create host endpoint objects in etcd for each interface you want Calico to police (in a later release, we plan to support interface templates to remove the need to explicitly configure every interface)
- insert policy into etcd for Calico to apply
- decide whether to disable "failsafe SSH/etcd" access.

2.4.2 Creating an etcd cluster

If you haven't already created an etcd cluster for your Calico deployment, you'll need to create one.

To create a single-node etcd cluster for testing, download an etcd v2.x release from the etcd releases archive; we recommend using the most recent bugfix release. Then follow the instructions on that page to unpack and run the etcd binary.

To create a production cluster, you should follow the guidance in the etcd manual. In particular, the clustering guide.

2.4.3 Installing Felix

There are several ways to install Felix.

- if you are running Ubuntu 14.04, then you can install a version from our PPA:

```
sudo apt-add-repository ppa:project-calico/calico-<version>
sudo apt-get update
sudo apt-get upgrade
sudo apt-get install calico-felix
```

  As of writing, <version> should be 1.4.

- if you are running a RedHat 7-derived distribution, you can install from our RPM repository:

```
cat > /etc/yum.repos.d/calico.repo <<EOF
[calico]
name=Calico Repository
baseurl=http://binaries.projectcalico.org/rpm_stable/
enabled=1
skip_if_unavailable=0
gpgcheck=1
gpgkey=http://binaries.projectcalico.org/rpm/key
priority=97
EOF

yum install calico-felix
```

- if you are running another distribution, follow the instructions in Alternative Felix Install with PyInstaller Bundle to use our installer bundle.

Until you initialise the database, Felix will make a regular log that it is in state "wait-for-ready". The default location for the log file is `/var/log/calico/felix.log`.

2.4.4 Initialising the etcd database

Calico doesn't (yet) have a tool to initialise the database for bare-metal only deplyments. To initialise the database manually, make sure the `etcdctl` tool (which ships with etcd) is available, then execute the following command on one of your etcd hosts:

```
etcdctl set /calico/v1/Ready true
```

If you check the felix logfile after this step, the logs should transition from periodic notifications that felix is in state "wait-for-ready" to a stream of initialisation messages.

2.4.5 Creating basic connectivity and Calico policy

When a host endpoint is added, if there is no security policy for that endpoint, Calico will default to denying traffic to/from that endpoint, except for traffic that is allowed by the *failsafe rules*.

While the *failsafe rules* provide protection against removing all connectivity to a host,

- they are overly broad in allowing inbound SSH on any interface and allowing traffic out to etcd's ports on any interface
- depending on your network, they may not cover all the ports that are required; for example, your network may reply on allowing ICMP, or DHCP.

Therefore, we recommend creating a failsafe Calico security policy that is tailored to your environment. The example commands below show one example of how you might do that; the commands:

- Create a new policy tier called "failsafe" with `order` 0. When creating other tiers, you should give them a higher `order` value so the failsafe rules get applied first.

- Add a single policy to that tier, which
  - applies to all known endpoints
  - allows inbound ssh access from a defined "management" subnet
  - allows outbound connectivity to etcd on a particular IP; if you have multiple etcd servers you should duplicate the rule for each destination
  - allows inbound ICMP
  - allows outbound UDP on port 67, for DHCP
  - uses a "next-tier" action to pass any non-matching packets to the next tier of policy.

```
etcdctl set /calico/v1/policy/tier/failsafe/metadata '{"order": 0}'
etcdctl set /calico/v1/policy/tier/failsafe/policy/failsafe \
   '{
      "selector": "all()",
      "order": 100,

      "inbound_rules": [
        {"protocol": "tcp",
         "dst_ports": [22],
         "src_net": "<your management CIDR>",
         "action": "allow"},
        {"protocol": "icmp", "action": "allow"},
        {"action": "next-tier"}
      ],

      "outbound_rules": [
        {"protocol": "tcp",
         "dst_ports": [<your etcd ports>],
         "dst_net": "<your etcd IP>/32",
         "action": "allow"},
        {"protocol": "udp", "dst_ports": [67], "action": "allow"},
        {"action": "next-tier"}
      ]
    }'
```

Once you have such a policy in place, you may want to disable the *failsafe rules*.

Note: The policy ends with a "next-tier" action so that traffic that is not explicitly matched gets passed to the next tier of policy rather than being dropped at the end of the tier.

Note: The selector in the policy, `all()`, will match *all* endpoints, including any workload endpoints. If you have workload endpoints as well as host endpoints then you may wish to use a more restrictive selector. For example, you could label management interfaces with label `endpoint_type = management` and then use selector `endpoint_type == "management"`

Note: If you are using Calico for networking workloads, you should add inbound and outbound rules to allow BGP, for example:

```
{"protocol": "tcp", "dst_ports": [179], "action": "allow"}
```

Calico's tiered policy data is described in detail in *Tiered security policy*.

2.4.6 Creating host endpoint objects

For each host endpoint that you want Calico to secure, you'll need to create a host endpoint object in etcd. At present, this must be done manually using `etcdctl set <key> <value>`.

There are two ways to specify the interface that a host endpoint should refer to. You can either specify the name of the interface or its expected IP address. In either case, you'll also need to know the hostname of the host that owns the interface.

For example, to secure the interface named `eth0` with IP 10.0.0.1 on host `my-host`, you could create a host endpoint object at `/calico/v1/host/<hostname>/endpoint/<endpoint-id>` (where `<hostname>` is the hostname of the host with the endpoint and `<endpoint-id>` is an arbitrary name for the interface, such as "data-1" or "management") with the following data:

```
{
  "name": "eth0",
  "expected_ipv4_addrs": ["10.0.0.1"],
  "profile_ids": [<list of profile IDs>],
  "labels": {
    "role": "webserver",
    "environment": "production",
  }
}
```

Note: Felix tries to detect the correct hostname for a system. It logs out the value it has determined at start-of-day in the following format:

```
2015-10-20 17:42:09,813 [INFO][30149/5] calico.felix.config 285: Parameter FelixHostname (Felix compu
```

The value (in this case "my-hostname") needs to match the hostname used in etcd. Ideally, the host's system hostname should be set correctly but if that's not possible, the Felix value can be overridden with the FelixHostname configuration setting. See Configuring Calico for more details.

Where `<list of profile IDs>` is an optional list of security profiles to apply to the endpoint and labels contains a set of arbitrary key/value pairs that can be used in selector expressions. For more information on profile IDs, labels, and selector expressions please see Calico etcd Data Model.

Warning: When rendering security rules on other hosts, Calico uses the `expected_ipvX_addrs` fields to resolve tags and label selectors to IP addresses. If the `expected_ipvX_addrs` fields are omitted then security rules that use labels and tags will fail to match this endpoint.

Or, if you knew that the IP address should be 10.0.0.1, but not the name of the interface:

```
{
  "expected_ipv4_addrs": ["10.0.0.1"],
  "profile_ids": [<list of profile IDs>],
  "labels": {
    "role": "webserver",
    "environment": "production",
  }
}
```

The format of a host endpoint object is described in detail in Calico etcd Data Model.

After you create host endpoint objects, Felix will start policing traffic to/from that interface. If you have no policy or profiles in place, then you should see traffic being dropped on the interface.

Note: By default, Calico has a failsafe in place that whitelists certain traffic such as ssh. See below for more details on disabling/configuring the failsafe rules.

If you don't see traffic being dropped, check the hostname, IP address and (if used) the interface name in the configuration. If there was something wrong with the endpoint data, Felix will log a validation error at `WARNING` level and it will ignore the endpoint:

```
$ grep "Validation failed" /var/log/calico/felix.log
2016-05-31 12:16:21,651 [WARNING][8657/3] calico.felix.fetcd 1017:
    Validation failed for host endpoint HostEndpointId<eth0>, treating as
    missing: 'name' or 'expected_ipvX_addrs' must be present.;
    '{ "labels": {"foo": "bar"}, "profile_ids": ["prof1"]}'
```

The error can be quite long but it should log the precise cause of the rejection; in this case "'name' or 'expected_ipvX_addrs' must be present" tells us that either the interface's name or its expected IP address must be specified.

2.4.7 Creating more security policy

The Calico team recommend using tiered policy with bare-metal workloads. This allows ordered policy to be applied to endpoints that match particular label selectors.

At a minimum, you'll need to create another policy tier. If you used `order` 0 for the failsafe tier above, any higher number will do:

```
etcdctl set /calico/v1/policy/tier/my-tier/metadata '{"order": 100}'
```

Then add at least one policy to the tier. The example below allows inbound traffic from the network to endpoints labeled with role "webserver" on port 80 and all outbound traffic:

```
etcdctl set /calico/v1/policy/tier/my-tier/policy/webserver \
    '{
       "selector": "role==\"webserver\"",
       "order": 100,
       "inbound_rules": [
         {"protocol": "tcp", "dst_ports": [80], "action": "allow"}
       ],
       "outbound_rules": [
         {"action": "allow"}
       ]
     }'
```

Calico's tiered policy data is described in detail in *Tiered security policy*.

2.4.8 Failsafe rules

To avoid completely cutting off a host via incorrect or malformed policy, Calico has a failsafe mechanism that keeps various pinholes open in the firewall.

By default, Calico keeps port 22 inbound open on *all* host endpoints, which allows access to ssh; as well as outbound communication to ports 2379, 2380, 4001 and 7001, which allows access to etcd's default ports.

The lists of failsafe ports can be configured via the configuration parameters described in Configuring Calico. They can be disabled by setting each configuration value to an empty string.

Warning: Removing the inbound failsafe rules can leave a host inaccessible.
Removing the outbound failsafe rules can leave Felix unable to connect to etcd.
Before disabling the failsafe rules, we recommend creating a policy to replace it with more-specific rules for your environment: see *Creating basic connectivity and Calico policy*

Note: This is the documentation for version 1.4.0 of Calico.

2.5 Next Steps

Once you have installed Calico onto an OpenStack system, you may wish to review the Calico configuration files and make adjustments (such as to the logging targets and levels). The following article provides a reference to the available configuration options.

Note: This is the documentation for version 1.4.0 of Calico.

2.5.1 Configuring Calico

This page describes how to configure Calico. We first describe the configuration of the core Calico component – Felix – because this is needed, and configured similarly, regardless of the surrounding environment (OpenStack, Docker, or whatever). Then, depending on that surrounding environment, there will be some further configuration of that environment needed, to tell it to talk to the Calico components.

Currently we have detailed environment configuration only for OpenStack. Work on other environments is in progress, and this page will be extended as that happens.

This page aims to be a complete Calico configuration reference, and hence to describe all the possible fields, files etc. For a more task-based approach, when installing Calico with OpenStack on Ubuntu or Red Hat, please see `ubuntu-opens-install` or `redhat-opens-install`.

System configuration

A common problem on Linux systems is running out of space in the conntrack table, which can cause poor iptables performance. This can happen if you run a lot of workloads on a given host, or if your workloads create a lot of TCP connections or bidirectional UDP streams.

To avoid this becoming a problem, we recommend increasing the conntrack table size. To do so, run the following commands:

```
sysctl -w net.netfilter.nf_conntrack_max=1000000
echo "net.netfilter.nf_conntrack_max=1000000" >> /etc/sysctl.conf
```

Felix configuration

The core Calico component is Felix. (Please see Calico Architecture for the Calico architecture.)

Configuration for Felix is read from one of four possible locations, in order, as follows.

1. Environment variables.

2. The Felix configuration file.
3. Host specific configuration in etcd.
4. Global configuration in etcd.

The value of any configuration parameter is the value read from the *first* location containing a value. If not set in any of these locations, most configuration parameters have defaults, and it should be rare to have to explicitly set them.

In OpenStack, we recommend putting all configuration into configuration files, since the etcd database is transient (and may be recreated by the OpenStack plugin in certain error cases). However, in a Docker environment the use of environment variables or etcd is often more convenient.

The full list of parameters which can be set is as follows.

Setting	Default	Meaning
EtcdAddr	localhost:4001	The location (IP / hostname and port) of the etcd node or proxy that Felix should connect to.
EtcdScheme	http	The protocol type (http or https) of the etcd node or proxy that Felix connects to.
EtcdKeyFile	None	The full path to the etcd public key file, as described in *Using TLS to encrypt and authenticate communication with etcd*
EtcdCertFile	None	The full path to the etcd certificate file, as described in *Using TLS to encrypt and authenticate communication with etcd*
EtcdCaFile	"/etc/ssl/certs/ca-certificates.crt"	The full path to the etcd Certificate Authority certificate file, as described in *Using TLS to encrypt and authenticate communication with etcd*. The default value is the standard location of the system trust store. To disable authentication of the server by Felix, set the value to "none".
DefaultEndpointToHostAction	DROP	By default Calico blocks traffic from endpoints to the host itself by using an iptables DROP action. If you want to allow some or all traffic from endpoint to host then set this parameter to "RETURN" (which causes the rest of the iptables INPUT chain to be processed) or "ACCEPT" (which immediately accepts packets).
DropActionOverride	DROP	Override for the action taken when a packet would normally be dropped by Calico's firewall rules. This setting is useful when prototyping policy. Note: if the policy is set to 'ACCEPT' or 'LOG-and-ACCEPT'; Calico's security is disabled! In the current implementation, the LOG-and-XXX actions use an iptables LOG action, which logs to syslog. Should be set to one of 'DROP', 'ACCEPT', 'LOG-and-DROP', 'LOG-and-ACCEPT'.
FelixHostname	socket.gethostname()	The hostname Felix reports to the plugin. Should be used if the hostname Felix autodetects is incorrect or does not match what the plugin will expect.
MetadataAddr	127.0.0.1	The IP address or domain name of the server that can answer VM queries for cloud-init metadata. In OpenStack, this corresponds to the machine running nova-api (or in Ubuntu, nova-api-metadata). A value of 'None' (case insensitive) means that Felix should not set up any NAT rule for the metadata path.

Environment variables

The highest priority of configuration is that read from environment variables. To set a configuration parameter via an environment variable, set the environment variable formed by taking `FELIX_` and appending the uppercase form of the variable name. For example, to set the etcd address, set the environment variable `FELIX_ETCDADDR`. Other examples include `FELIX_ETCDSCHEME`, `FELIX_ETCDKEYFILE`, `FELIX_ETCDCERTFILE`, `FELIX_ETCDCAFILE`, `FELIX_FELIXHOSTNAME`, `FELIX_LOGFILEPATH` and `FELIX_METADATAADDR`.

Configuration file

On startup, Felix reads an ini-style configuration file. The path to this file defaults to `/etc/calico/felix.cfg` but can be overridden using the `-c` or `--config-file` options on the command line. If the file exists, then it is read (ignoring section names) and all parameters are set from it.

etcd configuration

Note: etcd configuration cannot be used to set either EtcdAddr or FelixHostname, both of which are required before the etcd configuration can be read.

---

etcd configuration is read from etcd from two places.

1. For a host of FelixHostname value `HOSTNAME` and a parameter named `NAME`, it is read from `/calico/v1/host/HOSTNAME/config/NAME`.
2. For a parameter named `NAME`, it is read from `/calico/v1/config/NAME`.

Note that the names are case sensitive.

OpenStack environment configuration

When running Calico with OpenStack, you also need to configure various OpenStack components, as follows.

Nova (/etc/nova/nova.conf)

Calico uses the Nova metadata service to provide metadata to VMs, without any proxying by Neutron. To make that work:

- An instance of the Nova metadata API must run on every compute node.
- `/etc/nova/nova.conf` must not set `service_neutron_metadata_proxy` or `service_metadata_proxy` to `True`. (The default `False` value is correct for a Calico cluster.)

Neutron server (/etc/neutron/neutron.conf)

In `/etc/neutron/neutron.conf` you need the following settings to configure the Neutron service.

Setting	Value	Meaning
core_plugin	neutron.plugins.ml2.plugin.Ml2Plugin	Use ML2 plugin

With OpenStack releases earlier than Liberty you will also need:

Setting	Value	Meaning
dhcp_agents_per_network	9999	Allow unlimited DHCP agents per network

Optionally – depending on how you want the Calico mechanism driver to connect to the Etcd cluster – you can also set the following options in the `[calico]` section of `/etc/neutron/neutron.conf`.

Setting	Default Value	Meaning
etcd_host	localhost	The hostname or IP of the etcd node/proxy
etcd_port	4001	The port to use for the etcd node/proxy

ML2 (.../ml2_conf.ini)

In `/etc/neutron/plugins/ml2/ml2_conf.ini` you need the following settings to configure the ML2 plugin.

Setting	Value	Meaning
mechanism_drivers	calico	Use Calico
type_drivers	local, flat	Allow 'local' and 'flat' networks
tenant_network_types	local, flat	Allow 'local' and 'flat' networks

DHCP agent (.../dhcp_agent.ini)

With OpenStack releases earlier than Liberty, in `/etc/neutron/dhcp_agent.ini` you need the following setting to configure the Neutron DHCP agent.

Setting	Value	Meaning
interface_driver	RoutedInterfaceDriver	Use Calico's modified DHCP agent support for TAP interfaces that are routed instead of being bridged

If you're going to run Calico in production, you should also review the guide to securing Calico:

Note: This is the documentation for version 1.4.0 of Calico.

2.5.2 Securing Calico

What Calico does and does not provide

Currently, Calico implements security policy that ensures that:

- a workload endpoint cannot spoof its source address
- all traffic going to an endpoint must be accepted by the inbound policy attached to that endpoint
- all traffic leaving an endpoint must be accepted by the outbound policy attached to that endpoint.

However, there are several areas that Calico does not currently cover (we're working on these and we'd love to hear from you if you're interested!). Calico does not:

- prevent an endpoint from probing the network (if its outbound policy allows it); in particular, it doesn't prevent an endpoint from contacting compute hosts or etcd
- prevent an endpoint from flooding its host with DNS/DHCP/ICMP traffic
- prevent a compromised host from spoofing packets.

Since the outbound policy is typically controlled by the application developer who owns the endpoint (at least when Calico is used with OpenStack), it's a management challenge to use that to enforce *network* policy.

How Calico uses iptables

Calico needs to add its security policy rules to the "INPUT", "OUTPUT" and "FORWARD" chains of the iptables "filter" table. To minimise the impact on the top-level chains, Calico inserts a single rule at the start of each of the kernel chains, which jumps to Calico's own chain.

The INPUT chain is traversed by packets which are destined for the host itself. Calico applies workloads' outbound policy in the input chain as well as the policy for host endpoints.

For workload traffic hitting the INPUT chain, Calico whitelists some essential bootstrapping traffic, such as DHCP, DNS and the OpenStack metadata traffic. Other traffic from local workload endpoints passes through the outbound rules for the endpoint. Then, it hits a configurable rule that either drops the traffic or allows it to continue to the remainder of the INPUT chain.

Presently, the Calico FORWARD chain is not similarly configurable. All traffic that is heading to or from a local endpoint is processed through the relevant security policy. Then, if the policy accepts the traffic, it is accepted. If the policy rejects the traffic it is immediately dropped.

Calico installs host endpoint outbound rules in the OUTPUT chain.

To prevent IPv6-enabled endpoints from spoofing their IP addresses, Felix inserts a reverse path filtering rule in the iptables "raw" PREROUTING chain. (For IPv4, it enables the rp_filter sysctl on each interface that it controls.)

Securing iptables

In a production environment, we recommend setting the default policy for the INPUT and FORWARD chains to be DROP and then explicitly whitelisting the traffic that should be allowed.

Securing etcd

Limiting network access to etcd

Calico uses etcd to store and forward the configuration of the network from plugin to the Felix agent. By default, etcd is writable by anyone with access to its REST interface. We plan to use the RBAC feature of an upcoming etcd release to improve this dramatically. However, until that work is done, we recommend blocking access to etcd from all but the IP range(s) used by the compute nodes and plugin.

Calico's host endpoint support (see: **:doc:'bare-metal'_**) can be used to enforce such policy.

Using TLS to encrypt and authenticate communication with etcd

Calico supports etcd's TLS-based security model, which supports the encryption (and authentication) of traffic between Calico components and the etcd cluster.

Warning: Calico's TLS support uses the Python urllib3 library. Unfortunately, we've seen issues with TLS in some of the common versions of urllib3 including the version that ships with Ubuntu 14.04 (1.7.1) and versions 1.11-1.12. Version 1.13 appears to work well.
In addition, no versions of urllib3 support using IP addresses in the `subjectAltName` field of a TLS certificate. An IP address can still be used in the common name (CN) field but this restriction prevents the creation of a certificate that contains multiple IP addresses. urllib3 does support domain name-based `subjectAltNames`, allowing for multiple domain names to be embedded in the certificate.
For more details of the restrictions, see this GitHub issue

To enable TLS support:

- Follow the instructions in the etcd security guide to create a certificate authority and enable TLS in etcd. We recommend enabling both client and peer authentication. This will enable security between Calico and etcd as well as between different nodes in the etcd cluster.

  Note: etcd proxies communicate with the cluster as peers so they need to have peer certificates.

- Issue a private key and client certificate for each Calico component. In OpenStack, you'll need one certificate for each control node and one for each compute node.
- If using OpenStack, configure each Neutron control node to use TLS:
  - Put the PEM-encoded private key and certificates in a secure place that is only accessible by the user that Neutron will run as.

    Note: Each control node should have its own private key and certificate. The certificate encodes the IP address or hostname of the owner.

  - Modify `/etc/neutron/plugins/ml2_conf.ini` to include a `[calico]` section that tells it where to find the key and certificates:

```
[calico]
etcd_key_file=<location of PEM-encoded private key>
etcd_cert_file=<location of PEM-encoded client certificate>
etcd_ca_cert_file=<location of PEM-encoded CA certificate>
```

    Note: Calico will validate the etcd server's certificate against the `etcd_host` configuration parameter. `etcd_host` defaults to "localhost". Issuing a certificate for "localhost" doesn't tie the certificate to any particular server. Therefore, even if you're connecting to the local server, you may wish to issue the certificate for the server's domain name and configure `etcd_host` to match.

  - Restart neutron-server.
- Unless your Calico system uses `calicoctl node` to install and configure Felix, configure each Felix with its own key and certificate:

  Note: In systems that use `calicoctl node` (such as Docker, Kubernetes and other container orchestrators), you should use the `calicoctl` tool to configure TLS. See the Etcd Secure Cluster document in the projectcalico/calico-containers GitHub repo for details.

- Generate a certificate and key pair for each Felix.

- Put the PEM-encoded private key and certificates in a secure place that is only accessible by the root user. For example, create a directory `/etc/calico/secure`:

```
$ mkdir -p /etc/calico/secure
$ chown -R root:root /etc/calico/secure
$ chmod 0700 /etc/calico/secure
```

Note: Each Felix-controlled node should have its own private key and certificate. The certificate encodes the IP address or hostname of the owner.

- Modify Felix's configuration file `/etc/calico/felix.cfg` to tell it where to find the key and certificates:

```
[global]
EtcdScheme = https
EtcdKeyFile = <location of PEM-encoded private key>
EtcdCertFile = <location of PEM-encoded client certificate>
EtcdCaFile = <location of PEM-encoded CA certificate>
...
```

Note: Calico will validate the etcd server's certificate against the host part of the `EtcdAddr` configuration parameter. `EtcdAddr` defaults to "localhost:4001". Issuing a certificate for "localhost" doesn't tie the certificate to any particular server. Therefore, even if you're connecting to the local server, you may wish to issue the certificate for the server's domain name and configure `EtcdAddr` to match.

- Restart Felix.

Host endpoint failsafe rules

By default for host endpoints (in order to avoid breaking all connectivity to a host) Calico whitelists ssh to and etcd traffic from the host running Felix. The filter rules are based entirely on ports so they are fairly broad.

This behaviour can be configured or disabled via configuration parameters; see Configuring Calico.

Before you can use your new Calico install, you'll need to configure the IP address ranges your VMs will use. This following article explains how to do this (in particular *Part 2: Set Up OpenStack*).

Note: This is the documentation for version 1.4.0 of Calico.

2.5.3 Connectivity in OpenStack

An OpenStack deployment is of limited use if its VMs cannot reach and be reached by the outside world. This document will explain how to configure your Calico-based OpenStack deployment to ensure that you have the desired connectivity with the outside world.

Major Differences from Standard OpenStack

If you've deployed OpenStack before you'll be thinking in terms of routers, floating IPs, and external networks. Calico's focus on simplicity means that it doesn't use any of these concepts. This section is mostly a warning: even if you think you know what you're doing, please read the rest of this article. You might be surprised!

Setting Up Connectivity

Part 0: Deciding your address ranges

For Calico, it's best to pick up to three address ranges you're going to use from the following three options. If it's possible, use all three.

The first option is an IPv6 address range, assuming you want your VMs to have IPv6 connectivity. Note that you can only use this range if your data center network can route IPv6 traffic. All IPv6 addresses should be considered 'externally reachable', so this needs to be a range that will be routed to your gateway router: ideally globally scoped.

The second option is a 'private' IPv4 range, assuming you want your VMs to have IPv4 connectivity. This is the most likely range for you to configure. This range will contain all VMs that cannot be reached by traffic that originates from outside the data center.

The third option is a 'public' IPv4 range, assuming you want your VMs to have IPv4 connectivity. This range will contain all the VMs that want to be reachable by traffic that originates from outside the data center. Make sure that traffic destined for this range from outside the data center will be routed to your gateway, or nothing will work!

The minimum requirement is one of those address ranges.

Part 1: Configuring the Fabric

Your Calico deployment will require a gateway router. In most non-trivial cases this will be a heavy-duty router, but if you're deploying a smaller network (maybe for testing purposes) and don't have access to one you can use a Linux server in the role.

The gateway router needs to be on the default route for all of your compute hosts. This is to ensure that all traffic destined to leave the data center goes via the gateway. That means that in a flat L3 topology the gateway router needs to be set as the next hop. In a more complex setup such as a multi-tier L3 topology the next hop may need to be slightly shorter, for example to a top-of-rack router, which will in turn need to route towards the gateway router.

Then, the gateway router needs to be a BGP peer of the Calico network. This could be a peer of one or more route reflectors, or in smaller topologies directly peering with the compute hosts. This is to ensure it knows the routes to all the VMs, so that it knows which way to route traffic destined for them. Instructions for configuring your gateway (and potentially BGP route reflectors) are beyond the scope of this document. If you don't know how to do this or want to know how Calico fits into your existing deployment, please get in touch on our mailing list: it is difficult to add a generic solution to this problem to this article.

If your gateway uses eBGP to advertise routes externally, you'll need to configure the BGP policy on the gateway to ensure that it does not export routes to the private IPv4 address range you configured above. Otherwise, in smaller deployments, you just need to make sure that external traffic destined for your VMs will get routed to the gateway. How you do this is outside the scope of this document: please ask for assistance on our mailing list.

Finally, configure your gateway to do stateful PNAT for any traffic coming from the IPv4 internal range. This ensures that even VMs that cannot be directly reached from the external network can still contact servers themselves, in order to do things like request software updates. Again, the actual manner in which this is configured depends on your router.

Part 2: Set Up OpenStack

In OpenStack, you want to set up two shared Neutron networks. For the first, add one IPv4 subnet containing the 'external' IPv4 range. Make sure the subnet has a gateway IP, and that DHCP is enabled. Additionally, add one IPv6 subnet containing half your IPv6 range, again with a gateway IP and DHCP enabled. Make sure this network has a name that makes it clear that it's for your 'externally accessible' VMs. Maybe even mark it an 'external' network, though that has no effect on what Calico does.

For the second network, add one IPv4 subnet containing the 'private' IPv4 range and one IPv6 subnet containing the other half of your IPv6 range, both with gateway IPs and DHCP enabled. Make sure this network has a name that makes it clear that it's for your 'private' VMs. Note that if you give this network part of your IPv6 range these VMs will all be reachable over IPv6. It is expected that all users will want to deploy in this way, but if you don't, either don't give these VMs IPv6 addresses or give them private ones that are not advertised by your gateway.

Then, configure the default network, subnet, router and floating IP quota for all tenants to be 0 to prevent them from creating more networks and confusing themselves!

A sample configuration is below, showing the networks and two of the four subnets (as they differ only in their address ranges, all other configuration is the same):

```
ubuntu@controller:~$ neutron net-list
+--------------------------------------+----------+-------------------------------------------------
| id                                   | name     | subnets
+--------------------------------------+----------+-------------------------------------------------
| 8d5dec25-a6aa-4e18-8706-a51637a428c2 | external | 54db559c-5e1d-4bdc-83b0-c479ef2a0ead 172.18.208.0
|                                      |          | cf6ceea0-dde0-4018-ab9a-f8f68935622b 2001:db8:a41
| fa52b704-7b3c-4c83-8698-244807352711 | internal | 301b3e63-5324-4d62-8e22-ed8dddd50689 10.65.0.0/16
|                                      |          | bf94ccb1-c57c-4c9a-a873-c20cbfa4ecaf 2001:db8:a41
+--------------------------------------+----------+-------------------------------------------------

ubuntu@controller:~$ neutron net-show external
+---------------------------+--------------------------------------+
| Field                     | Value                                |
+---------------------------+--------------------------------------+
| admin_state_up            | True                                 |
| id                        | 8d5dec25-a6aa-4e18-8706-a51637a428c2 |
| name                      | external                             |
| provider:network_type     | local                                |
| provider:physical_network |                                      |
| provider:segmentation_id  |                                      |
| router:external           | True                                 |
| shared                    | True                                 |
| status                    | ACTIVE                               |
| subnets                   | 54db559c-5e1d-4bdc-83b0-c479ef2a0ead |
|                           | cf6ceea0-dde0-4018-ab9a-f8f68935622b |
| tenant_id                 | ed34337f935745bb911eeb741bc4374b     |
+---------------------------+--------------------------------------+

ubuntu@controller:~$ neutron net-show internal
+---------------------------+--------------------------------------+
| Field                     | Value                                |
+---------------------------+--------------------------------------+
| admin_state_up            | True                                 |
| id                        | fa52b704-7b3c-4c83-8698-244807352711 |
| name                      | internal                             |
| provider:network_type     | local                                |
| provider:physical_network |                                      |
| provider:segmentation_id  |                                      |
| router:external           | False                                |
| shared                    | True                                 |
| status                    | ACTIVE                               |
| subnets                   | 301b3e63-5324-4d62-8e22-ed8dddd50689 |
|                           | bf94ccb1-c57c-4c9a-a873-c20cbfa4ecaf |
| tenant_id                 | ed34337f935745bb911eeb741bc4374b     |
+---------------------------+--------------------------------------+

ubuntu@controller:~$ neutron subnet-show external4
```

```
+------------------+-----------------------------------------------------+
| Field            | Value                                               |
+------------------+-----------------------------------------------------+
| allocation_pools | {"start": "172.18.208.2", "end": "172.18.208.255"} |
| cidr             | 172.18.208.0/24                                     |
| dns_nameservers  |                                                     |
| enable_dhcp      | True                                                |
| gateway_ip       | 172.18.208.1                                        |
| host_routes      |                                                     |
| id               | 54db559c-5e1d-4bdc-83b0-c479ef2a0ead                |
| ip_version       | 4                                                   |
| name             | external4                                           |
| network_id       | 8d5dec25-a6aa-4e18-8706-a51637a428c2                |
| tenant_id        | ed34337f935745bb911eeb741bc4374b                    |
+------------------+-----------------------------------------------------+

ubuntu@controller:~$ neutron subnet-show external6
+------------------+-------------------------------------------------------------------------------------+
| Field            | Value                                                                               |
+------------------+-------------------------------------------------------------------------------------+
| allocation_pools | {"start": "2001:db8:a41:2::2", "end": "2001:db8:a41:2:ffff:ffff:ffff:fffe"} |
| cidr             | 2001:db8:a41:2::/64                                                                 |
| dns_nameservers  |                                                                                     |
| enable_dhcp      | True                                                                                |
| gateway_ip       | 2001:db8:a41:2::1                                                                   |
| host_routes      |                                                                                     |
| id               | cf6ceea0-dde0-4018-ab9a-f8f68935622b                                                |
| ip_version       | 6                                                                                   |
| name             | external6                                                                           |
| network_id       | 8d5dec25-a6aa-4e18-8706-a51637a428c2                                                |
| tenant_id        | ed34337f935745bb911eeb741bc4374b                                                    |
+------------------+-------------------------------------------------------------------------------------+
```

Part 3: Start Using Your Networks

At this stage, all configuration is done! When you spin up a new VM, you have to decide if you want it to be contactable from outside the data center. If you do, give it a network interface on the 'external' network: otherwise, give it one on the 'internal' network. Obviously, a machine that originally wasn't going to be reachable can be made reachable by plugging a new interface into it on the 'external' network.

Right now we don't support address mobility, so an address is tied to a single port until that port is no longer in use. We plan to address this in the future.

The next step in configuring your OpenStack deployment is to configure security. We'll have a document addressing this shortly.

Now you've installed and configured Calico you'll want to test that it is functioning correctly. The following article describes how you can verify that Calico is functioning.

Note: This is the documentation for version 1.4.0 of Calico.

2.5.4 Calico Verification

This document takes you through the steps you can perform to verify that a Calico-based OpenStack deployment is running correctly.

Note: we have tested Calico with CirrOS as the guest VM operating system. Other operating systems may function differently. If unsure, or if you have problems, please contact the maintainers as described on the Support page.

Prerequisites

This document requires you have the following things:

- SSH access to the nodes in your Calico-based OpenStack deployment.
- Access to an administrator account on your Calico-based OpenStack deployment.

Procedure

Begin by creating several instances on your OpenStack deployment using your administrator account. Confirm that these instances all launch and correctly obtain IP addresses.

You'll want to make sure that your new instances are evenly striped across your hypervisors. On your control node, run:

```
nova list --fields host
```

Confirm that there is an even spread across your compute nodes. If there isn't, it's likely that an error has happened in either nova or Calico on the affected compute nodes. Check the logs on those nodes for more logging, and report your difficulty on the mailing list.

Now, SSH into one of your compute nodes. We're going to verify that the FIB on the compute node has been correctly populated by Calico. To do that, run the `route` command. You'll get output something like this:

```
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
default         net-vl401-hsrp- 0.0.0.0         UG    0      0        0 eth0
10.65.0.0       *               255.255.255.0   U     0      0        0 ns-b1163e65-42
10.65.0.103     npt06.datcon.co 255.255.255.255 UGH   0      0        0 eth0
10.65.0.104     npt09.datcon.co 255.255.255.255 UGH   0      0        0 eth0
10.65.0.105     *               255.255.255.255 UH    0      0        0 tap242f8163-08
10.65.0.106     npt09.datcon.co 255.255.255.255 UGH   0      0        0 eth0
10.65.0.107     npt07.datcon.co 255.255.255.255 UGH   0      0        0 eth0
10.65.0.108     npt08.datcon.co 255.255.255.255 UGH   0      0        0 eth0
10.65.0.109     npt07.datcon.co 255.255.255.255 UGH   0      0        0 eth0
10.65.0.110     npt06.datcon.co 255.255.255.255 UGH   0      0        0 eth0
10.65.0.111     npt08.datcon.co 255.255.255.255 UGH   0      0        0 eth0
10.65.0.112     *               255.255.255.255 UH    0      0        0 tap3b561211-dd
link-local      *               255.255.0.0     U     1000   0        0 eth0
172.18.192.0    *               255.255.255.0   U     0      0        0 eth0
```

You'll expect to see one route for each of the VM IP addresses in this table. For VMs on other compute nodes, you should see that compute node's IP address (or domain name) as the `gateway`. For VMs on this compute node, you should see `*` as the `gateway`, and the tap interface for that VM in the `Iface` field. As long as routes are present to all VMs, the FIB has been configured correctly. If any VMs are missing from the routing table, you'll want to verify the state of the BGP connection(s) from the compute node hosting those VMs.

Having confirmed the FIB is present and correct, open the console for one of the VM instances you just created. Confirm that the machine has external connectivity by pinging `google.com` (or any other host you are confident

is routable and that will respond to pings). Additionally, confirm it has internal connectivity by pinging the other instances you've created (by IP).

If all of these tests behave correctly, your Calico-based OpenStack deployment is in good shape.

Troubleshooting

If you find that none of the advice below solves your problems, please use our diagnostics gathering script to generate diagnostics, and then raise a GitHub issue against our repository. To generate the diags, run

```
$ /usr/bin/calico-diags
```

VMs cannot DHCP

This can happen if your iptables is configured to have a default DROP behaviour on the INPUT or FORWARD chains. You can test this by running `iptables -L -t filter` and checking the output. You should see something that looks a bit like this:

```
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
ACCEPT     all  --  anywhere             anywhere            state RELATED,ESTABLISHED
ACCEPT     icmp --  anywhere             anywhere
ACCEPT     all  --  anywhere             anywhere
ACCEPT     tcp  --  anywhere             anywhere            state NEW tcp dpt:ssh
REJECT     all  --  anywhere             anywhere            reject-with icmp-host-prohibited

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination
REJECT     all  --  anywhere             anywhere            reject-with icmp-host-prohibited

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
```

The important sections are `Chain INPUT` and `Chain FORWARD`. Each of those needs to have a policy of `ACCEPT`. In some systems, this policy may be set to `DENY`. To change it, run `iptables -P <chain> ACCEPT`, replacing `<chain>` with either `INPUT` or `FORWARD`.

Note that doing this may be considered a security risk in some networks. A future Calico enhancement will remove the requirement to perform this step.

Routes are missing in the FIB.

If routes to some VMs aren't present when you run `route`, this suggests that your BGP sessions are not functioning correctly. Your BGP daemon should have either an interactive console or a log. Open the relevant one and check that all of your BGP sessions have come up appropriately and are replicating routes. If you're using a full mesh configuration, confirm that you have configured BGP sessions with *all* other Calico nodes.

VMs Cannot Ping Non-VM IPs

Assuming all the routes are present in the FIB (see above), this most commonly happens because the gateway is not configured with routes to the VM IP addresses. To get full Calico functionality the gateway should also be a BGP peer of the compute nodes (or the route reflector).

Confirm that your gateway has routes to the VMs. Assuming it does, make sure that your gateway is also advertising those routes to its external peers. It may do this using eBGP, but it may also be using some other routing protocol.

VMs Cannot Ping Other VMs

Before continuing, confirm that the two VMs are in security groups that allow inbound traffic from each other (or are both in the same security group which allows inbound traffic from itself). Traffic will not be routed between VMs that do not allow inbound traffic from each other.

Assuming that the security group configuration is correct, confirm that the machines hosting each of the VMs (potentially the same machine) have routes to both VMs. If they do not, check out the troubleshooting section *Routes are missing in the FIB.*.

Web UI Shows Error Boxes Saying "Error: Unable to get quota info" and/or "Error: Unable to get volume limit"

This is likely a problem encountered with mapping devices in `cinder`, OpenStack's logical volume management component. Many of these can be resolved by restarting `cinder`:

```
service cinder-volume restart
service cinder-scheduler restart
service cinder-api restart
```

Cannot create instances, error log says "could not open /dev/net/tun: Operation not permitted"

This is caused by having not restarted libvirt after you add lines to the end of `/etc/libvirt/qemu.conf`. This can be fixed by either rebooting your entire system or running:

```
service libvirt-bin restart
```

If you would like your workloads to have IPv6 connectivity as well as or instead of IPv4, the following page explains how to do that.

Note: This is the documentation for version 1.4.0 of Calico.

2.5.5 IPv6 Support

Calico supports connectivity over IPv6, between compute hosts, and between compute hosts and their VMs. This means that, subject to security configuration, a VM can initiate an IPv6 connection to another VM, or to an IPv6 destination outside the data center; and that a VM can terminate an IPv6 connection from outside.

Requirements for containers

Containers have no specific requirements for utilising IPv6 connectivity.

Requirements for guest VM images

When using Calico with a VM platform (e.g. OpenStack), obtaining IPv6 connectivity requires certain configuration in the guest VM image:

- When it boots up, the VM should issue a DHCPv6 request for each of its interfaces, so that it can learn the IPv6 addresses that OpenStack has allocated for it.
- The VM must be configured to accept Router Advertisements.
- If it uses the widely deployed DHCP client from ISC, the VM must have a fix or workaround for this known issue.

These requirements are not yet all met in common cloud images - but it is easy to remedy that by launching an image, making appropriate changes to its configuration files, taking a snapshot, and then using that snapshot thereafter instead of the original image.

For example, starting from the Ubuntu 14.04 cloud image, the following changes will suffice to meet the requirements just listed.

- In `/etc/network/interfaces.d/eth0.cfg`, add:

```
iface eth0 inet6 dhcp
        accept_ra 1
```

- In `/sbin/dhclient-script`, add at the start of the script:

```
new_ip6_prefixlen=128
```

- In `/etc/sysctl.d`, create a file named `30-eth0-rs-delay.conf` with contents:

```
net.ipv6.conf.eth0.router_solicitation_delay = 10
```

Implementation details

Following are the key points of how IPv6 connectivity is currently implemented in Calico.

- IPv6 forwarding is globally enabled on each compute host.
- Felix (the Calico agent):
  - does `ip -6 neigh add lladdr dev`, instead of IPv4 case `arp -s`, for each endpoint that is created with an IPv6 address
  - adds a static route for the endpoint's IPv6 address, via its tap or veth device, just as for IPv4.
- Dnsmasq provides both Router Advertisements and DHCPv6 service (neither of which are required for container environments).
  - Router Advertisements, without SLAAC or on-link flags, cause each VM to create a default route to the link-local address of the VM's TAP device on the compute host.
  - DHCPv6 allows VMs to get their orchestrator-allocated IPv6 address.
- For container environments, we don't Dnsmasq:
  - rather than using Router Advertisements to create the default route, we Proxy NDP to ensure that routes to all machines go via the compute host.
  - rather than using DHCPv6 to allocate IPv6 addresses, we allocate the IPv6 address directly to the container interface before we move it into the container.
- BIRD6 runs between the compute hosts to distribute routes.

OpenStack Specific Details

In OpenStack, IPv6 connectivity requires defining an IPv6 subnet, in each Neutron network, with:

- the IPv6 address range that you want your VMs to use
- DHCP enabled
- (from Juno onwards) IPv6 address mode set to DHCPv6 stateful.

We suggest initially configuring both IPv4 and IPv6 subnets in each network. This allows handling VM images that support only IPv4 alongside those that support both IPv4 and IPv6, and allows a VM to be accessed over IPv4 in case this is needed to troubleshoot any issues with its IPv6 configuration.

In principle, though, we are not aware of any problems with configuring and using IPv6-only networks in OpenStack.

To update your Calico installation, please consult the following page for instructions on how to do so.

Note: This is the documentation for version 1.4.0 of Calico.

2.5.6 Upgrade Procedure (OpenStack)

This document details the procedure for upgrading a Calico-based OpenStack system. It contains the full steps for upgrading all components and the order in which that upgrade should be performed. Most releases do not concurrently upgrade all of these components: if a release does not upgrade a given component, you may skip those steps.

> **Warning:** While the upgrade procedure is very safe, you will be unable to issue API requests to your OpenStack system during the procedure. Please plan your upgrade window accordingly, and see the *Service Impact* section for more details.

Service Impact

During the upgrade, **all VMs will continue to function normally**: there should be no impact on the data plane. However, control plane traffic may fail at different points throughout the upgrade.

Generally, users should be prevented from creating or updating virtual machines during this procedure, as these actions will fail. VM deletion *may* succeed, but will likely be delayed until the end of the upgrade.

For this reason, we highly recommend planning a maintenance window for the upgrade. During this window, you should disable all user API access to your OpenStack deployment.

Upgrade Procedure

Upgrade is performed in the following stages, which must be performed in the order shown.

1: Upgrade etcd

This step should be run on every machine in your deployment that runs any Calico code, and also on the machine running the etcd cluster.

Ubuntu 14.04 Use apt-get to obtain the more recent version:

```
apt-get update
apt-get install etcd
```

Red Hat 7 Stop the etcd process:

```
systemctl stop etcd
```

Then, download the tested binary (currently 2.0.11) and install it:

```
curl -L  https://github.com/coreos/etcd/releases/download/v2.0.11/etcd-v2.0.11-linux-amd64.tar.gz -o
tar xvf etcd-v2.0.11-linux-amd64.tar.gz
cd etcd-v2.0.11-linux-amd64
mv etcd* /usr/local/bin/
```

Now, restart etcd:

```
systemctl start etcd
```

2: Upgrade compute software

On each machine running the Calico compute software (the component called Felix), run the following upgrade steps.

Uninstall pip-installed networking-calico If present, uninstall any pip-installed networking-calico package:

```
pip uninstall networking-calico
```

(networking-calico function is now installed as a Debian or RPM package instead.)

Ubuntu 14.04 First, use `apt-get` to install the updated packages. On each compute host upgrade the Calico packages, as follows:

```
apt-get update
apt-get install calico-compute calico-felix calico-common python-etcd \
                networking-calico
```

> **Warning:** Running `apt-get upgrade` is not sufficient to upgrade Calico due to new dependent packages added in version 1.3. If you want to upgrade Calico as part of a system-wide update, you must use `apt-get dist-upgrade`.

Then, restart Felix to ensure that it picks up any changes:

```
service calico-felix restart
```

Finally, if dnsmasq was upgraded, kill it and restart the DHCP agent. This is required due to an upstream problem: oslo-rootwrap can't kill a process when the binary has been updated since it started running:

```
pkill dnsmasq
```

For OpenStack Liberty or later, install the new Calico DHCP agent and disable the Neutron-provided one. The Calico DHCP agent is backed by etcd, allowing it to scale to higher numbers of hosts:

```
service neutron-dhcp-agent stop
echo manual > /etc/init/neutron-dhcp-agent.override
apt-get install calico-dhcp-agent
```

Check that only the Calico DHCP agent is now running:

```
# status calico-dhcp-agent
calico-dhcp-agent start/running, process <PID>
# status neutron-dhcp-agent
neutron-dhcp-agent stop/waiting
```

Or if you are using an earlier OpenStack release, restart the Neutron-provided DHCP agent:

```
service neutron-dhcp-agent restart
```

Red Hat 7 First, upgrade python-etcd:

```
curl -L https://github.com/projectcalico/python-etcd/archive/master.tar.gz -o python-etcd.tar.gz
tar xvf python-etcd.tar.gz
cd python-etcd-master
python setup.py install
```

Then, update packaged components:

```
yum update
```

We recommend upgrading the whole distribution as shown here. In case you prefer to upgrade particular packages only, those needed for a Calico compute node are the following.

```
calico-common
calico-compute
calico-dhcp-agent
calico-felix
dnsmasq
networking-calico
openstack-neutron
openstack-nova-api
openstack-nova-compute
```

Finally, if dnsmasq was upgraded, kill it and restart the DHCP agent. This is required due to an upstream problem: oslo-rootwrap can't kill a process when the binary has been updated since it started running:

```
pkill dnsmasq
```

For OpenStack Liberty or later, modify `/etc/neutron/neutron.conf`. In the `[oslo_concurrency]` section, ensure that the `lock_path` variable is uncommented and set as follows:

```
# Directory to use for lock files. For security, the specified directory should
# only be writable by the user running the processes that need locking.
# Defaults to environment variable OSLO_LOCK_PATH. If external locks are used,
# a lock path must be set.
lock_path = $state_path/lock
```

For OpenStack Liberty or later, install the new Calico DHCP agent and disable the Neutron-provided one. The Calico DHCP agent is backed by etcd, allowing it to scale to higher numbers of hosts:

```
systemctl stop neutron-dhcp-agent
systemctl disable neutron-dhcp-agent
yum install calico-dhcp-agent
```

Check that (only) the Calico DHCP agent is started:

```
# systemctl status calico-dhcp-agent
...
Active: active (running)
...
# systemctl status neutron-dhcp-agent
...
Active: inactive
...
```

Or if you are using an earlier OpenStack release:

```
systemctl restart neutron-dhcp-agent
```

3: Upgrade control software

On each machine running the Calico control software (every machine running neutron-server), run the following upgrade steps.

Ubuntu 14.04 First, use `apt-get` to install the updated packages. On each control host you can upgrade only the Calico packages, as follows:

```
apt-get update
apt-get install calico-control calico-common python-etcd networking-calico
```

> **Warning:** Running `apt-get upgrade` is not sufficient to upgrade Calico due to new dependent packages added in version 1.3. If you want to upgrade Calico as part of a system-wide update, you must use `apt-get dist-upgrade`.

Then, restart Neutron to ensure that it picks up any changes:

```
service neutron-server restart
```

Red Hat 7 First, upgrade python-etcd:

```
curl -L https://github.com/projectcalico/python-etcd/archive/master.tar.gz -o python-etcd.tar.gz
tar xvf python-etcd.tar.gz
cd python-etcd-master
python setup.py install
```

Then, update packaged components:

```
yum update
```

We recommend upgrading the whole distribution as shown here. In case you prefer to upgrade particular packages only, those needed for a Calico control node are the following.

```
calico-common
calico-control
networking-calico
openstack-neutron
```

Then, restart Neutron to ensure that it picks up any changes:

```
systemctl restart neutron-server
```

Note: This is the documentation for version 1.4.0 of Calico.

2.6 Support

There are three support mechanisms for Calico:

- Mailing list. This is for questions about Calico, or for asking for help.
- Issues. These are for problems with Calico. If you're not sure whether a certain behaviour is expected or not, you should ask on the mailing list first.
- Pull Requests. These allow you to contribute fixes and enhancements back to Calico.

Information about the mailing list and pull requests can be found on our Getting Involved page, as they are also how to get proactively involved in Calico development.

2.6.1 Issues

Calico uses GitHub's Issues system to track problems. To access the Issues system, first determine which project the issue is with - ask in the mailing list if you're not sure. Then go to that project's repository page in GitHub and click the Issues tab at the top.

Remember to do a search for existing issues covering your problem before raising a new one!

If you'd like to learn more about how Calico works 'under the hood', please take a look at the following.

Note: This is the documentation for version 1.4.0 of Calico.

CHAPTER 3

Understanding Calico in more detail

The following articles cover aspects of Calico's internal architecture, and also of considerations for deploying Calico at large scale.

Note: This is the documentation for version 1.4.0 of Calico.

3.1 Calico over an Ethernet interconnect fabric

3.1.1 At scale, and no, we're not joking

This is the first of a few *tech notes* that I will be authoring that will discuss some of the various interconnect fabric options in a Calico network.

Any technology that is capable of transporting IP packets can be used as the interconnect fabric in a Calico network (the first person to test and publish the results of using IP over Avian Carrier as a transport for Calico will earn a very nice dinner on or with the core Calico team). This means that the standard tools used to transport IP, such as MPLS and Ethernet can be used in a Calico network.

In this note, I'm going to focus on Ethernet as the interconnect network. Talking to most at-scale cloud operators, they have converted to IP fabrics, and as will cover in the next blog post that infrastructure will work for Calico as well. However, the concerns that drove most of those operators to IP as the interconnection network in their pods are largely ameliorated by Project Calico, allowing Ethernet to be viably considered as a Calico interconnect, even in large-scale deployments.

Concerns over Ethernet at scale

It has been acknowledged by the industry for years that, beyond a certain size, classical Ethernet networks are unsuitable for production deployment. Although there have been multiple attempts to address these issues, the scale-out networking community has, largely abandoned Ethernet for anything other than providing physical point-to-point links in the networking fabric. The principal reasons for Ethernet failures at large scale are:

1. Large numbers of *end points* [1]. Each switch in an Ethernet network must learn the path to all Ethernet endpoints that are connected to the Ethernet network. Learning this amount of state can become a substantial task when we are talking about hundreds of thousands of *end points*.

[1] In this document (and in all Calico documents) we tend to use the terms *end point* to refer to a virtual machine, container, appliance, bare metal server, or any other entity that is connected to a Calico network. If we are referring to a specific type of end point, we will call that out (such as referring to the behavior of VMs as distinct from containers).

2. High rate of *churn* or change in the network. With that many end points, most of them being ephemeral (such as virtual machines or containers), there is a large amount of *churn* in the network. That load of re-learning paths can be a substantial burden on the control plane processor of most Ethernet switches.
3. High volumes of broadcast traffic. As each node on the Ethernet network must use Broadcast packets to locate peers, and many use broadcast for other purposes, the resultant packet replication to each and every end point can lead to *broadcast storms* in large Ethernet networks, effectively consuming most, if not all resources in the network and the attached end points.
4. Spanning tree. Spanning tree is the protocol used to keep an Ethernet network from forming loops. The protocol was designed in the era of smaller, simpler networks, and it has not aged well. As the number of links and interconnects in an Ethernet network goes up, many implementations of spanning tree become more *fragile*. Unfortunately, when spanning tree fails in an Ethernet network, the effect is a catastrophic loop or partition (or both) in the network, and, in most cases, difficult to troubleshoot or resolve.

While many of these issues are crippling at *VM scale* (tens of thousands of end points that live for hours, days, weeks), they will be absolutely lethal at *container scale* (hundreds of thousands of end points that live for seconds, minutes, days).

If you weren't ready to turn off your Ethernet data center network before this, I bet you are now. Before you do, however, let's look at how Project Calico can mitigate these issues, even in very large deployments.

How does Calico tame the Ethernet daemons?

First, let's look at how Calico uses an Ethernet interconnect fabric. It's important to remember that an Ethernet network *sees* nothing on the other side of an attached IP router, the Ethernet network just *sees* the router itself. This is why Ethernet switches can be used at Internet peering points, where large fractions of Internet traffic is exchanged. The switches only see the routers from the various ISPs, not those ISPs' customers' nodes. We leverage the same effect in Calico.

To take the issues outlined above, let's revisit them in a Calico context.

1. Large numbers of end points. In a Calico network, the Ethernet interconnect fabric only sees the routers/compute servers, not the end point. In a standard cloud model, where there is tens of VMs per server (or hundreds of containers), this reduces the number of nodes that the Ethernet sees (and has to learn) by one to two orders of magnitude. Even in very large pods (say twenty thousand servers), the Ethernet network would still only see a few tens of thousands of end points. Well within the scale of any competent data center Ethernet top of rack (ToR) switch.
2. High rate of *churn*. In a classical Ethernet data center fabric, there is a *churn* event each time an end point is created, destroyed, or moved. In a large data center, with hundreds of thousands of endpoints, this *churn* could run into tens of events per second, every second of the day, with peaks easily in the hundreds or thousands of events per second. In a Calico network, however, the *churn* is very low. The only event that would lead to *churn* in a Calico network's Ethernet fabric would be the addition or loss of a compute server, switch, or physical connection. In a twenty thousand server pod, even with a 5% daily failure rate (a few orders of magnitude more than what is normally experienced), there would only be two thousand events per **day**. Any switch that can not handle that volume of change in the network should not be used for any application.
3. High volume of broadcast traffic. Since the first (and last) hop for any traffic in a Calico network is an IP hop, and IP hops terminate broadcast traffic, there is no endpoint broadcast network in the Ethernet fabric, period. In fact, the only broadcast traffic that should be seen in the Ethernet fabric is the ARPs of the compute servers locating each other. If the traffic pattern is fairly consistent, the steady-state ARP rate should be almost zero. Even in a pathological case, the ARP rate should be well within normal accepted boundaries.
4. Spanning tree. Depending on the architecture chosen for the Ethernet fabric, it may even be possible to turn off spanning tree. However, even if it is left on, due to the reduction in node count, and reduction in churn, most competent spanning tree implementations should be able to handle the load without stress.

With these considerations in mind, it should be evident that an Ethernet connection fabric in Calico is not only possible, it is practical and should be seriously considered as the interconnect fabric for a Calico network.

As mentioned in the IP fabric post, an IP fabric is also quite feasible for Calico, but there are more considerations that must be taken into account. The Ethernet fabric option has fewer architectural considerations in its design.

A brief note about Ethernet topology

As mentioned elsewhere in the Calico documentation, since Calico can use most of the standard IP tooling, some interesting options regarding fabric topology become possible.

We assume that an Ethernet fabric for Calico would most likely be constructed as a *leaf/spine* architecture. Other options are possible, but the *leaf/spine* is the predominant architectural model in use in scale-out infrastructure today.

Since Calico is an IP routed fabric, a Calico network can use ECMP to distribute traffic across multiple links (instead of using Ethernet techniques such as MLAG). By leveraging ECMP load balancing on the Calico compute servers, it is possible to build the fabric out of multiple *independent* leaf/spine planes using no technologies other than IP routing in the Calico nodes, and basic Ethernet switching in the interconnect fabric. These planes would operate completely independently and could be designed such that they would not share a fault domain. This would allow for the catastrophic failure of one (or more) plane(s) of Ethernet interconnect fabric without the loss of the pod (the failure would just decrease the amount of interconnect bandwidth in the pod). This is a gentler failure mode than the pod-wide IP or Ethernet failure that is possible with today's designs.

A more in-depth discussion is possible, so if you'd like, please make a request, and I will put up a post or white paper. In the meantime, it may be interesting to venture over to Facebook's blog post on their fabric approach. A quick picture to visualize the idea is shown below.

I am not showing the end points in this diagram, and the end points would be unaware of anything in the fabric (as noted above).

In the particular case of this diagram, each ToR is segmented into four logical switches (possibly by using 'port VLANs'), [2] and each compute server has a connection to each of those logical switches. We will identify those logical switches by their color. Each ToR would then have a blue, green, orange, and red logical switch. Those 'colors' would be members of a given *plane*, so there would be a blue plane, a green plane, an orange plane, and a red plane. Each plane would have a dedicated spine switch. and each ToR in a given spine would be connected to its spine, and only its spine.

Each plane would constitute an IP network, so the blue plane would be 2001:db8:1000::/36, the green would be 2001:db8:2000::/36, and the orange and red planes would be 2001:db8:3000::/36 and 2001:db8:4000::/36 respectively. [3]

Each IP network (plane) requires it's own BGP route reflectors. Those route reflectors need to be peered with each other within the plane, but the route reflectors in each plane do not need to be peered with one another. Therefore, a fabric of four planes would have four route reflector meshes. Each compute server, border router, *etc.* would need to be a route reflector client of at least one route reflector in each plane, and very preferably two or more in each plane.

A diagram that visualizes the route reflector environment can be found below.

These route reflectors could be dedicated hardware connected to the spine switches (or the spine switches themselves), or physical or virtual route reflectors connected to the necessary logical leaf switches (blue, green, orange, and red). That may be a route reflector running on a compute server and connected directly to the correct plane link, and not routed through the vRouter, to avoid the chicken and egg problem that would occur if the route reflector were "behind" the Calico network.

Other physical and logical configurations and counts are, of course, possible, this is just an example.

[2] We are using logical switches in this example. Physical ToRs could also be used, or a mix of the two (say 2 logical switches hosted on each physical switch).

[3] We use IPv6 here purely as an example. IPv4 would be configured similarly. I welcome your questions, either here on the blog, or via the Project Calico mailing list.

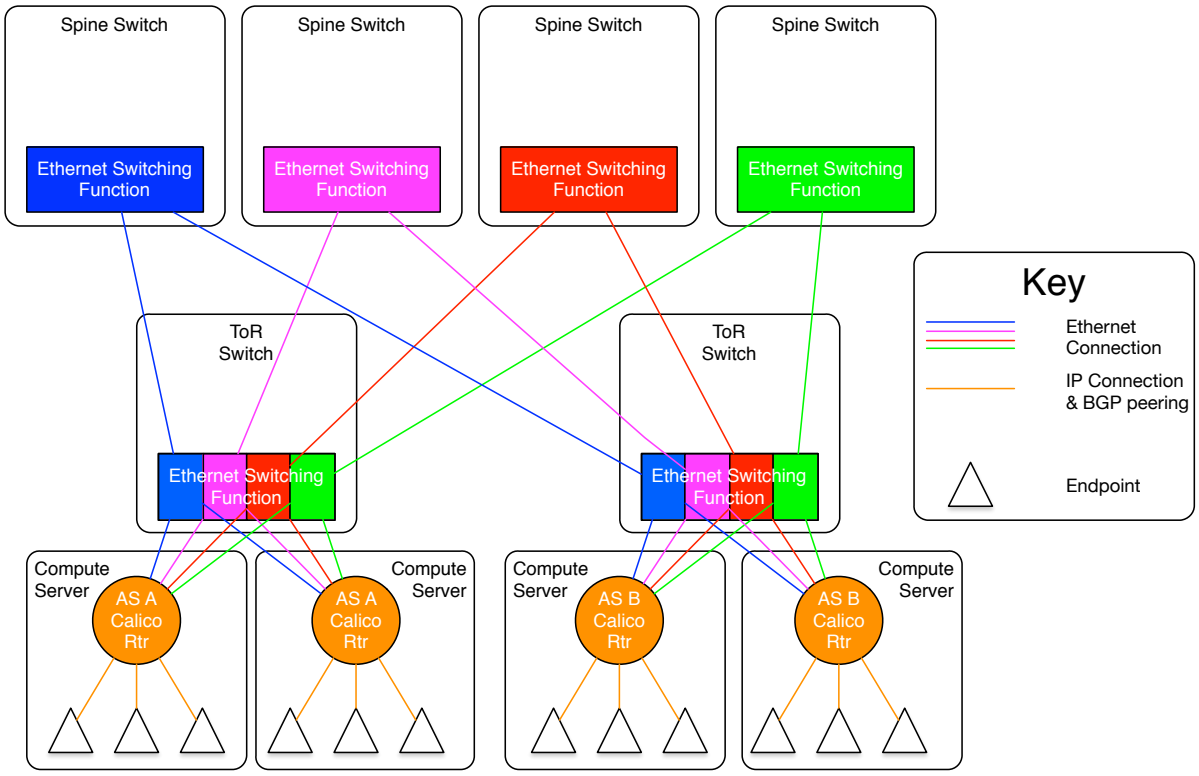


Fig. 3.1: A diagram showing the Ethernet spine planes. Each color represents a distinct Ethernet network, transporting a unique IP network.

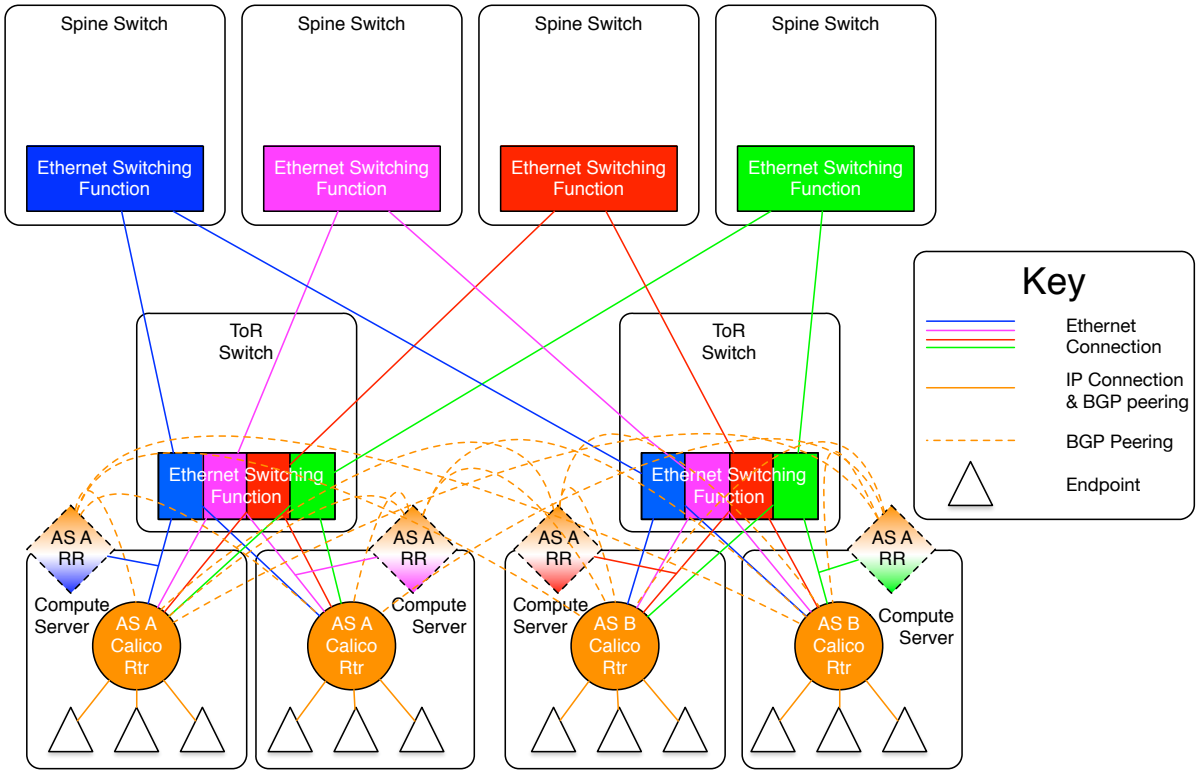


Fig. 3.2: A diagram showing the route reflector topology in the l2 spine plane architecture. The dashed diamonds are the route reflectors, with one or more per L2 spine plane. All compute servers are peered to all route reflectors, and all the route reflectors in a given plane are also meshed. However, the route reflectors in each spine plane are not meshed together (*e.g.* the *blue* route reflectors are not peered or meshed with the *red* route reflectors. The route reflectors themselves could be daemons running on the actual compute servers or on other dedicated or networking hardware.

The logical configuration would then have each compute server would have an address on each plane's subnet, and announce its end points on each subnet. If ECMP is then turned on, the compute servers would distribute the load across all planes.

If a plane were to fail (say due to a spanning tree failure), then only that one plane would fail. The remaining planes would stay running.

Note: This is the documentation for version 1.4.0 of Calico.

3.2 IP Interconnect Fabrics in Calico

3.2.1 Where large-scale IP networks and hardware collide

Calico provides an end-to-end IP network that interconnects the endpoints [1] in a scale-out or cloud environment. To do that, it needs an *interconnect fabric* to provide the physical networking layer on which Calico operates [2].

While Calico is designed to work with any underlying interconnect fabric that can support IP traffic, the fabric that has the least considerations attached to its implementation is an Ethernet fabric as discussed in our earlier technical note.

In most cases, the Ethernet fabric is the appropriate choice, but there are infrastructures where L3 (an IP fabric) has already been deployed, or will be deployed, and it makes sense for Calico to operate in those environments.

However, since Calico is, itself, a routed infrastructure, there are more engineering, architecture, and operations considerations that have to be weighed when running Calico with an IP routed interconnection fabric. We will briefly outline those in the rest of this post. That said, Calico operates equally well with Ethernet or IP interconnect fabrics.

Background

Basic Calico architecture overview

A description of the Calico architecture can be found in our architectural overview. However, a brief discussion of the routing and data paths is useful for the discussion.

In a Calico network, each compute server acts as a router for all of the endpoints that are hosted on that compute server. We call that function a vRouter. The data path is provided by the Linux kernel, the control plane by a BGP protocol server, and management plane by Calico's on-server agent, *Felix*.

Each endpoint can only communicate through its local vRouter, and the first and last *hop* in any Calico packet flow is an IP router hop through a vRouter. Each vRouter announces all of the endpoints it is attached to to all the other vRouters and other routers on the infrastructure fabric, using BGP, usually with BGP route reflectors to increase scale. A discussion of why we use BGP can be found in the Why BGP? blog post.

Access control lists (ACLs) enforce security (and other) policy as directed by whatever cloud orchestrator is in use. There are other components in the Calico architecture, but they are irrelevant to the interconnect network fabric discussion.

[1] In Calico's terminology, an endpoint is an IP address and interface. It could refer to a VM, a container, or even a process bound to an IP address running on a bare metal server.

[2] This interconnect fabric provides the connectivity between the Calico (v)Router (in almost all cases, the compute servers) nodes, as well as any other elements in the fabric (*e.g.* bare metal servers, border routers, and appliances).

Overview of current common IP scale-out fabric architectures

There are two approaches to building an IP fabric for a scale-out infrastructure. However, all of them, to date, have assumed that the edge router in the infrastructure is the top of rack (TOR) switch. In the Calico model, that function is pushed to the compute server itself.

Furthermore, in most current virtualized environments, the actual endpoint is not addressed by the fabric. If it is a VM, it is usually encapsulated in an overlay, and if it is a container, it may be encapsulated in an overlay, or NATed by some form of proxy, such as is done in the weave project network model, or the router in standard docker networking.

The two approaches are outlined below, in this technical note, we will cover the second option, as it is more common in the scale-out world. If there is interest in the first approach, please contact Project Calico, and we can discuss, and if there is enough interest, maybe we will do another technical note on that approach. If you know of other approaches in use, we would be happy to host a guest technical note.

1. The routing infrastructure is based on some form of IGP. Due to the limitations in scale of IGP networks (see the why BGP post for discussion of this topic), the project Calico team does not believe that using an IGP to distribute endpoint reachability information will adequately scale in a Calico environment. However, it is possible to use a combination of IGP and BGP in the interconnect fabric, where an IGP communicates the path to the *next-hop* router (in Calico, this is often the destination compute server) and BGP is used to distribute the actual next-hop for a given endpoint. This is a valid model, and, in fact is the most common approach in a widely distributed IP network (say a carrier's backbone network). The design of these networks is somewhat complex though, and will not be addressed further in this technical note. [3]
2. The other model, and the one that this note concerns itself with, is one where the routing infrastructure is based entirely on BGP. In this model, the IP network is "tight enough" or has a small enough diameter that BGP can be used to distribute endpoint routes, and the paths to the next-hops for those routes is known to all of the routers in the network (in a Calico network this includes the compute servers). This is the network model that this note will address.

BGP-only interconnect fabrics

There are multiple methods to build a BGP-only interconnect fabric. We will focus on three models, each with two widely viable variations. There are other options, and we will briefly touch on why we didn't include some of them in the *Other Options* appendix.

The two methods are:

1. A BGP fabric where each of the TOR switches (and their subsidiary compute servers) are a unique Autonomous System (AS) and they are interconnected via either an Ethernet switching plane provided by the spine switches in a leaf/spine architecture, or via a set of spine switches, each of which is also a unique AS. We'll refer to this as the *AS per rack* model. This model is detailed in this IETF working group draft.
2. A BGP fabric where each of the compute servers is a unique AS, and the TOR switches make up a transit AS. We'll refer to this as the *AS per server* model.

Each of these models can either have an Ethernet or IP spine. In the case of an Ethernet spine, each spine switch provides an isolated Ethernet connection *plane* as in the Calico Ethernet interconnect fabric model and each TOR switch is connected to each spine switch.

Another model is where each spine switch is a unique AS, and each TOR switch BGP peers with each spine switch. In both cases, the TOR switches use ECMP to load-balance traffic between all available spine switches.

[3] If there is interest in a discussion of this approach, please let us know. The Project Calico team could either arrange a discussion, or if there was enough interest, publish a follow-up tech note.

Some BGP network design considerations

Contrary to popular opinion, BGP is actually a fairly simple protocol. For example, the BGP configuration on a Calico compute server is approximately sixty lines long, not counting comments. The perceived complexity is due to the things that you can *do* with BGP. Many uses of BGP involve complex policy rules, where the behavior of BGP can be modified to meet technical (or business, financial, political, *etc.*) requirements. A default Calico network does not venture into those areas, [6] and therefore is fairly straight forward.

That said, there are a few design rules for BGP that need to be kept in mind when designing an IP fabric that will interconnect nodes in a Calico network. These BGP design requirements *can* be worked around, if necessary, but doing so takes the designer out of the standard BGP *envelope* and should only be done by an implementer who is *very* comfortable with advanced BGP design.

These considerations are:

AS continuity or *AS puddling* Any router in an AS *must* be able to communicate with any other router in that same AS without transiting another AS.

Next hop behavior By default BGP routers do not change the *next hop* of a route if it is peering with another router in its same AS. The inverse is also true, a BGP router will set itself as the *next hop* of a route if it is peering with a router in another AS.

Route reflection All BGP routers in a given AS must *peer* with all the other routers in that AS. This is referred to a *complete BGP mesh*. This can become problematic as the number of routers in the AS scales up. The use of *route reflectors* reduce the need for the complete BGP mesh. However, route reflectors also have scaling considerations.

Endpoints In a Calico network, each endpoint is a route. Hardware networking platforms are constrained by the number of routes they can learn. This is usually in range of 10,000's or 100,000's of routes. Route aggregation can help, but that is usually dependent on the capabilities of the scheduler used by the orchestration software (*e.g.* OpenStack).

A deeper discussion of these considerations can be found in the *IP Fabric Design Considerations* appendix.

The designs discussed below address these considerations.

The *AS Per Rack* model

This model is the closest to the model suggested by the IETF's Routing Area Working Group draft on BGP use in data centers.

As mentioned earlier, there are two versions of this model, one with an set of Ethernet planes interconnecting the ToR switches, and the other where the core planes are also routers. The following diagrams may be useful for the discussion.

In this approach, every ToR-ToR or ToR-Spine (in the case of an AS per spine) link is an eBGP peering which means that there is no route-reflection possible (using standard BGP route reflectors) *north* of the ToR switches.

If the L2 spine option is used, the result of this is that each ToR must either peer with every other ToR switch in the cluster (which could be hundreds of peers).

If the AS per spine option is used, then each ToR only has to peer with each spine (there are usually somewhere between two and sixteen spine switches in a pod). However, the spine switches must peer with all ToR switches (again, that would be hundreds, but most spine switches have more control plane capacity than the average ToR, so this might be more scalable in many circumstances).

Within the rack, the configuration is the same for both variants, and is somewhat different than the configuration north of the ToR.

[6] However those tools are available if a given Calico instance needs to utilize those policy constructs.

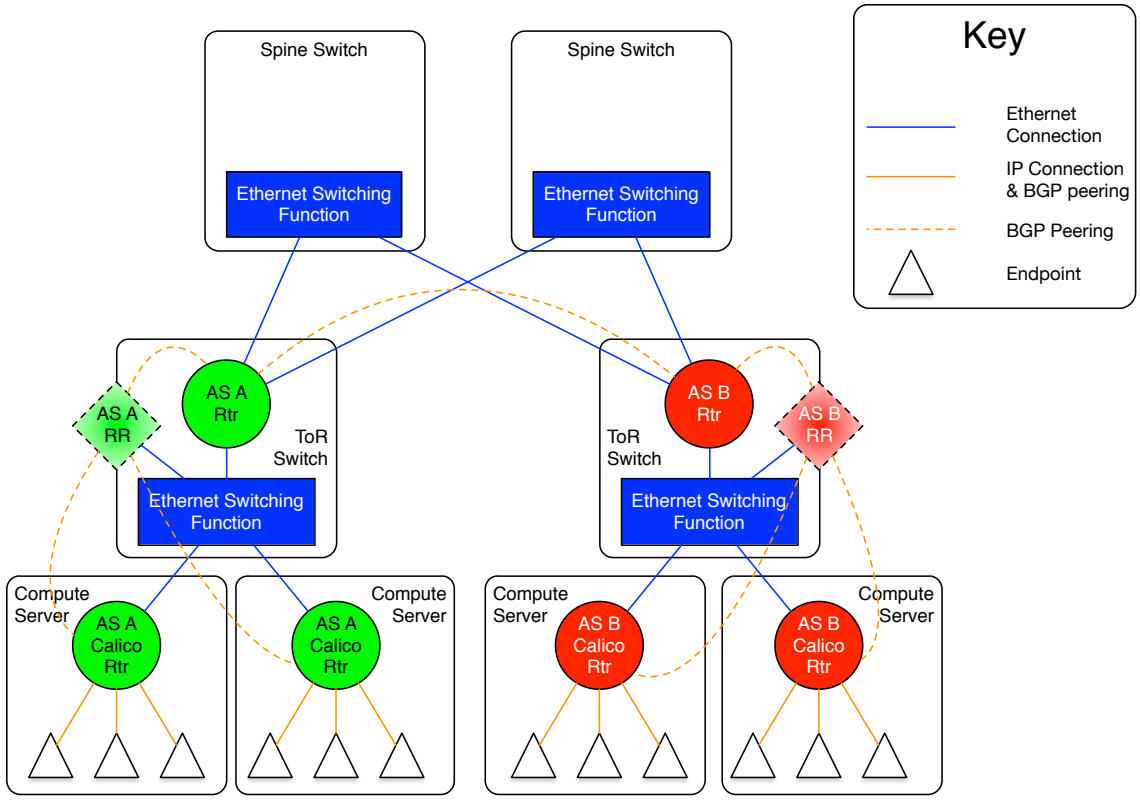


Fig. 3.3: This diagram shows the *AS per rack model* where the ToR switches are physically meshed via a set of Ethernet switching planes.

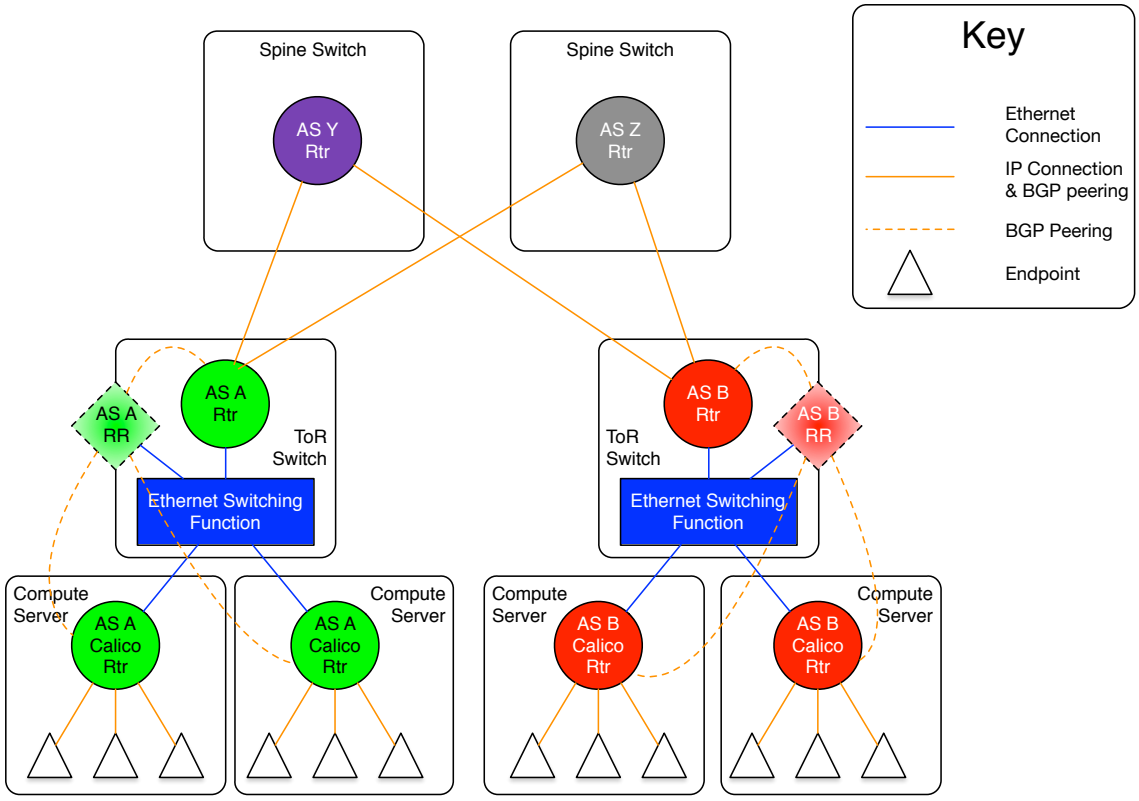


Fig. 3.4: This diagram shows the *AS per rack model* where the ToR switches are physically meshed via a set of discrete BGP spine routers, each in their own AS.

Every router within the rack, which, in the case of Calico is every compute server, shares the same AS as the ToR that they are connected to. That connection is in the form of an Ethernet switching layer. Each router in the rack must be directly connected to enable the AS to remain contiguous. The ToR's *router* function is then connected to that Ethernet switching layer as well. The actual configuration of this is dependent on the ToR in use, but usually it means that the ports that are connected to the compute servers are treated as *subnet* or *segment* ports, and then the ToR's *router* function has a single interface into that subnet.

This configuration allows each compute server to connect to each other compute server in the rack without going through the ToR router, but it will, of course, go through the ToR switching function. The compute servers and the ToR router could all be directly meshed, or a route reflector could be used within the rack, either hosted on the ToR itself, or as a virtual function hosted on one or more compute servers within the rack.

The ToR, as the eBGP router redistributes all of the routes from other ToRs as well as routes external to the data center to the compute servers that are in its AS, and announces all of the routes from within the AS (rack) to the other ToRs and the larger world. This means that each compute server will see the ToR as the next hop for all external routes, and the individual compute servers are the next hop for all routes external to the rack.

The *AS per Compute Server* model

This model takes the concept of an AS per rack to its logical conclusion. In the earlier referenced IETF draft the assumption in the overall model is that the ToR is first tier aggregating and routing element. In Calico, the ToR, if it is an L3 router, is actually the second tier. Remember, in Calico, the compute server is always the first/last router for an endpoint, and is also the first/last point of aggregation.

Therefore, if we follow the architecture of the draft, the compute server, not the ToR should be the AS boundary. The differences can be seen in the following two diagrams.

As can be seen in these diagrams, there are still the same two variants as in the *AS per rack* model, one where the spine switches provide a set of independent Ethernet planes to interconnect the ToR switches, and the other where that is done by a set of independent routers.

The real difference in this model, is that the compute servers as well as the ToR switches are all independent autonomous systems. To make this work at scale, the use of four byte AS numbers as discussed in **RFC 4893**. Without using four byte AS numbering, the total number of ToRs and compute servers in a calico fabric would be limited to the approximately five thousand available private AS [4] numbers. If four byte AS numbers are used, there are approximately ninety-two million private AS numbers available. This should be sufficient for any given Calico fabric.

The other difference in this model *vs.* the AS per rack model, is that there are no route reflectors used, as all BGP peerings are eBGP. In this case, each compute server in a given rack peers with its ToR switch which is also acting as an eBGP router. For two servers within the same rack to communicate, they will be routed through the ToR. Therefore, each server will have one peering to each ToR it is connected to, and each ToR will have a peering with each compute server that it is connected to (normally, all the compute servers in the rack).

The inter-ToR connectivity considerations are the same in scale and scope as in the AS per rack model.

The *Downward Default* model

The final model is a bit different. Whereas, in the previous models, all of the routers in the infrastructure carry full routing tables, and leave their AS paths intact, this model [5] removes the AS numbers at each stage of the routing path. This is to prevent routes from other nodes in the network from not being installed due to it coming from the *local* AS (since they share the source and dest of the route share the same AS).

[4] The two byte AS space reserves approximately the last five thousand AS numbers for private use. There is no technical reason why other AS numbers could not be used. However the re-use of global scope AS numbers within a private infrastructure is strongly discouraged. The chance for routing system failure or incorrect routing is substantial, and not restricted to the entity that is doing the reuse.

[5] We first saw this design in a customer's lab, and thought it innovative enough to share (we asked them first, of course). Similar *AS Path Stripping* approaches are used in ISP networks, however.

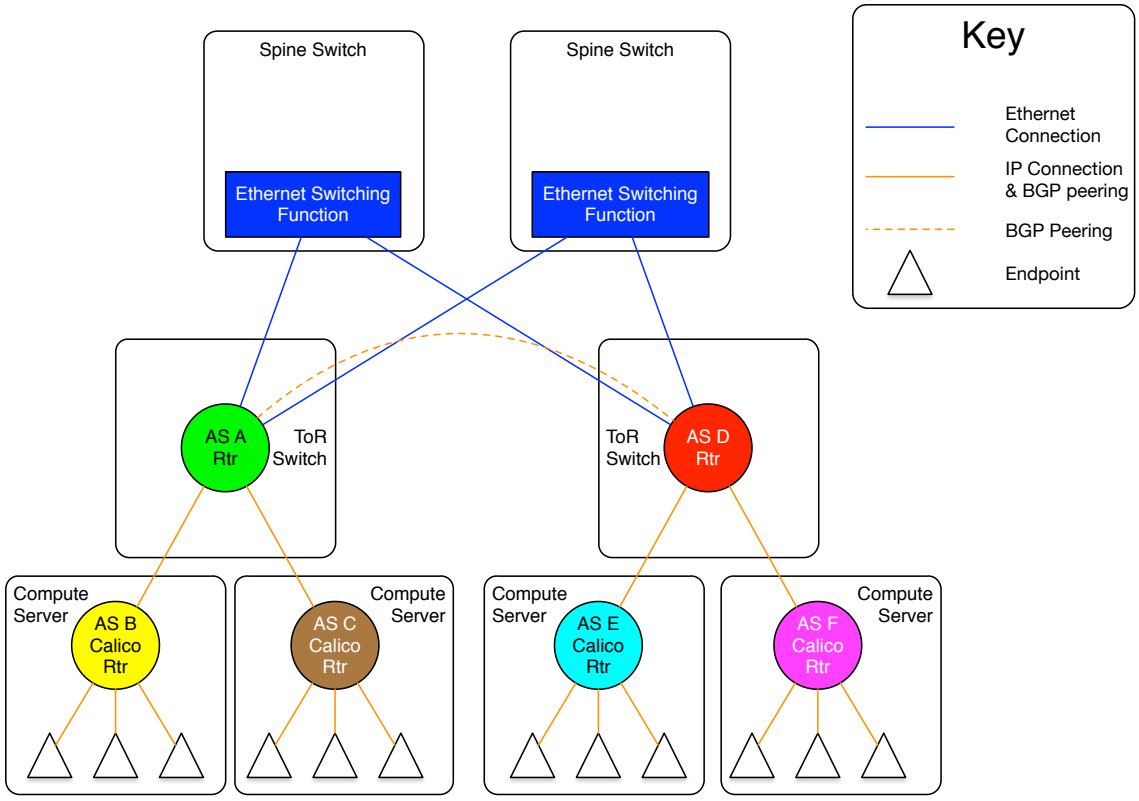


Fig. 3.5: This diagram shows the *AS per compute server model* where the ToR switches are physically meshed via a set of Ethernet switching planes.

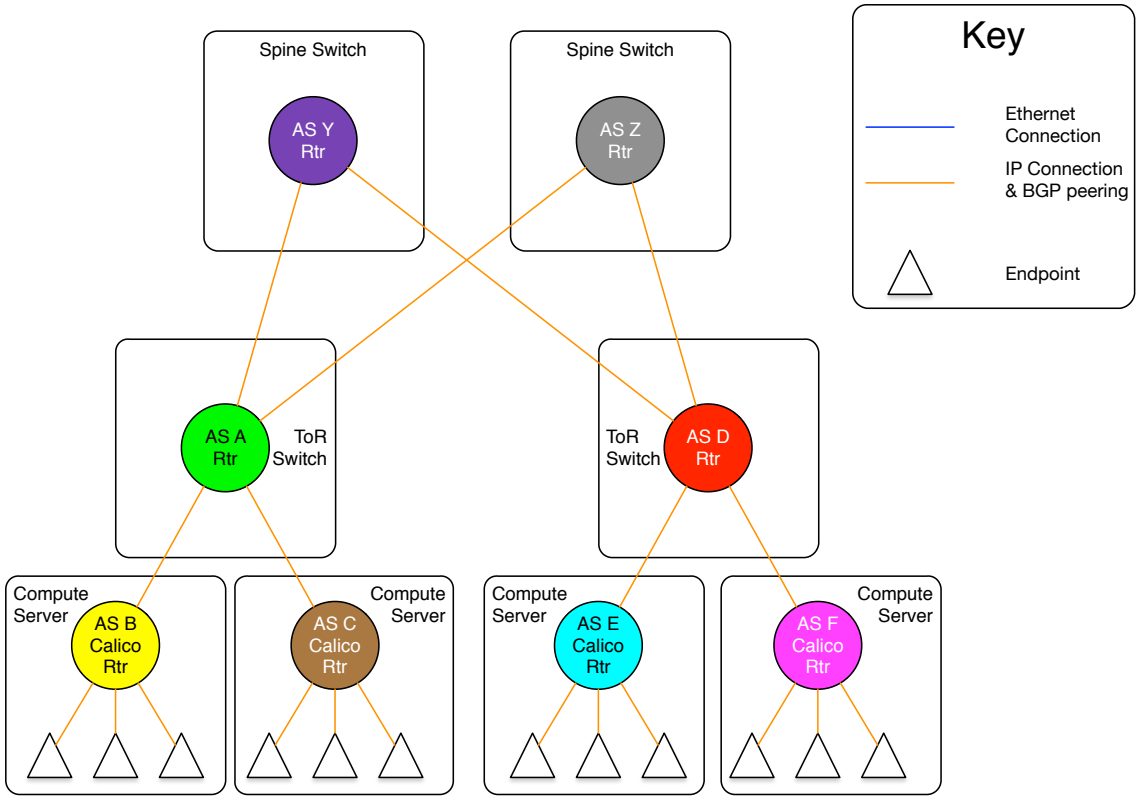


Fig. 3.6: This diagram shows the *AS per compute server model* where the ToR switches are physically connected to a set of independent routing planes.

The following diagram will show the AS relationships in this model.

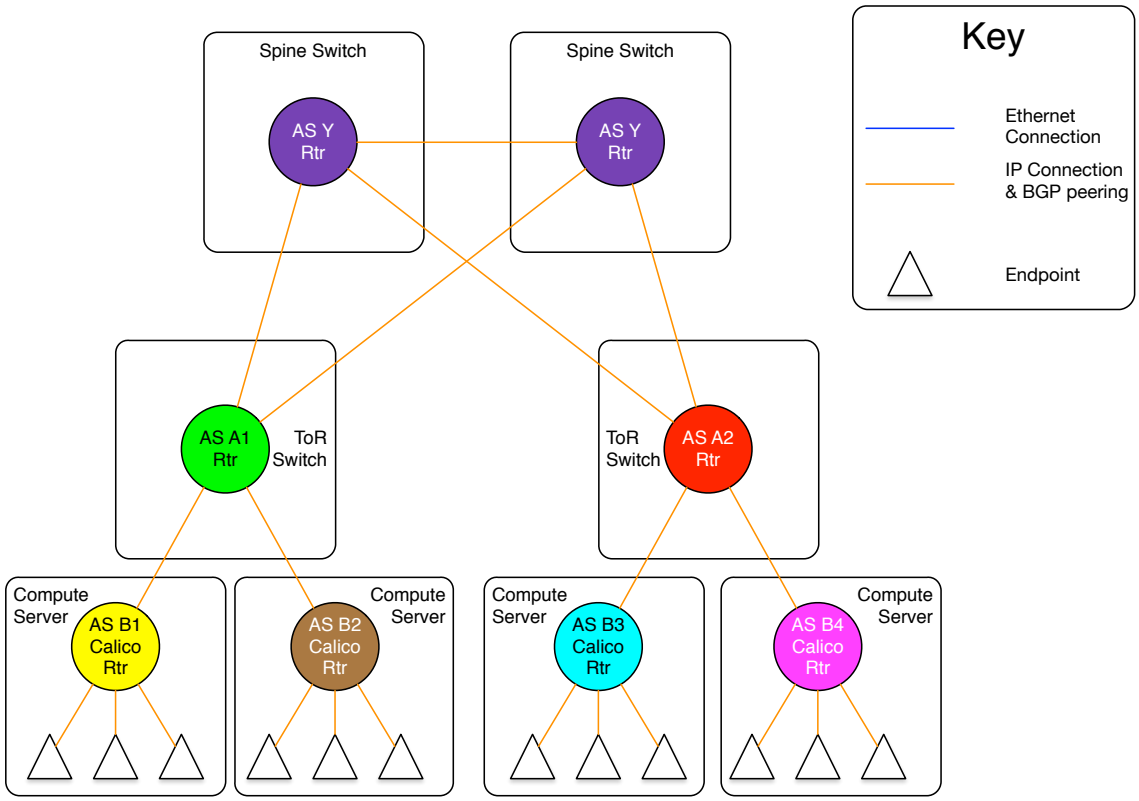


Fig. 3.7: In this diagram, we are showing that all Calico nodes share the same AS number, as do all ToR switches. However, those ASs are different (*A1* is not the same network as *A2*, even though the both share the same AS number *A*).

While the use of a single AS for all ToR switches, and another for all compute servers simplifies deployment (standardized configuration), the real benefit comes in the offloading of the routing tables in the ToR switches.

In this model, each router announces all of its routes to its upstream peer (the Calico routers to their ToR, the ToRs to the spine switches). However, in return, the upstream router only announces a default route. In this case, a given Calico router only has routes for the endpoints that are locally hosted on it, as well as the default from the ToR. Since the ToR is the only route for the Calico network the rest of the network, this matches reality. The same happens between the ToR switches and the spine. This means that the ToR only has to install the routes that are for endpoints that are hosted on its downstream Calico nodes. Even if we were to host 200 endpoints per Calico node, and stuff 80 Calico nodes in each rack, that would still limit the routing table on the ToR to a maximum of 16,000 entries (well within the capabilities of even the most modest of switches).

Since the default is originated by the Spine (originally) there is no chance for a downward announced route to originate from the recipient's AS, preventing the *AS puddling* problem.

There is one (minor) drawback to this model, in that all traffic that is destined for an invalid destination (the destination IP does not exist) will be forwarded to the spine switches before they are dropped.

It should also be noted that the spine switches do need to carry all of the Calico network routes, just as they do in the routed spines in the previous examples. In short, this model imposes no more load on the spines than they already would have, and substantially reduces the amount of routing table space used on the ToR switches. It also reduces the number of routes in the Calico nodes, but, as we have discussed before, that is not a concern in most deployments

as the amount of memory consumed by a full routing table in Calico is a fraction of the total memory available on a modern compute server.

Recommendation

The Project Calico team recommends the use of the *The AS Per Rack model* model if the resultant routing table size can be accommodated by the ToR and spine switches, remembering to account for projected growth.

If there is concern about the route table size in the ToR switches, the team recommends the *The Downward Default model*.

If there are concerns about both the spine and ToR switch route table capacity, or there is a desire to run a very simple L2 fabric to connect the Calico nodes, then the user should consider the Ethernet fabric as detailed in Calico over an Ethernet interconnect fabric

If a Calico user is interested in the AS per compute server, the Project Calico team would be very interested in discussing the deployment of that model.

Appendix

Other Options

The way the physical and logical connectivity is laid out in this note, and the Ethernet fabric note, The next hop router for a given route is always directly connected to the router receiving that route. This makes the need for another protocol to distribute the next hop routes unnecessary.

However, in many (or most) WAN BGP networks, the routers within a given AS may not be directly adjacent. Therefore, a router may receive a route with a next hop address that it is not directly adjacent to. In those cases, an IGP, such as OSPF or IS-IS, is used by the routers within a given AS to determine the path to the BGP next hop route.

There may be Calico architectures where there are similar models where the routers within a given AS are not directly adjacent. In those models, the use of an IGP in Calico may be warranted. The configuration of those protocols are, however, beyond the scope of this technical note.

IP Fabric Design Considerations

AS puddling The first consideration is that an AS must be kept contiguous. This means that any two nodes in a given AS must be able to communicate without traversing any other AS. If this rule is not observed, the effect is often referred to as *AS puddling* and the network will *not* function correctly.

A corollary of that rule is that any two administrative regions that share the same AS number, are in the same AS, even if that was not the desire of the designer. BGP has no way of identifying if an AS is local or foreign other than the AS number. Therefore re-use of an AS number for two *networks* that are not directly connected, but only connected through another *network* or AS number will not work without a lot of policy changes to the BGP routers.

Another corollary of that rule is that a BGP router will not propagate a route to a peer if the route has an AS in its path that is the same AS as the peer. This prevents loops from forming in the network. The effect of this prevents two routers in the same AS from transiting another router (either in that AS or not).

Next hop behavior Another consideration is based on the differences between iBGP and eBGP. BGP operates in two modes, if two routers are BGP peers, but share the same AS number, then they are considered to be in an *internal* BGP (or iBGP) peering relationship. If they are members of different AS's, then they are in an *external* or eBGP relationship.

BGP's original design model was that all BGP routers within a given AS would know how to get to one another (via static routes, IGP [7] routing protocols, or the like), and that routers in different ASs would not know how to reach one another unless they were directly connected.

Based on that design point, routers in an iBGP peering relationship assume that they do not transit traffic for other iBGP routers in a given AS (i.e. A can communicate with C, and therefore will not need to route through B), and therefore, do not change the *next hop* attribute in BGP [8].

A router with an eBGP peering, on the other hand, assumes that its eBGP peer will not know how to reach the next hop route, and then will substitute its own address in the next hop field. This is often referred to as *next hop self*.

In the Calico Ethernet fabric model, all of the compute servers (the routers in a Calico network) are directly connected over one or more Ethernet network(s) and therefore are directly reachable. In this case, a router in the Calico network does not need to set *next hop self* within the Calico fabric.

The models we present in this technical note insure that all routes that may traverse a non-Calico router are eBGP routes, and therefore *next hop self* is automatically set correctly. If a deployment of Calico in an IP interconnect fabric does not satisfy that constraint, then *next hop self* must be appropriately configured.

Route reflection As mentioned above, BGP expects that all of the iBGP routers in a network can see (and speak) directly to one another, this is referred to as a *BGP full mesh*. In small networks this is not a problem, but it does become interesting as the number of routers increases. For example, if you have 99 BGP routers in an AS and wish to add one more, you would have to configure the peering to that new router on each of the 99 existing routers. Not only is this a problem at configuration time, it means that each router is maintaining 100 protocol adjacencies, which can start being a drain on constrained resources in a router. While this might be *interesting* at 100 routers, it becomes an impossible task with 1000's or 10,000's of routers (the potential size of a Calico network).

Conveniently, large scale/Internet scale networks solved this problem almost 20 years ago by deploying BGP route reflection as described in **RFC 1966**. This is a technique supported by almost all BGP routers today. In a large network, a number of route reflectors [9] are evenly distributed and each iBGP router is *peered* with one or more route reflectors (usually 2 or 3). Each route reflector can handle 10's or 100's of route reflector clients (in Calico's case, the compute server), depending on the route reflector being used. Those route reflectors are, in turn, peered with each other. This means that there are an order of magnitude less route reflectors that need to be completely meshed, and each route reflector client is only configured to peer to 2 or 3 route reflectors. This is much easier to manage.

Other route reflector architectures are possible, but those are beyond the scope of this document.

Endpoints The final consideration is the number of endpoints in a Calico network. In the Ethernet fabric case the number of endpoints is not constrained by the interconnect fabric, as the interconnect fabric does not *see* the actual endpoints, it only *sees* the actual vRouters, or compute servers. This is not the case in an IP fabric, however. IP networks forward by using the destination IP address in the packet, which, in Calico's case, is the destination endpoint. That means that the IP fabric nodes (ToR switches and/or spine switches, for example) must know the routes to each endpoint in the network. They learn this by participating as route reflector clients in the BGP mesh, just as the Calico vRouter/compute server does.

However, unlike a compute server which has a relatively unconstrained amount of memory, a physical switch is either memory constrained, or quite expensive. This means that the physical switch has a limit on how many *routes* it can handle. The current industry standard for modern commodity switches is in the range of 128,000 routes. This means

[7] An Interior Gateway Protocol is a local routing protocol that does not cross an AS boundary. The primary IGPs in use today are OSPF and IS-IS. While complex iBGP networks still use IGP routing protocols, a data center is normally a fairly simple network, even if it has many routers in it. Therefore, in the data center case, the use of an IGP can often be disposed of.

[8] A Next hop is an attribute of a route announced by a routing protocol. In simple terms a route is defined by a *target*, or the destination that is to be reached, and a *next hop*, which is the next router in the path to reach that target. There are many other characteristics in a route, but those are well beyond the scope of this post.

[9] A route reflector may be a physical router, a software appliance, or simply a BGP daemon. It only processes routing messages, and does not pass actual data plane traffic. However, some route reflectors are co-resident on regular routers that do pass data plane traffic. While they may sit on one platform, the functions are distinct.

that, without other routing *tricks*, such as aggregation, a Calico installation that uses an IP fabric will be limited to the routing table size of its constituent network hardware, with a reasonable upper limit today of 128,000 endpoints.

Note: This is the documentation for version 1.4.0 of Calico.

3.3 Calico Architecture

This document discusses the various pieces of the Calico etcd-based architecture, with a focus on what specific role each component plays in the Calico network. This does not discuss the Calico etcd data model, which also acts as the primary API into the Calico network: for more on that, see Calico etcd Data Model.

3.3.1 Components

Calico is made up of the following interdependent components:

- *Felix*, the primary Calico agent that runs on each machine that hosts endpoints.
- *Orchestrator Plugin*, orchestrator-specific code that tightly integrates Calico into that orchestrator.
- *etcd*, the data store.
- *BGP Client (BIRD)*, a BGP client that distributes routing information.
- *BGP Route Reflector (BIRD)*, an optional BGP route reflector for higher scale.

The following sections break down each component in more detail.

3.3.2 Felix

Felix is a daemon that runs on every machine that provides endpoints: in most cases that means on nodes that host containers or VMs. It is responsible for programming routes and ACLs, and anything else required on the host, in order to provide the desired connectivity for the endpoints on that host.

Depending on the specific orchestrator environment, Felix is responsible for some or all of the following tasks:

Interface Management

Felix programs some information about interfaces into the kernel in order to get the kernel to correctly handle the traffic emitted by that endpoint. In particular, it will ensure that the host responds to ARP requests from each workload with the MAC of the host, and will enable IP forwarding for interfaces that it manages.

It also monitors for interfaces to appear and disappear so that it can ensure that the programming for those interfaces is applied at the appropriate time.

Route Programming

Felix is responsible for programming routes to the endpoints on its host into the Linux kernel FIB. This ensures that packets destined for those endpoints that arrive on at the host are forwarded accordingly.

ACL Programming

Felix is also responsible for programming ACLs into the Linux kernel. These ACLs are used to ensure that only valid traffic can be sent between endpoints, and ensure that endpoints are not capable of circumventing Calico's security measures. For more on this, see Security Policy Model.

State Reporting

Felix is responsible for providing data about the health of the network. In particular, it reports errors and problems with configuring its host. This data is written into etcd, to make it visible to other components and operators of the network.

3.3.3 Orchestrator Plugin

Unlike Felix there is no single 'orchestrator plugin': instead, there are separate plugins for each major cloud orchestration platform (e.g. OpenStack, Kubernetes). The purpose of these plugins is to bind Calico more tightly into the orchestrator, allowing users to manage the Calico network just as they'd manage network tools that were built in to the orchestrator.

A good example of an orchestrator plugin is the Calico Neutron ML2 mechanism driver. This component integrates with Neutron's ML2 plugin, and allows users to configure the Calico network by making Neutron API calls. This provides seamless integration with Neutron.

The orchestrator plugin is responsible for the following tasks:

API Translation

The orchestrator will inevitably have its own set of APIs for managing networks. The orchestrator plugin's primary job is to translate those APIs into the Calico etcd data model (see Calico etcd Data Model) to allow Calico to perform the appropriate network programming.

Some of this translation will be very simple, other bits may be more complex in order to render a single complex operation (e.g. live migration) into the series of simpler operations the rest of the Calico network expects.

Feedback

If necessary, the orchestrator plugin will provide feedback from the Calico network into the orchestrator. Examples include: providing information about Felix liveness; marking certain endpoints as failed if network setup failed.

3.3.4 etcd

etcd is a distributed key-value store that has a focus on consistency. Calico uses etcd to provide the communication between components and as a consistent data store, which ensures Calico can always build an accurate network.

Depending on the orchestrator plugin, etcd may either be the master data store or a lightweight mirror of a separate data store. For example, in an OpenStack deployment, the OpenStack database is considered the "source of truth" and etcd is used to mirror information about the network to the other Calico components.

The etcd component is distributed across the entire deployment. It is divided into two groups of machines: the core cluster, and the proxies.

For small deployments the core cluster can be an etcd cluster of one node (which would typically be co-located with the *Orchestrator Plugin* component). This deployment model is simple but provides no redundancy for etcd – in the

case of etcd failure the *Orchestrator Plugin* would have to rebuild the database which, as noted for OpenStack, will simply require that the plugin resynchronizes state to etcd from the OpenStack database.

In larger deployments the core cluster can be scaled up, as per the etcd admin guide.

Additionally, on each machine that hosts either a *Felix* or a *Orchestrator Plugin*, we run an etcd proxy. This reduces load on the core cluster and shields nodes from the specifics of the etcd cluster. In the case where the etcd cluster has a member on the same machine as a *Orchestrator Plugin*, we can forgo the proxy on that machine.

etcd is responsible for performing all of the following tasks:

Data Storage

etcd stores the data for the Calico network in a distributed, consistent, fault-tolerant manner (for cluster sizes of at least three etcd nodes). This set of properties ensures that the Calico network is always in a known-good state, while allowing for some number of the machines hosting etcd to fail or become unreachable.

This distributed storage of Calico data also improves the ability of the Calico components to read from the database (which is their most common operation), as they can distribute their reads around the cluster.

Communication

etcd is also used as a communication bus between components. We do this by having the non-etcd components watch certain points in the keyspace to ensure that they see any changes that have been made, allowing them to respond to those changes in a timely manner. This allows the act of committing state to the database to cause that state to be programmed into the network.

3.3.5 BGP Client (BIRD)

Calico deploys a BGP client on every node that also hosts a *Felix*. The role of the BGP client is to read routing state that *Felix* programs into the kernel and distribute it around the data center.

In Calico, this BGP component is most commonly BIRD, though any BGP client that can draw routes from the kernel and distribute them is suitable in this role.

The BGP client is responsible for performing the following tasks:

Route Distribution

When *Felix* inserts routes into the Linux kernel FIB, the BGP client will pick them up and distribute them to the other nodes in the deployment. This ensures that traffic is efficiently routed around the deployment.

3.3.6 BGP Route Reflector (BIRD)

For larger deployments, simple BGP can become a limiting factor because it requires every BGP client to be connected to every other BGP client in a mesh topology. This requires an increasing number of connections that rapidly become tricky to maintain, due to the N^2 nature of the increase.

For that reason, in larger deployments Calico will deploy a BGP route reflector. This component, commonly used in the Internet, acts as a central point to which the BGP clients connect, preventing them from needing to talk to every single BGP client in the cluster.

For redundancy, multiple BGP route reflectors can be deployed seamlessly. The route reflectors are purely involved in the control of the network: no endpoint data passes through them.

In Calico, this BGP component is also most commonly BIRD, configured as a route reflector rather than as a standard BGP client.

The BGP route reflector is responsible for the following tasks:

Centralized Route Distribution

When *BGP Client (BIRD)* advertises routes from its FIB to the route reflector, the route reflector advertises those routes out to the other nodes in the deployment.

Note: This is the documentation for version 1.4.0 of Calico.

3.4 How Calico Interprets Neutron API Calls

When running in an OpenStack deployment, Calico receives and interprets certain Neutron API actions, in order to program those actions down into the network. However, because Calico is substantially simpler than much of what Neutron generally allows (see Connectivity in OpenStack) and because it's a purely layer 3 model (see The Calico Data Path: IP Routing and iptables), not all Neutron API calls will have the same effect as they would with other backends.

This document will go into detail on the full range of Neutron API calls, and will discuss the effect they have on the network. It uses the Networking API v2.0 document from OpenStack as a basis for listing the various objects that the Neutron API uses: see that document for more information about what Neutron expects more generally.

Additionally, there is a section of this document that briefly covers Horizon actions: *Horizon*.

3.4.1 Networks

Networks are the basic networking concept in Neutron. A Neutron network is considered to be roughly equivalent to a physical network in terms of function: it defines a single layer 2 connectivity graph.

In vanilla Neutron, these can map to the underlay network in various ways, either by being encapsulated over it or by being directly mapped to it.

Generally speaking, Neutron networks can be created by all tenants. The administrator tenant will generally create some public Neutron networks that map to the underlay physical network directly for providing floating IPs: other tenants will create their own private Neutron networks as necessary.

In Calico, because all traffic is L3 and routed, the role of Neutron network as L2 connectivity domain is not helpful. Therefore, in Calico, Neutron networks are simply containers for subnets. Best practices for operators configuring Neutron networks in Calico deployments can be found in *Part 2: Set Up OpenStack*.

It is not useful for non-administrator tenants to create their own Neutron networks. Although Calico will allow non-administrator tenants to create Neutron networks, generally speaking administrators should use Neutron quotas to prevent non-administrator tenants from doing this.

Network creation events on the API are no-op events in Calico: a positive (2XX) response will be sent but no programming will actually occur.

Extended Attributes: Provider Networks

Neutron Provider networks are not used in Calico deployments. Setting provider network extended attributes will have no effect. See Connectivity in OpenStack to understand why Neutron provider networks are not needed.

3.4.2 Subnets

Neutron subnets are child objects of Neutron networks. In vanilla Neutron, a subnet is a collection of IP addresses and other network configuration (e.g. DNS servers) that is associated with a single Neutron network. A single Neutron network may have multiple Neutron subnets associated with it. Each Neutron subnet represents either an IPv4 or IPv6 block of addresses.

Best practices for configuring Neutron subnets in Calico deployments can be found in *Part 2: Set Up OpenStack*.

In Calico, these roles for the Neutron subnet are preserved in their entirety. All properties associated with these Neutron subnets are preserved and remain meaningful except for:

`host_routes` These have no effect, as the compute nodes will route traffic immediately after it egresses the VM.

3.4.3 Ports

In vanilla Neutron, a port represents a connection from a VM to a single layer 2 Neutron network. Obviously, the meaning of this object changes in a Calico deployment: instead, a port is a connection from a VM to the shared layer 3 network that Calico builds in Neutron.

All properties on a port work as normal, except for the following:

`network_id` The network ID still controls which Neutron network the port is attached to, and therefore still controls which Neutron subnets it will be placed in. However, as per the note in *Networks*, the Neutron network that a port is placed in does not affect which machines in the deployment it can contact.

Extended Attributes: Port Binding Attributes

The `binding:host-id` attribute works as normal. The following notes apply to the other attributes:

`binding:profile` This is unused in Calico.

`binding:vnic_type` This field, if used, **must** be set to `normal`. If set to any other value, Calico will not correctly function!

3.4.4 Quotas

Neutron quotas function unchanged.

In most deployments we recommend setting non-administrator tenant quotas for almost all Neutron objects to zero. For more information, see *Part 2: Set Up OpenStack*.

3.4.5 Security Groups

Security groups in vanilla OpenStack provide packet filtering processing to individual ports. They can be used to limit the traffic a port may issue.

In Calico, security groups have all the same function. Additionally, they serve to provide the connectivity-limiting function that in vanilla OpenStack is provided by Neutron networks. For more information, see Security Policy Model.

All the attributes of security groups remain unchanged in Calico.

3.4.6 Layer 3 Routing: Routers and Floating IPs

Layer 3 routing objects are divided into two categories: routers and floating IPs. Neither of these objects are supported by Calico: they simply aren't required. For more information, see Connectivity in OpenStack.

Any attempt to create these objects will fail, as Calico does not set up any Neutron L3 Agents.

3.4.7 LBaaS (Load Balancer as a Service)

Load Balancer as a Service does not function in a Calico network. Any attempt to create one will fail.

Note: It is possible that in a future version of Calico LBaaS may be functional. Watch this space.

3.4.8 Horizon

Horizon makes many provisioning actions available that mirror options on the Neutron API. This section lists them, and indicates whether they can be used or not, and any subtleties that might be present in them.

Much of the detail has been left out of this section, and is instead present in the relevant Neutron API sections above: please consult them for more.

Section: Project

Tab: Compute -> Instances

When launching instances, remember that security groups are used to determine reachability, not networks. Choose networks based on whether you need an external or an internal IP address, and choose security groups based on the machines you'd like to talk to in the cloud. See *Part 2: Set Up OpenStack* for more.

Tab: Compute -> Access & Security

As noted above, tenants should ensure they configure their security groups to set up their connectivity appropriately.

Tab: Network -> Network Topology

For the 'Create Network' button, see the *Networks* section. For the 'Create Router' button, see the *Layer 3 Routing: Routers and Floating IPs* section.

Tab: Network -> Networks

For networks and subnets, see the sections *Networks* and *Subnets*.

Tab: Network -> Routers

Tenants should be prevented from creating routers, as they serve no purpose in a Calico network. See *Layer 3 Routing: Routers and Floating IPs* for more.

Section: Admin

Tab: System Panel -> Networks

In the course of general operation administrators are not expected to make changes to their networking configuration. However, for initial network setup, this panel may be used to make changes. See Connectivity in OpenStack for details on how to achieve this setup.

Tab: System Panel -> Routers

Administrators should not create routers, as they serve no purpose in a Calico network. See *Layer 3 Routing: Routers and Floating IPs* for more.

Note: This is the documentation for version 1.4.0 of Calico.

3.5 Calico etcd Data Model

In Calico, etcd is used as the data store and communication mechanism for all the Calico components. This data store contains all the information the various Calico components require to set up the Calico network.

This document discusses the way Calico stores its data in etcd. This data store and structure acts as Calico's primary external and internal API, granting developers exceptional control over what Calico does. This document does not describe the components that read and write this data to provide the connectivity that endpoints in a Calico network want: for more on that, see Calico Architecture.

3.5.1 Objects

Calico focuses on the following major object types, stored in etcd:

endpoints An endpoint object represents a single source+sink of data in a Calico network; for example, the virtual NIC of a VM or a host's Linux interface. A single virtual machine, container or host may own multiple endpoints (e.g. if it has multiple vNICs). See *Endpoints* for more.

security profiles A security profile encapsulates a specific set of security rules to apply to an endpoint. Each endpoint can reference one or more security profiles. See *Security Profiles* for more.

security policies Similarly, a security policy contains a set of security rules to apply. Security policies allow a tiered security model, which can override the security profiles directly referenced by an endpoint.

Each security policy has a selector predicate, such as "type == 'webserver' && role == 'frontend"', that picks out the endpoints it should apply to, and an ordering number that specifies the policy's priority. For each endpoint, Calico applies the security policies that apply to it, in priority order, and then that endpoint's security profiles.

See *Tiered security policy* for more.

The structure of all of this information can be found below.

Endpoints

The Calico datamodel supports two types of endpoint:

- Workload endpoints refer to interfaces attached to workloads such as VMs or containers, which are running on the host that is running Calico's agent, Felix. Calico identifies such interfaces by a name prefix; for example OpenStack VM interfaces always start with "tap...". By default, Calico blocks all traffic to and from workload interfaces.

  Each workload endpoint is stored in an etcd key that matches the following pattern:

```
/calico/v1/host/<hostname>/workload/<orchestrator_id>/<workload_id>/endpoint/<endpoint_id>
```

- Host endpoints refer to the "bare-metal" interfaces attached to the host that is running Calico's agent, Felix. By default, Calico doesn't apply any policy to such interfaces.

  Each host endpoint is stored in an etcd key that matches the following pattern:

```
/calico/v1/host/<hostname>/endpoint/<endpoint_id>
```

The parameters in the paths have the following meanings:

hostname the hostname of the compute server

orchestrator_id for workload endpoints only, the name of the orchestrator that owns the endpoint, e.g. `"docker"` or `"openstack"`.

workload_id for workload endpoints only, an identifier provided by the orchestrator to relate multiple endpoints that belong to the same workload (e.g. a single VM).

endpoint_id an (opaque) identifier for a specific endpoint

Workload endpoints

For workload endpoints, the object stored is a JSON blob with the following structure:

```
{
  "state": "active|inactive",
  "name": "<name of linux interface>",
  "mac": "<MAC of the interface>",
  "profile_ids": ["<profile_id>", ...],
  "ipv4_nat": [
    {"int_ip": "198.51.100.17", "ext_ip": "192.168.0.1"},
    ...
  ],
  "ipv4_nets": [
    "198.51.100.17/32",
    ...
  ],
  "ipv6_nat": [
    {"int_ip": "2001:db8::19", "ext_ip": "2001::2"},
    ...
  ],
  "ipv6_nets": [
    "2001:db8::19/128",
    ...
  ],
  "ipv4_gateway": "<IP address>",
  "ipv6_gateway": "<IP address>",
  "labels": {
    "<key>": "<value>",
    "<key>": "<value>",
    ...
```

```
  }
}
```

The various properties in this object have the following meanings:

state one of "active" or "inactive". If "active", the endpoint should be able to send and receive traffic: if inactive, it should not.

name the name of the Linux interface on the host: for example, `tap80`.

mac the MAC address of the endpoint interface.

profile_ids a list of identifiers of *Security Profiles* objects that apply to this endpoint. Each profile is applied to packets in the order that they appear in this list.

ipv4_nat a list of 1:1 NAT mappings to apply to the endpoint. Inbound connections to ext_ip will be forwarded to int_ip. Connections initiated from int_ip will not have their source address changed, except when an endpoint attempts to connect one of its own ext_ips. Each int_ip must be associated with the same endpoint via ipv4_nets.

ipv4_nets a list of IPv4 subnets allocated to this endpoint. IPv4 packets will only be allowed to leave this interface if they come from an address in one of these subnets.

> **Note:** Currently only /32 subnets are supported.

ipv6_nat a list of 1:1 NAT mappings to apply to the endpoint. Inbound connections to ext_ip will be forwarded to int_ip. Connections initiated from int_ip will not have their source address changed, except when an endpoint attempts to connect one of its own ext_ips. Each int_ip must be associated with the same endpoint via ipv6_nets.

ipv6_nets a list of IPv6 subnets allocated to this endpoint. IPv6 packets will only be allowed to leave this interface if they come from an address in one of these subnets.

> **Note:** Currently only /128 subnets are supported.

ipv4_gateway the gateway IPv4 address for traffic from the VM.

ipv6_gateway the gateway IPv6 address for traffic from the VM.

labels An optional dict of string key-value pairs. Labels are used to attach useful identifying information to endpoints. It is expected that many endpoints share the same labels. For example, they could be used to label all "production" workloads with "deployment=prod" so that security policy can be applied to production workloads.

If `labels` is missing, it is treated as if there was an empty dict.

Host endpoints

For host enpdoints, the object stored is a JSON blob of the following form; the fields are described below:

```
{
  "name": "<name of linux interface>",

  "expected_ipv4_addrs": ["10.0.0.0", ...],
  "expected_ipv6_addrs": ["2201:db8::19", ...],

  "profile_ids": ["<profile_id>", ...],

  "labels": {
```

```
    "<key>": "<value>",
    "<key>": "<value>",
    ...
  }
}
```

The various properties in this object have the following meanings:

name Required if none of the `expected_ipvX_addrs` fields are present: the name of the interface to apply policy to; for example "eth0". If "name" is not present then at least one expected IP must be specified.

expected_ipv4_addrs and expected_ipv6_addrs At least one required if `name` is not present: the expected local IP address of the endpoint. If `name` is not present, Calico will look for an interface matching *any* of the IPs in the list and apply policy to that.

profile_ids a list of identifiers of *Security Profiles* objects that apply to this endpoint. Each profile is applied to packets in the order that they appear in this list.

labels An optional dict of string key-value pairs. Labels are used to attach useful identifying information to endpoints. It is expected that many endpoints share the same labels. For example, they could be used to label all "production" workloads with "deployment=prod" so that security policy can be applied to production workloads.

If `labels` is missing, it is treated as if there was an empty dict.

Note: When using the `src_selector|tag` or `dst_selector|tag` match criteria in a firewall rule, Calico converts the selector into a set of IP addresses. For host endpoints, the `expected_ipvX_addrs` fields are used for that purpose. (If only the interface name is specified, Calico does not learn the IP of the interface for use in match criteria.)

Security Profiles

Each security profile is split up into three bits of data: 'rules', 'tags' and 'labels'.

The 'rules' are an ordered list of ACLs, specifying what should be done with specific kinds of IP traffic. Traffic that matches a set of rule criteria will be accepted or dropped, depending on the rule.

The 'tags' are a list of classifiers that apply to each endpoint that refences the profile. The purpose of the tags is to allow for rules in other profiles/policies to refer to profiles by name, rather than by membership.

Finally, labels contains a JSON dict with a set of key/value labels (as described above). The labels on a profile are inherited by all the endpoints that directly reference that profile and they can be used in selectors as if they were directly applied to the endpoint. 'labels' is optional.

For each profile, the rules, tags and labels objects are stored in different keys, of the form:

```
/calico/v1/policy/profile/<profile_id>/rules
/calico/v1/policy/profile/<profile_id>/tags
/calico/v1/policy/profile/<profile_id>/labels
```

Tiered security policy

In addition to directly-referenced security profiles, Calico supports an even richer security model that we call "tiered policy". Tiered policy consists of a series of explicitly ordered "tiers". Tiers contain (explicitly ordered) "policies". Each policy has a Boolean selector expression that decides whether it applies to a given endpoint. Selector expressions match against an endpoint's labels.

Each tier might have a different owner; for example, an enterprise's NetSec team could install a global black/white list that comes before rules generated by a Calico plugin:

```
tier 1: global "netsec" rules
    policy 1, all endpoints: <global blacklist>
    policy 2, all endpoints: <global whitelist>
    ...
tier 2: Calico plugin-defined rules
    policy 1, role == "webserver" && deployment == "prod": <prod webserver rules>
tier 3: ...
```

Each policy must do one of the following:

- Match the packet and apply a "next-tier" action; this skips the rest of the tier, deferring to the next tier (or the explicit profiles if this is the last tier.
- Match the packet and apply an "allow" action; this immediately accepts the packet, skipping all further tiers and profiles. This is not recommended in general, because it prevents further policy from being executed.
- Match the packet and apply a "deny" action; this drops the packet immediately, skipping all further tiers and profiles.
- Fail to match the packet; in which case the packet proceeds to the next policy in the tier. If there are no more policies in the tier then the packet is dropped.

Note: If no policies in a tier match an endpoint then the packet skips the tier completely. The "default deny" behavior described above only applies once one of the profiles in a tier has matched a packet.

Calico implements the security policy for each endpoint individually and only the policies that have matching selectors are implemented. This ensures that the number of rules that actually need to be inserted into the kernel is proportional to the number of local endpoints rather than the total amount of policy. If no policies in a tier match a given endpoint then that tier is skipped.

Tiered security policies are stored in etcd in the keys of the form:

```
/calico/v1/policy/tier/<tier_name>/policy/<policy_id>
```

Each `<tier-name>` directory defines a tier and each tier is required to have a metadata key inside it:

```
/calico/v1/policy/tier/<tier_name>/metadata
```

The metadata key contains a JSON dict, which currently contains only the order for the tier:

```
{"order": <number>|"default"}
```

Tiers with higher "order" values are applied after those with lower numbers. If the `order` is omitted or set to "default" then the tier effectively has infinite order, it will be applied after any other tiers.

The security policy itself is very similar to the `rules` JSON dict that is used for policy, with the addition of a selector and order of its own:

```
{
    "selector": "<selector-expression>",
    "order": <number>|"default",
    "inbound_rules": [{<rule>}, ...],
    "outbound_rules": [{<rule>}, ...]
}
```

Note: Security policies do not have an associated `labels` or `tags` object.

Similarly to the tier order, policies with lower values for "order" are applied first.

Selector expressions follow this syntax:

```
label == "string_literal"  ->  comparison, e.g. my_label == "foo bar"
label != "string_literal"   ->  not equal; also matches if label is not present
label in { "a", "b", "c", ... }  ->  true if the value of label X is one of "a", "b", "c"
label not in { "a", "b", "c", ... }  ->  true if the value of label X is not one of "a", "b", "c"
has(label_name)  -> True if that label is present
! expr -> negation of expr
expr && expr  -> Short-circuit and
expr || expr  -> Short-circuit or
( expr ) -> parens for grouping
all() or the empty selector -> matches all endpoints.
```

Label names are allowed to contain alphanumerics, `-`, `_` and `/`. String literals are more permissive but they do not support escape characters.

Examples (with made-up labels):

```
type == "webserver" && deployment == "prod"
type in {"frontend", "backend"}
deployment != "dev"
! has(label_name)
```

Rules

The 'rules' key contains the following JSON-encoded data:

```
{
  "inbound_rules": [{<rule>}, ...],
  "outbound_rules": [{<rule>}, ...]
}
```

Two lists of rules objects, one applying to traffic destined for that endpoint (`inbound_rules`), one applying to traffic emitted by that endpoint (`outbound_rules`).

Each rule sub-object has the following JSON-encoded structure:

```
{
  # Positive matches:
  "protocol": "tcp|udp|icmp|icmpv6|<number>",
  "src_tag": "<tag_name>",
  "src_selector": "<selector expression>",
  "src_net": "<CIDR>",
  "src_ports": [1234, "2048:4000"],
  "dst_tag": "<tag_name>",
  "dst_net": "<CIDR>",
  "dst_ports": [1234, "2048:4000"],
  "icmp_type": <int>, "icmp_code": <int>,  # Treated together, see below.

  # Negated matches:
  "!protocol": ...,
  "!src_tag": ...,
  "!src_selector": ...,
```

```
  "!src_net": ...,
  "!src_ports": ...,
  "!dst_tag": ...,
  "!dst_net": ...,
  "!dst_ports": ...,
  "!icmp_type": ..., "!icmp_code": ...,  # Treated together, see below.

  # If present, "log_prefix" causes the matched packet to be logged
  # with the given prefix.
  "log_prefix": "<log-prefix>",

  "action": "deny | allow | next-tier",
}
```

Each positive match criteria has a negated version, prefixed with "!". All the match criteria within a rule must be satisfied for a packet to match. A single rule can contain the positive and negative version of a match and both must be satisfied for the rule to match.

All of these properties are optional but some have dependencies (such as requiring the protocol to be specified):

`protocol` if present, restricts the rule to only apply to traffic of a specific IP protocol. Required if `*_ports` is used (becuase ports only apply to certain protocols).

Must be one of these string values: `"tcp"`, `"udp"`, `"icmp"`, `"icmpv6"`, `"sctp"`, `"udplite"` or an integer in the range 1-255.

`src_tag` if present, restricts the rule to only apply to traffic that originates from endpoints that have profiles with the given tag in them.

`src_net` if present, restricts the rule to only apply to traffic that originates from IP addresses in the given subnet.

`src_selector` if present, contains a selector expression as described in *Tiered security policy*. Only traffic that originates from endpoints matching the selector will be matched.

> **Warning:** In addition to the negative version of "src_selector" (which is "!src_selector") the selector expression syntax itself supports negation. The two types of negation are subtly different. One negates the set of matched endpoints, the other negates the whole match:
> `"src_selector": !has(my_label)` matches packets that are from other Calico-controlled endpoints that **do not** have the label "my_label".
> `"!src_selector": has(my_label)` matches packets that are not from Calico-controlled endpoints that **do** have the label "my_label".
> The effect is that the latter will accept packets from non-Calico sources whereas the former is limited to packets from Calico-controlled endpoints.

`src_ports` if present, restricts the rule to only apply to traffic that has a source port that matches one of these ranges/values. This value is a list of integers or strings that represent ranges of ports.

Since only some protocols have ports, requires the (positive) `protocol` match to be set to `"tcp"` or `"udp"` (even for a negative match).

`dst_tag` if present, restricts the rule to only apply to traffic that is destined for endpoints that have profiles with the given tag in them.

`dst_selector` if present, contains a selector expression as described in *Tiered security policy*. Only traffic that is destined for endpoints matching the selector will be matched.

> **Warning:** The subtlety described above around negating `"src_selector"` also applies to `"dst_selector"`.

dst_net if present, restricts the rule to only apply to traffic that is destined for IP addresses in the given subnet.

dst_ports if present, restricts the rule to only apply to traffic that is destined for a port that matches one of these ranges/values. This value is a list of integers or strings that represent ranges of ports.

Since only some protocols have ports, requires the (positive) `protocol` match to be set to `"tcp"` or `"udp"` (even for a negative match).

icmp_type and icmp_code if present, restricts the rule to apply to a specific type and code of ICMP traffic (e.g. `"icmp_type8": 8` would correspond to ICMP Echo Request, better known as ping traffic). May only be present if the (positive) `protocol` match is set to `"icmp"` or `"icmpv6"`.

If `icmp_code` is specified then `icmp_type` is required. This is a technical limitation imposed by the kernel's iptables firewall, which Calico uses to enforce the rule.

> **Warning:** Due to the same kernel limiation, the negated versions of the ICMP matches are treated together as a single match. A rule that uses `!icmp_type` and `!icmp_code` together will match all ICMP traffic apart from traffic that matches **both** type and code.

log_prefix if present, in addition to doing the configured action, Calico will log the packet with this prefix. The current implementation uses iptables LOG action, which results in a log to syslog.

For iptables compatibility, Calico will truncate the prefix to 27 characters and limit the character set.

action what action to take when traffic matches this rule. One of

- `deny`, which drops the packet immediately
- `allow`, which accepts the packet unconditionally
- `next-tier`, which, in tiered security policies, jumps to the next tier and continues processing. (In profiles, the `next-tier` action is a synonym for `allow`.)
- `log`, which logs the packet (to syslog) and then continues processing rules.

> **Note:** Since Calico implements a stateful firewall, normally only the first packet in a TCP or ICMP flow will be logged.

Tags

The value of the tag key is a JSON list of tag strings, as shown below:

```
["A", "B", "C", ...]
```

Each tag in this list applies to every endpoint that is associated with this policy. These tags can be referred to by rules, as shown above.

A single tag may be associated with multiple security profiles, in which case it expands to reference all endpoints in all of those profiles.

Calico is an open source project and we would value your help, involvement and input.

Note: This is the documentation for version 1.4.0 of Calico.

CHAPTER 4

Getting Involved

Calico is an open source project, and we'd love you to get involved. Whether that might be by reading and participating on our mailing list, or by diving into the code to propose enhancements or integrate with other systems. To see the options for getting involved with Calico the project, please take a look at the following.

4.1 Mailing Lists

Project Calico runs two mailing lists:

- calico-announce provides a read-only list providing a regular update on Project Calico. Please subscribe so that you can keep up to date on the project.
- calico-tech provides a list for technical discussions and queries about the project. You're welcome to subscribe and to post any Calico-related discussion to this list, including problems, ideas, or requirements.

When reporting a problem on calico-tech, please try to:

- provide a clear subject line
- provide as much diagnostic information as possible.

That will help us (or anyone else) to answer your question quickly and correctly.

4.2 Read the Source, Luke!

All Calico's code is on GitHub, in the following repositories, separated by function.

- calico - All of the core Calico code except for that specific to Docker/container environments: the Felix agent, the OpenStack plugin; testing for all of those; and the source for Calico's documentation.
- calico-containers - Calico code and components specific to Docker/container environments: the lightweight orchestrator for Docker environments, Powerstrip adapter, and so on; and instructions for demonstrating Calico networking in various container environments.
- calico-neutron - Calico-specific patched version of OpenStack Neutron.
- calico-nova - Calico-specific patched version of OpenStack Nova.
- calico-dnsmasq - Calico-specific patched version of Dnsmasq.
- calico-chef - Chef cookbooks for installing test versions of OpenStack-using-Calico.

4.3 Contributing

Calico follows the "Fork & Pull" model of collaborative development, with changes being offered to the main Calico codebase via Pull Requests. So you can contribute a fix, change or enhancement by forking one of our repositories and making a GitHub pull request. If you're interested in doing that:

- Thanks!
- See the GitHub docs for how to create a Pull Request.
- Check our Contribution Guidelines for more information.

Note: This is the documentation for version 1.4.0 of Calico.

CHAPTER 5

Contribution Guidelines

Features or any changes to the codebase should be done as follows:

1. Pull latest code in the **master** branch and create a feature branch off this.
2. Implement your feature. Commits are cheap in Git, try to split up your code into many. It makes reviewing easier as well as for saner merging.
    - If your commit fixes an existing issue #123, include the text "fixes #123" in at least one of your commit messages. This ensures the pull request is attached to the existing issue (see How do you attach a new pull request to an existing issue on GitHub?).
3. Push your feature branch to GitHub. Note that before we can accept your changes, you need to agree to one of our contributor agreements. See *Contributor Agreements*.
4. Create a pull request using GitHub, from your branch to master.
5. Reviewer process:
    - Receive notice of review by GitHub email, GitHub notification, or by checking your assigned Project Calico GitHub issues.
    - Make markups as comments on the pull request (either line comments or top-level comments).
    - Make a top-level comment saying something along the lines of "Fine; some minor comments" or "Some issues to address before merging".
    - If there are no issues, merge the pull request and close the branch. Otherwise, assign the pull request to the developer and leave this to them.
6. Developer process:
    - Await review.
    - Address code review issues on your feature branch.
    - Push your changes to the feature branch on GitHub. This automatically updates the pull request.
    - If necessary, make a top-level comment along the lines of "Please re-review", assign back to the reviewer, and repeat the above.
    - If no further review is necessary and you have the necessary privileges, merge the pull request and close the branch. Otherwise, make a top-level comment and assign back to the reviewer as above.

5.1 Contributor Agreements

If you plan to contribute in the form of documentation or code, we need you to sign our Contributor License Agreement before we can accept your contribution. You will be prompted to do this as part of the PR process on Github.

Note: This is the documentation for version 1.4.0 of Calico.

CHAPTER 6

Frequently Asked Questions

This page contains answers to several frequently asked technical questions about Calico. It is updated on a regular basis: please check back for more information.

- *"Why use Calico?"*
- *"Does Calico work with IPv6?"*
- *"Is Calico compliant with PCI/DSS requirements?"*
- *"How does Calico maintain saved state?"*
- *"I heard Calico is suggesting layer 2: I thought you were layer 3! What's happening?"*
- *"I need to use hard-coded private IP addresses: how do I do that?"*
- *"How do I control policy/connectivity without virtual/physical firewalls?"*
- *"How does Calico interact with the Neutron API?"*

6.1 "Why use Calico?"

The problem Calico tries to solve is the networking of workloads (VMs, containers, etc) in a high scale environment. Existing L2 based methods for solving this problem have problems at high scale. Compared to these, we think Calico is more scalable, simpler and more flexible. We think you should look into it if you have more than a handful of nodes on a single site.

For a more detailed discussion of this topic, see our blog post at Why Calico?.

6.2 "Does Calico work with IPv6?"

Yes! We have demonstrated IPv6 with Calico on OpenStack and Docker/Powerstrip.

6.3 "Is Calico compliant with PCI/DSS requirements?"

PCI certification applies to the whole end-to-end system, of which Calico would be a part. We understand that most current solutions use VLANs, but after studying the PCI requirements documents, we believe that Calico does meet those requirements and that nothing in the documents *mandates* the use of VLANs.

6.4 "How does Calico maintain saved state?"

State is saved in a few places in a Calico deployment, depending on whether it's global or local state.

Local state is state that belongs on a single compute host, associated with a single running Felix instance (things like kernel routes, tap devices etc.). Local state is entirely stored by the Linux kernel on the host, with Felix storing it only as a temporary mirror. This makes Felix effectively stateless, with the kernel acting as a backing data store on one side and *etcd* as a data source on the other.

If Felix is restarted, it learns current local state by interrogating the kernel at start up. It then reads from `etcd` all the local state which it should have, and updates the kernel to match. This approach has strong resiliency benefits, in that if Felix restarts you don't suddenly lose access to your VMs or containers. As long as the Linux kernel is running, you've still got full functionality.

The bulk of global state is mastered in whatever component hosts the plugin.

- In the case of OpenStack, this means a Neutron database. Our OpenStack plugin (more strictly a Neutron ML2 driver) queries the Neutron database to find out state about the entire deployment. That state is then reflected to `etcd` and so to Felix.
- In certain cases, `etcd` itself contains the master copy of the data. This is because some Docker deployments have an `etcd` cluster that has the required resiliency characteristics, used to store all system configuration - and so `etcd` is configured so as to be a suitable store for critical data.
- In other orchestration systems, it may be stored in distributed databases, either owned directly by the plugin or by the orchestrator itself.

The only other state storage in a Calico network is in the BGP sessions, which approximate a distributed database of routes. This BGP state is simply a replicated copy of the per-host routes configured by Felix based on the global state provided by the orchestrator.

This makes the Calico design very simple, because we store very little state. All of our components can be shutdown and restarted without risk, because they resynchronize state as necessary. This makes modelling their behaviour extremely simple, reducing the complexity of bugs.

6.5 "I heard Calico is suggesting layer 2: I thought you were layer 3! What's happening?"

It's important to distinguish what Calico provides to the workloads hosted in a data center (a purely layer 3 network) with what the Calico project *recommends* operators use to build their underlying network fabric.

Calico's core principle is that *applications* and *workloads* overwhelmingly need only IP connectivity to communicate. For this reason we build an IP-forwarded network to connect the tenant applications and workloads to each other, and the broader world.

However, the underlying physical fabric obviously needs to be set up too. Here, Calico has discussed how both a layer 2 (see Calico over an Ethernet interconnect fabric) or a layer 3 (see IP Interconnect Fabrics in Calico) fabric could be integrated with Calico. This is one of the great strengths of the Calico model: it allows the infrastructure to be decoupled from what we show to the tenant applications and workloads.

We have some thoughts on different interconnect approaches (as noted above), but just because we say that there are layer 2 and layer 3 ways of building the fabric, and that those decisions may have an impact on route scale, does not mean that Calico is "going back to Ethernet" or that we're recommending layer 2 for tenant applications. In all cases we forward on IP packets, no matter what architecture is used to build the fabric.

6.6 "I need to use hard-coded private IP addresses: how do I do that?"

While this isn't supported today, this is on our roadmap using a stateless variant of RFC 6877 (464-XLAT). For more detail, see Overlapping IPv4 address ranges.

6.7 "How do I control policy/connectivity without virtual/physical firewalls?"

Calico provides an extremely rich security policy model, detailed in Security Policy Model. This model applies the policy at the first and last hop of the routed traffic within the Calico network (the source and destination compute hosts).

This model is substantially more robust to failure than a centralised firewall-based model. In particular, the Calico approach has no single-point-of-failure: if the device enforcing the firewall has failed then so has one of the workloads involved in the traffic (because the firewall is enforced by the compute host).

This model is also extremely amenable to scaling out. Because we have a central repository of policy configuration, but apply it at the edges of the network (the hosts) where it is needed, we automatically ensure that the rules match the topology of the data center. This allows easy scaling out, and gives us all the advantages of a single firewall (one place to manage the rules), but none of the disadvantages (single points of failure, state sharing, hairpinning of traffic, etc.).

Lastly, we decouple the reachability of nodes and the policy applied to them. We use BGP to distribute the topology of the network, telling every node how to get to every endpoint in case two endpoints need to communicate. We use policy to decide *if* those two nodes should communicate, and if so, how. If policy changes and two endpoints should now communicate, where before they shouldn't have, all we have to do is update policy: the reachability information does not change. If later they should be denied the ability to communicate, the policy is updated again, and again the reachability doesn't have to change.

6.8 "How does Calico interact with the Neutron API?"

The How Calico Interprets Neutron API Calls document goes into extensive detail about how various Neutron API calls translate into Calico actions.

Calico is under active and rapid development. Here are our thoughts on some of the features that Calico doesn't support today, but likely will do soon.

Note: This is the documentation for version 1.4.0 of Calico.

CHAPTER 7

Future Plans

This section of the documentation covers proposals for the future of Calico. Feel free to use these documents to get an understanding of how the Calico team plan to add new features and functionality. Please also do let us know if any of these are of particular interest to you, or if you have other Calico-related requirements that are not yet covered here.

Note: This is the documentation for version 1.4.0 of Calico.

7.1 Overlapping IPv4 address ranges

This document describes how we can use 464XLAT to allow multiple tenants within the same data center to use the same IPv4 address ranges (including actually using the same IPv4 addresses). This is known as "overlapping" IPv4 addresses, and also as "address space isolation", because it means that an IP address such as 10.10.0.2 (for example) for one tenant has nothing to do with the same IP address being used by a different tenant.

7.1.1 RFC 6877

464XLAT is specified by RFC 6877 and describes how an IPv4-based application on a client device can access IPv4-based services somewhere in the Internet, via an IPv6 network.

- An IPv4 packet sent from the client's application is translated (statelessly, per RFC 6145) into an IPv6 packet.

```
IPv4 packet           IPv6 packet
SRC 192.168.0.2 ----> SRC 2001:db8:a41:23::192.168.0.2
DST 72.51.34.34       DST 2001:db8:a41:23::72.51.34.34
```

- The IPv6 packet is routed over the IPv6 network to the relevant server.
- The IPv6 packet is translated back into an IPv4 packet. This step needs to be stateful (per RFC 6146) because an arbitrary number of clients can connect to the server, and it isn't possible to map all their possible IPv6 addresses onto a range of IPv4 addresses in any way that's reversible without state.

```
IPv6 packet                           IPv4 packet
SRC 2001:db8:a41:23::192.168.0.2 ----> SRC 192.168.11.5
DST 2001:db8:a41:23::72.51.34.34       DST 72.51.34.34
```

- The IPv4 packet is delivered to the server application, and the server application responds.
- The response IPv4 packet is translated into an IPv6 packet. This uses the state established by the incoming packet, in particular to translate the response packet's destination IPv4 address to an IPv6 address.

```
IPv4 packet            IPv6 packet
SRC 72.51.34.34  ----> SRC 2001:db8:a41:23::72.51.34.34
DST 192.168.11.5       DST 2001:db8:a41:23::192.168.0.2
```

- The response IPv6 packet is routed over the IPv6 network back to the client.
- The response IPv6 packet is translated back into an IPv4 packet.

```
IPv6 packet                              IPv4 packet
SRC 2001:db8:a41:23::72.51.34.34 ----> SRC 72.51.34.34
DST 2001:db8:a41:23::192.168.0.2       DST 192.168.0.2
```

- The response IPv4 packet is delivered to the client application.

7.1.2 Data center usage

The use of 464XLAT for overlapping IPv4 addresses in a data center is largely similar. In particular:

- An IPv4 packet coming from an instance is translated to IPv6 by the compute host.
- The core transport of the data center, i.e. the L3 network connecting the compute hosts to each other, is IPv6 only.
- An IPv6 packet that arrives on a compute host, and that is destined for an instance on that compute host, is translated back to an IPv4 packet and then routed to the relevant instance.

Allowing multiple tenants to use the same IPv4 address ranges is achieved by including a number in the IPv6 translation that represents the tenant - or more generally, that represents the address space. So, an IPv4 address W.X.Y.Z, in address space <ID>, would be mapped to the IPv6 address:

```
<prefix>:<ID>::W.X.Y.Z
```

The address space needs to be associated, on each compute host, with the TAP interfaces of all VM ports whose IPv4 addresses are to be considered as belonging to that address space. Then, when an IPv4 packet is received by the compute host on one of those TAP interfaces, it can be translated to an IPv6 packet whose addresses contain the correct <ID>.

There are many possible mappings between tenants and address spaces.

- Complete tenant isolation corresponds to an address space that is only allowed to be used by that tenant.
- The current non-isolated Calico model corresponds to an address space that is shared across all tenants.
- Between these extremes, it is also possible for an address space to be shared by a specific group of tenants.

Also note that a given tenant may use multiple address spaces – for example, one that is private to itself, and one that is shared.

7.1.3 Compute host processing

How might this look in detail, on a given compute host?

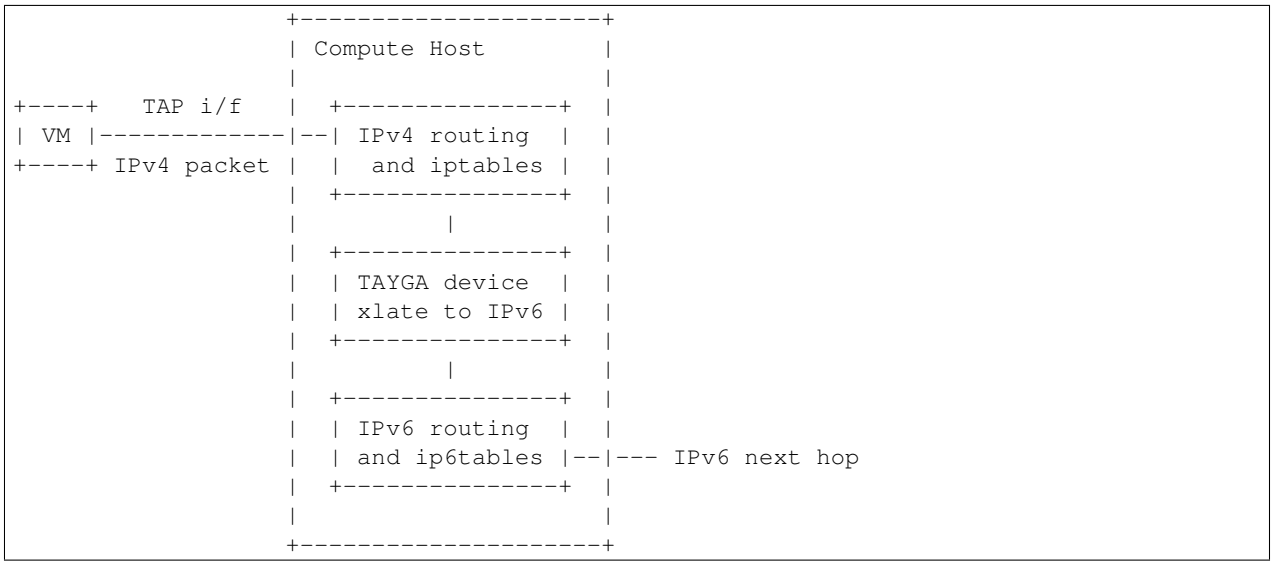
TAYGA (http://www.litech.org/tayga/) is an open source daemon that translates between IPv4 and IPv6. It presents itself as a network device, to which packets wanting translation should be routed. A packet received from a VM will pass through Linux routing and iptables processing twice: once as an IPv4 packet, which routing should direct into the TAYGA device; and then again as an IPv6 packet, which routing should forward to the IPv6 next hop.

Routing namespaces may be needed, as the compute host may have VMs using the same IPv4 addresses in different address spaces; for example two packets, both addressed to 10.10.0.2 but from different VMs, might need to be

translated using different address space IDs. Possibly this might be achievable by some marking scheme instead of by using namespaces - TBD.

To route translated packets across the network between compute hosts, BGP must distribute IPv6-translated addresses for instances, instead of the original IPv4 addresses.

So the picture for processing a packet from a VM looks like this:



For a packet received on a compute host, the first step is to decide whether the packet's destination IPv6 address maps to one of that compute host's VMs, and if so directing it into the TAYGA device for translation. This can be done with routing table entries like those that Calico programs today, but with IPv6 addresses and pointing to TAYGA instead of down TAP interfaces.

After translation back to IPv4, the traditional Calico routing rules will route down the correct TAP interface. Except that we have the namespace problem again: if there are two local VMs with the same address, which of them should get the packet?

7.1.4 Further study and questions

Further work will be needed on (at least) the following points.

Pin down the use of namespaces and/or an alternative marking scheme, on the compute host. If multiple namespaces are used, does this require a separate TAYGA device in each namespace?

One key difference between the RFC 6877 client-server scenario and the data center scenario is that in the data center case we expect that the IPv6->IPv4 translation can be stateless. Broadly, because all of the possible IPv4 source addresses can be represented as themselves on the destination compute host. Need to further pin down and describe precisely whether and why this is true.

How does IPv4<->6 translation interact with external access? Or, how does a VM with an overlapped IPv4 address communicate with an IPv4-based server outside the data center? I think that answering this depends on first pinning down our more general external access story.

Calico stands on the shoulders of many open source projects, as well as being open source itself.

Note: This is the documentation for version 1.4.0 of Calico.

CHAPTER 8

License and Acknowledgements

Calico's license is documented in our license.

It also makes use of other open source components as acknowledged in `licenses`.

R

RFC
RFC 1966, 54
RFC 4893, 49