MIRANTIS

MCP Networking

Overlay network solutions for OpenStack and Kubernetes supported by Mirantis Cloud Platform (MCP)

training.mirantis.com

Objectives

- Understand basic networking architectures for MCP clusters (OpenStack, Kubernetes)
- Know the best practices for configuring networks in MCP clusters

MCP Networking Highlights

- OpenStack
  - OpenContrail
  - Neutron with Open vSwitch
- Kubernetes
  - Calico

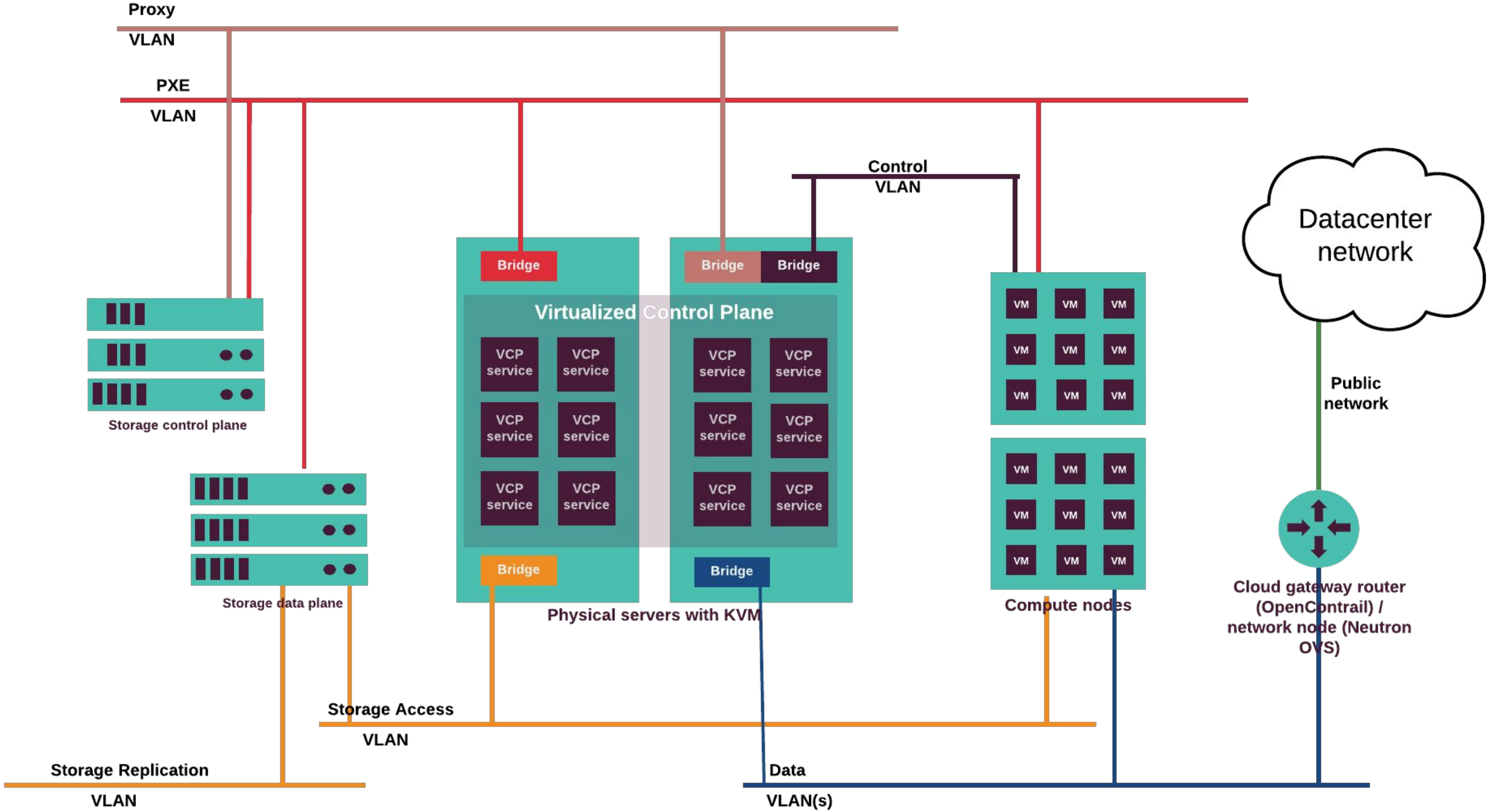
OpenStack Networking

Overlay network solutions for OpenStack supported by MCP

MCP Network Reference Architecture

- OpenContrail (Recommended)
- Neutron with Open vSwitch
  - VxLAN or VLAN
  - DVR or no DVR

MCP Networking Reference Architecture



MCP Underlay Networks

- PXE / Management
- Public
- Proxy
- Control
- Data
- Storage access (optional)
- Storage replication (optional)

MCP Networking Recommendations

- Isolated virtual or physical networks for PXE, Proxy, and all other traffic.
- Recommended minimal networking configuration:
    - PXE network – one 1 Gbit
    - Other networks – two bonded 10 Gbit
    - Use VLANs to isolate Proxy, Data, and Control networks

MCP Network Node Configuration

Port	Description	IP Address	VLAN
br-mesh	Tenant overlay traffic (VXLAN)	Routed, Subnet	Leaf switch only
br-mgmt	Openstack and other management traffic	Routed, Subnet	Leaf switch only
br-stor	Storage traffic	Routed, Subnet	Leaf switch only
eth0	PXE Boot traffic	VLAN, Subnet, Default	Global
br-prv	Tenant VLAN traffic bridge	VLAN range	Global
br-floating	External VLAN traffic bridge	VLAN range	Global

MCP VCP Network Configuration

Port	Description	IP Address	VLAN
br-pxe	PXE Boot traffic	VLAN, Subnet	Global
br-mgmt	Openstack and other management traffic	Routed, Subnet	Leaf switch only
br-stor	Storage traffic	Routed, Subnet	Leaf switch only
eth0	PXE Boot traffic	VLAN, Subnet, Default	Global

OpenStack with OpenContrail

OpenStack overlay networking with OpenContrail

OpenContrail only features

- Service chaining
- MPLS over UDP/GRE with vMX router
- Multi-site SDN
- Network analytics

OpenContrail vs Juniper Contrail

- Both are based on the same open source code
- Integration with MCP
  - OpenContrail is a part of MCP
  - Extra integration work will need to be done for Juniper Contrail
- Updates
  - Juniper Contrail users are tied to Juniper's release cycle
  - Mirantis can deliver package updates for OpenContrail in a matter of hours

Clear criteria for using Juniper Contrail

- Using Juniper for everything
- OpenStack SDN needs to integrate into the existing hardware and orchestrate special functions, such as
  - extending tenant networks across WAN connectivity with MPLS
  - having those links dynamically change based on events within Contrail

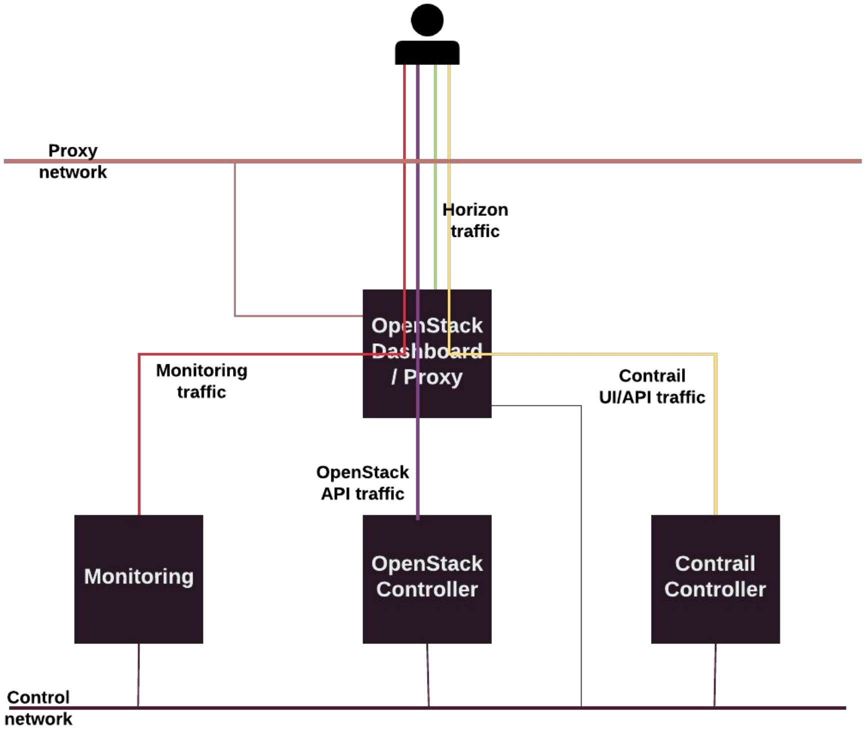
Mirantis OpenContrail

- There are no 'community' OpenContrail packages
- Mirantis OpenContrail repository
  - https://github.com/Mirantis/contrail-packages
- Mirantis consumes upstream OpenContrail repositories as follows:
  - Stores them on a mirror, and mirror output of Mirantis' branches to GitHub for the community
  - Fixes issues and cherry-picks from the master and most recent branches for backports
  - Runs daily builds from stable branches
  - Mirantis' production repositories publicly available via Launchpad

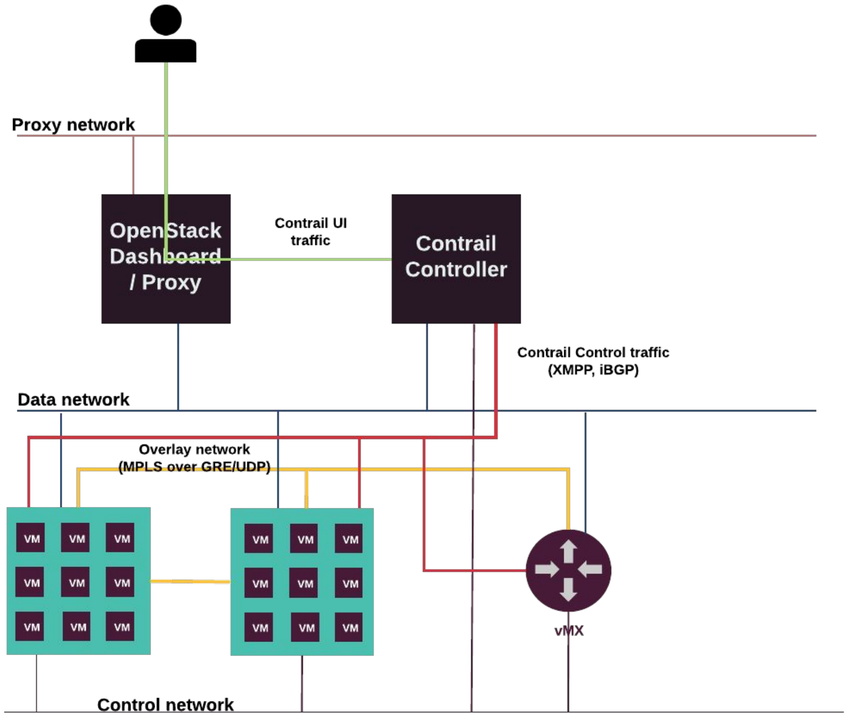
OpenContrail limitations

- Exit routers
  - At the moment only Juniper MX and vMX (hardware and software routers) are officially supported

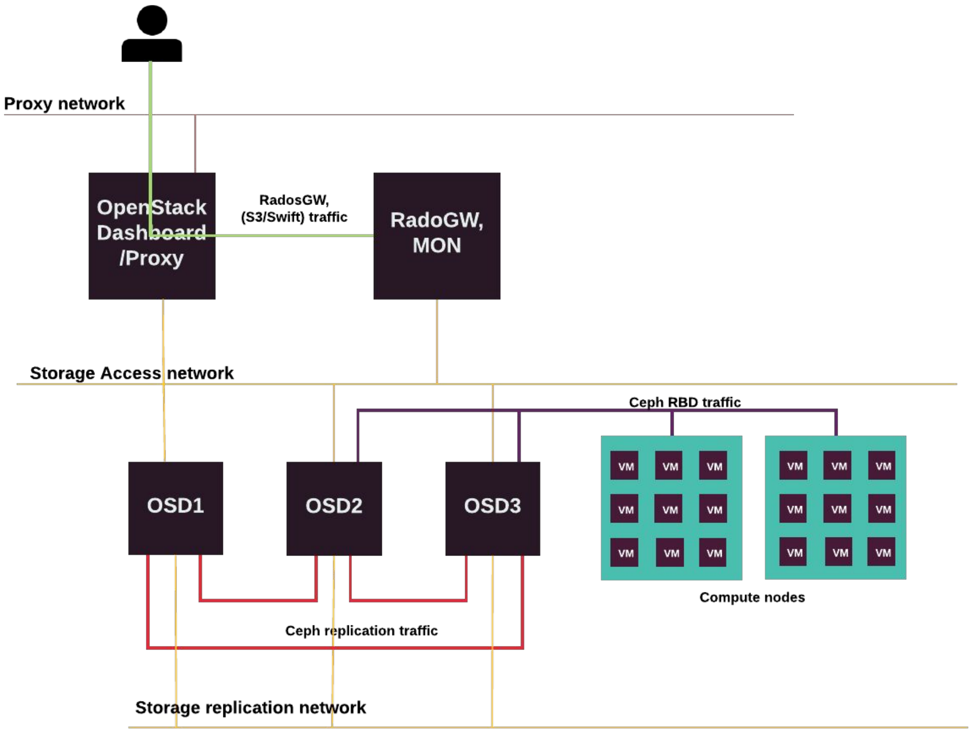
OpenContrail: User Interface and API traffic



OpenContrail: SDN traffic



OpenContrail: Storage traffic



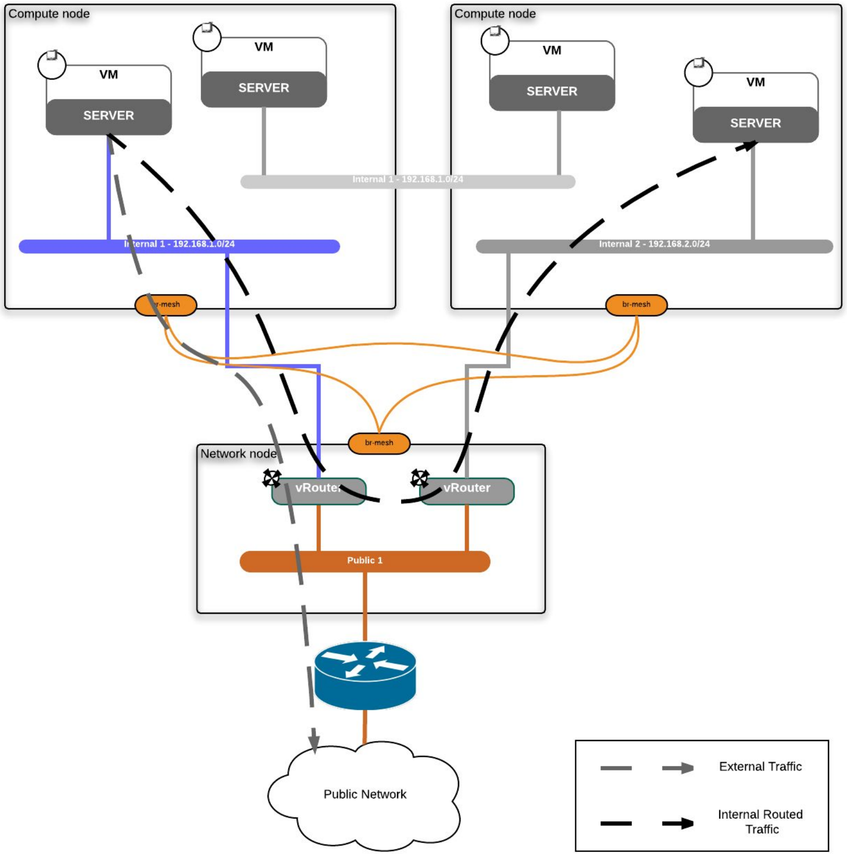
OpenStack with Open vSwitch

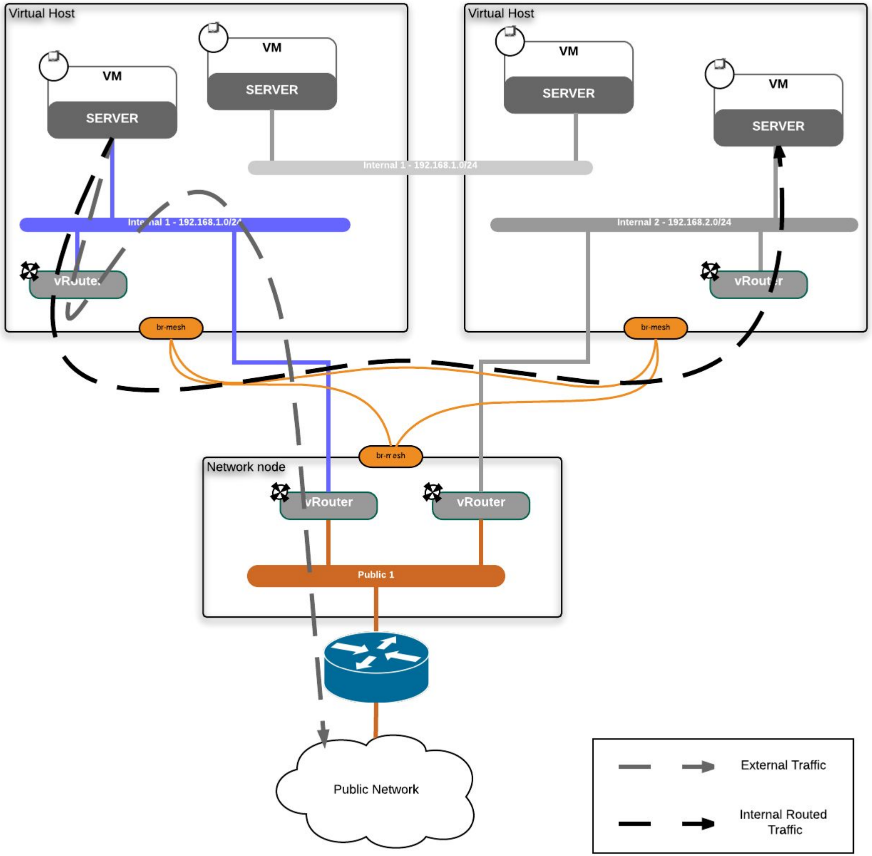
OpenStack overlay networking with Open vSwitch

MCP Deployment Models for Open vSwitch

- Neutron VXLAN tenant networks
  - Network Nodes for East-West and North-South
  - DVR for East-West and Network nodes for North-South*
  - DVR for East-West and North-South, Network Node for SNAT**
- Neutron VLAN tenant networks
  - Network Nodes for East-West and North-South**
  - DVR for East-West and North-South, Network Node for SNAT**

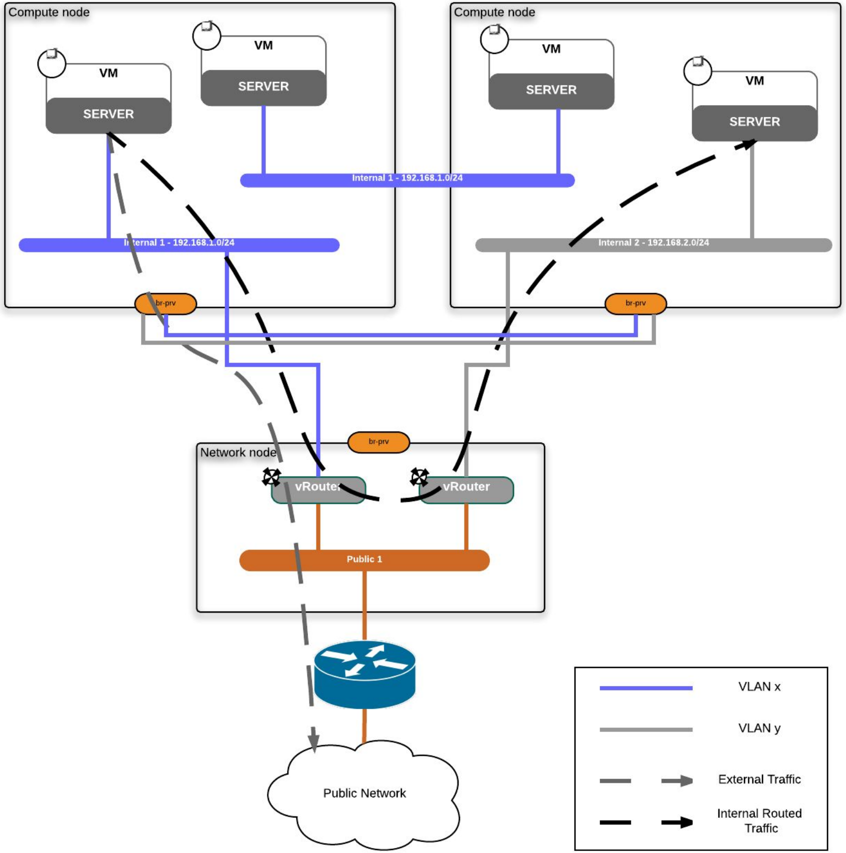
VXLAN, Network Nodes for East-West and North-South



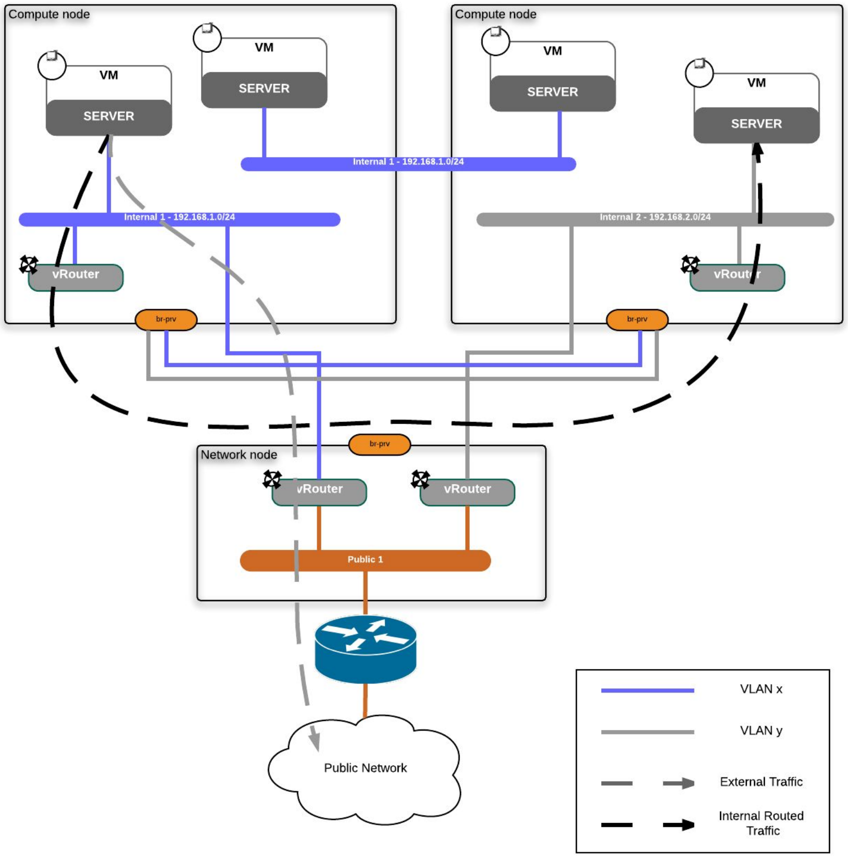


Virtual Host
VM
SERVER
VM
SERVER
Internal 1 - 192.168.1.0/24
vRouter
br-mesh
Virtual Host
VM
SERVER
VM
SERVER
Internal 1 - 192.168.1.0/24
Internal 2 - 192.168.2.0/24
vRouter
br-mesh
br-mesh
Network node
vRouter
vRouter
Public 1
Public Network
External Traffic
Internal Routed Traffic

VLAN, Network Nodes for East-West and North-South



VLAN, DVR for East-West and North-South, Network Node for SNAT



MCP Definition of Bridges

```
cat ./system/linux/network/interface/single_ovs_dvr.yml
parameters:
  linux:
    network:
      bridge: openvswitch
      interface:
        primary_interface:
          name: ${_param:primary_interface}
          type: eth
          mtu: ${_param:interface_mtu}
        tenant_interface:
          name: ${_param:tenant_interface}
          type: eth
          mtu: ${_param:interface_mtu}
          proto: manual
```

```
        br-int:
          enabled: true
          mtu: ${_param:interface_mtu}
          type: ovs_bridge
        br-floating:
          enabled: true
          mtu: ${_param:interface_mtu}
          type: ovs_bridge
        float-to-ex:
          enabled: true
          type: ovs_port
          mtu: 65000
          bridge: br-floating
```

Kubernetes with Calico

Kubernetes overlay networking with Calico

Kubernetes networking backends

- Plugins that implement Container Networking Interface (CNI)
- CNI plugins are typically executable scripts + config files
- Which are called by kubelet throughout pod lifecycle management i.e. when pod is created and deleted

Types of Kubernetes networking

- **Underlay**
  Pod interfaces are plugged into network fabric (by virtual switch) and traffic is typically bridged across network fabric (works with physical L2 fabric or cloud)
- **Overlay**
  Traffic from pod interfaces is sent encapsulated (typically by kernel) over tunnels across network fabric (works with all types of fabrics)
- **Native routing**
  Traffic from pods/containers interfaces is natively (by kernel) routed/forwarded from nodes across network fabric

Native routing vs. other types

Native routing has many advantages when compared with underlay and overlay:

- it uses native Linux kernel routing/forwarding capabilities
- no need for additional software switch/router
- it has low overhead
- it works with any type of fabric

What is Calico

Calico provides:

- SDN – provides connectivity between pods (based on native routing)
  - by exchanging /32 pod IP routes using BGP, and
  - optionally provides IPAM
- Network policy – to restrict traffic between pods (using iptables)

Both functionalities can be used together or separately:

- SDN + policy from Calico
- SDN from Calico alone (without policy)
- SDN from Flannel and policy from Calico

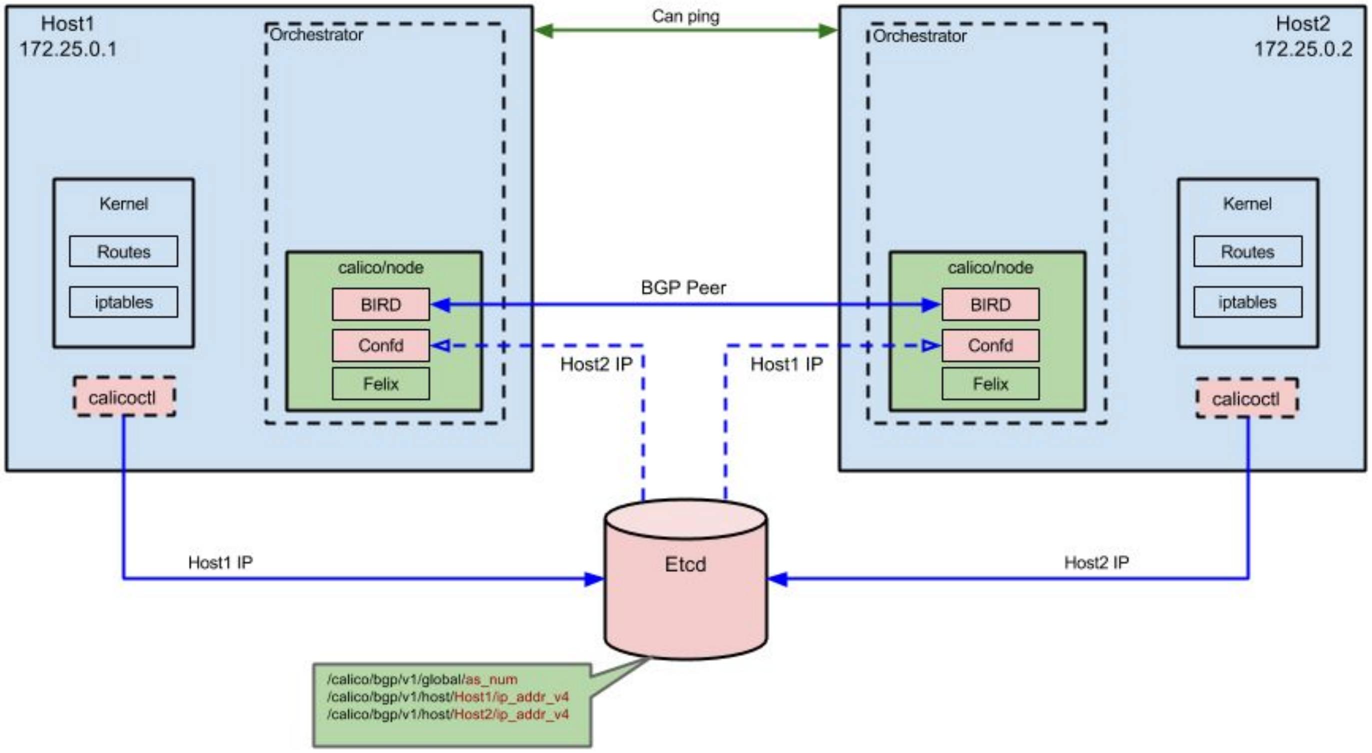
- Control plane
    - BGP routing daemon – bird or gobgp (experimental), it is optional – not needed when Calico is used to provide policy, not SDN
    - Confd – monitor and store config for bird on etcd
- Management plane
    - Felix – main Calico daemon
    - Calicoctl - command line tool to query/modify Calico API primitives/resources on Etcd (supports YAML)
- API (RESTful), internal message bus (REST pub-sub) and persistent storage
    - Etcd – used as external API server, internal message bus and persistent storage, all the internal and external communication on Calico goes through etcd

Calico architecture - Data plane

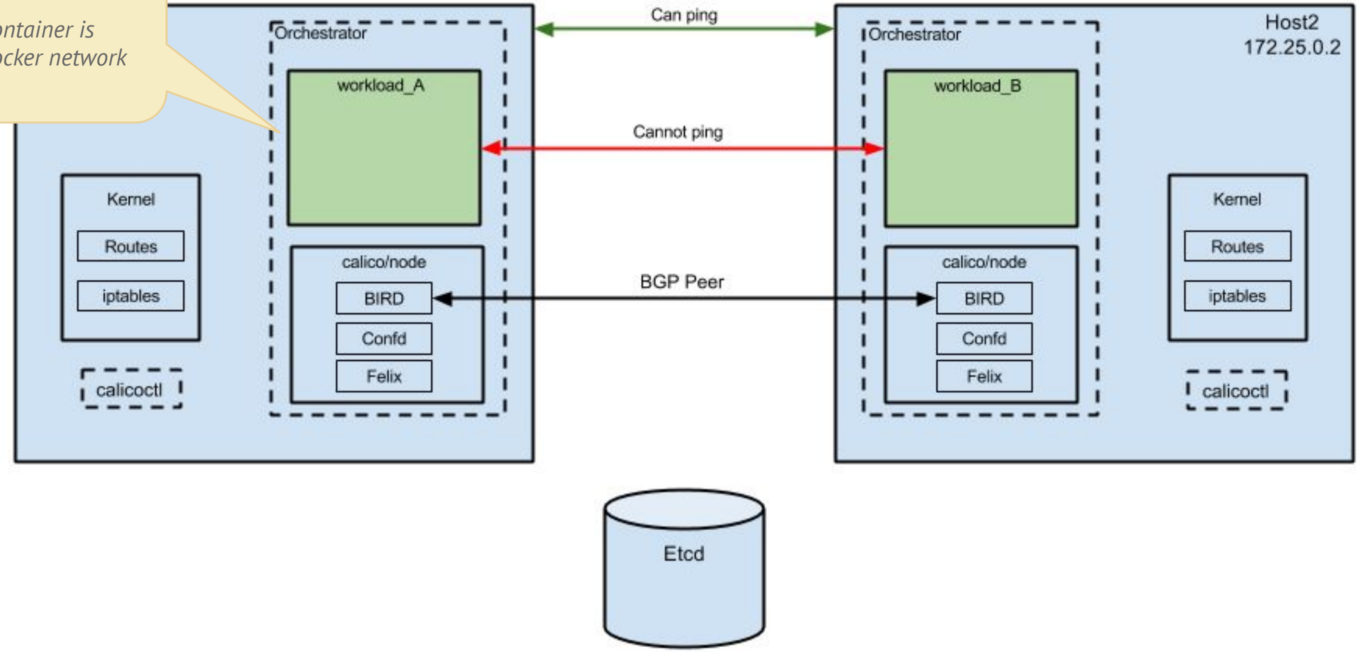
- standard kernel forwarding (native L3 routing)
- iptables – policy
- veth is set with arp proxy enabled

Calico integration with Kubernetes

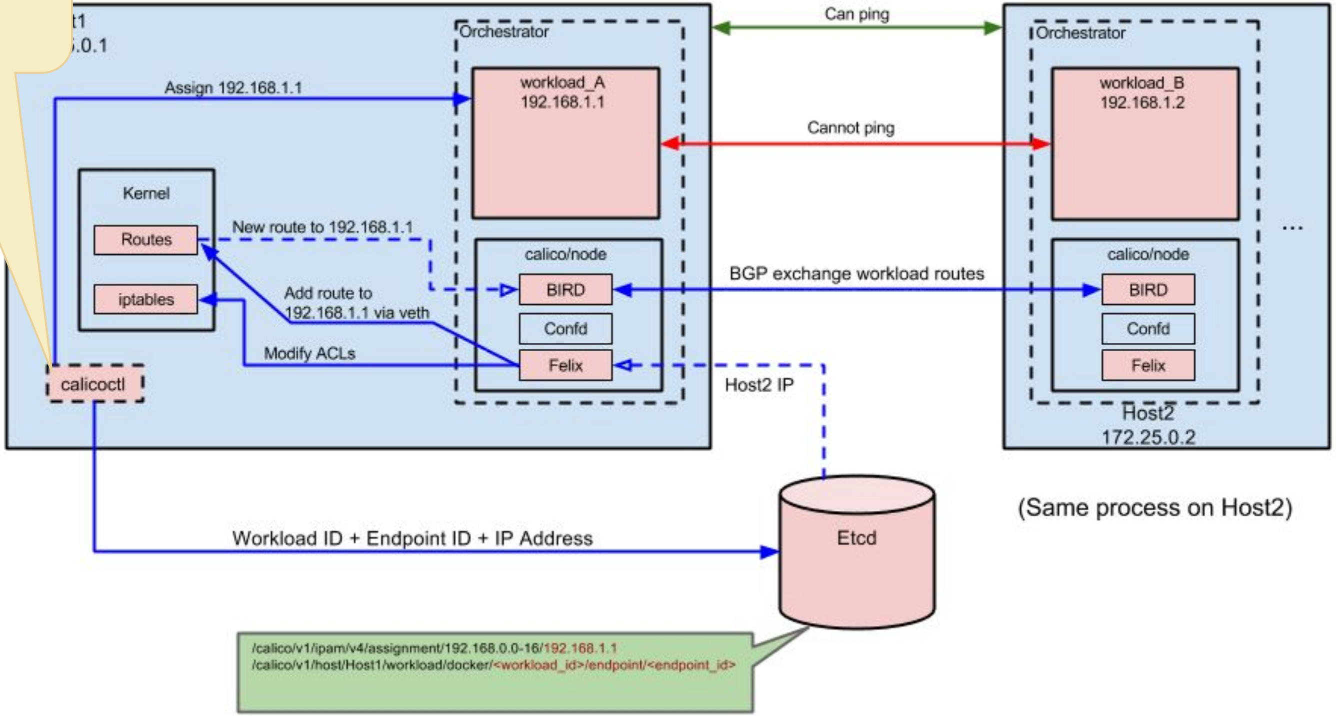
- SDN
  - calico CNI (SDN) plugin
  - calico CNI IPAM plugin (optional)
  - CNI config file
- Policy
  - policy controller translates Kubernetes policy (from Kubernetes API) into Calico policy stored on etcd
- Service IPs and local internal load balancing
  - are configured by Kubernetes on its own, using kube-proxy and iptables

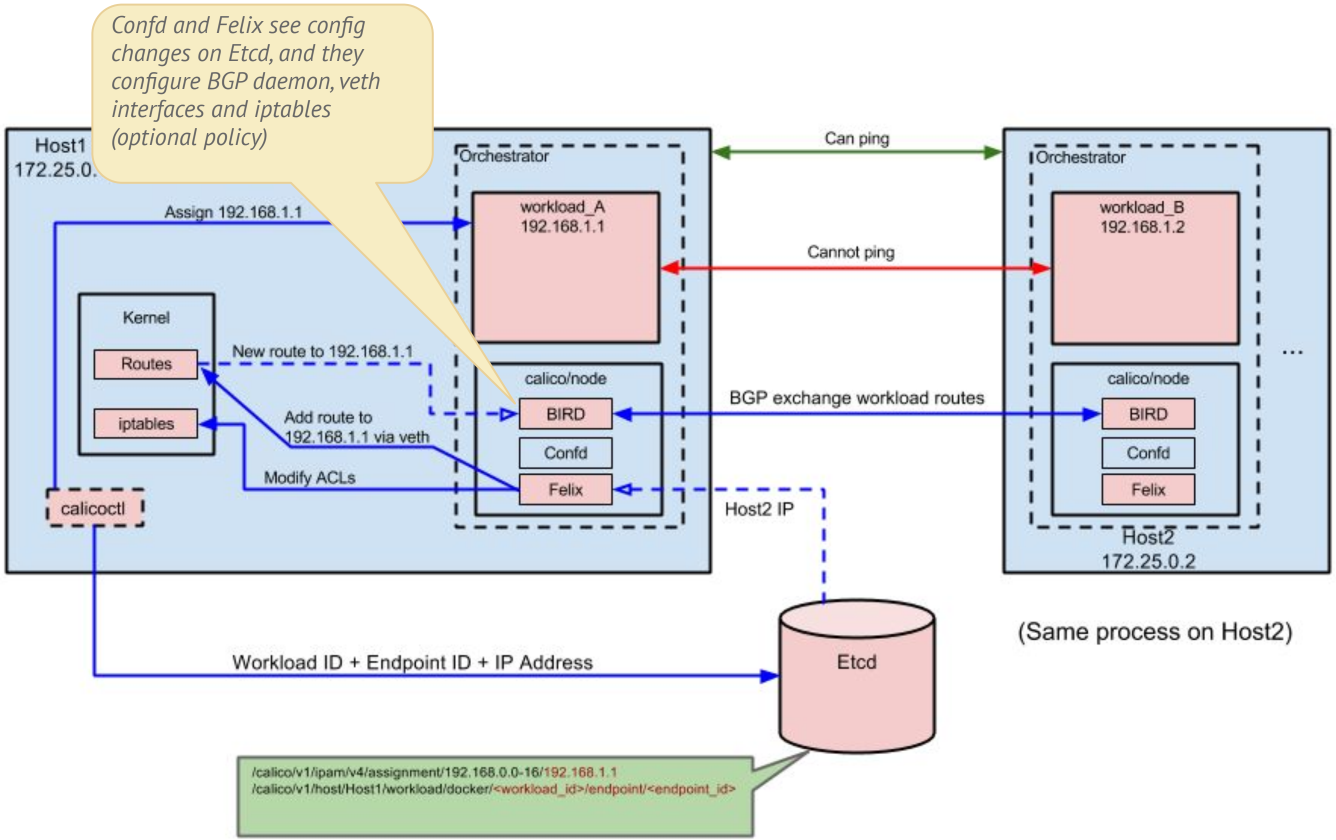

Host1
172.25.0.1
Orchestrator
Can ping
Orchestrator
Host2
172.25.0.2
Kernel
Routes
iptables
calico/node
BIRD
Confd
Felix
BGP Peer
Host2 IP
Host1 IP
calico/node
BIRD
Confd
Felix
Kernel
Routes
iptables
calicoctl
calicoctl
Etcd
Host1 IP
Host2 IP
/calico/bgp/v1/global/as_num
/calico/bgp/v1/host/Host1/ip_addr_v4
/calico/bgp/v1/host/Host2/ip_addr_v4

Pods (workload And B) are created by Kubernetes - orchestrator).
Pause Docker container is connected to Docker network none

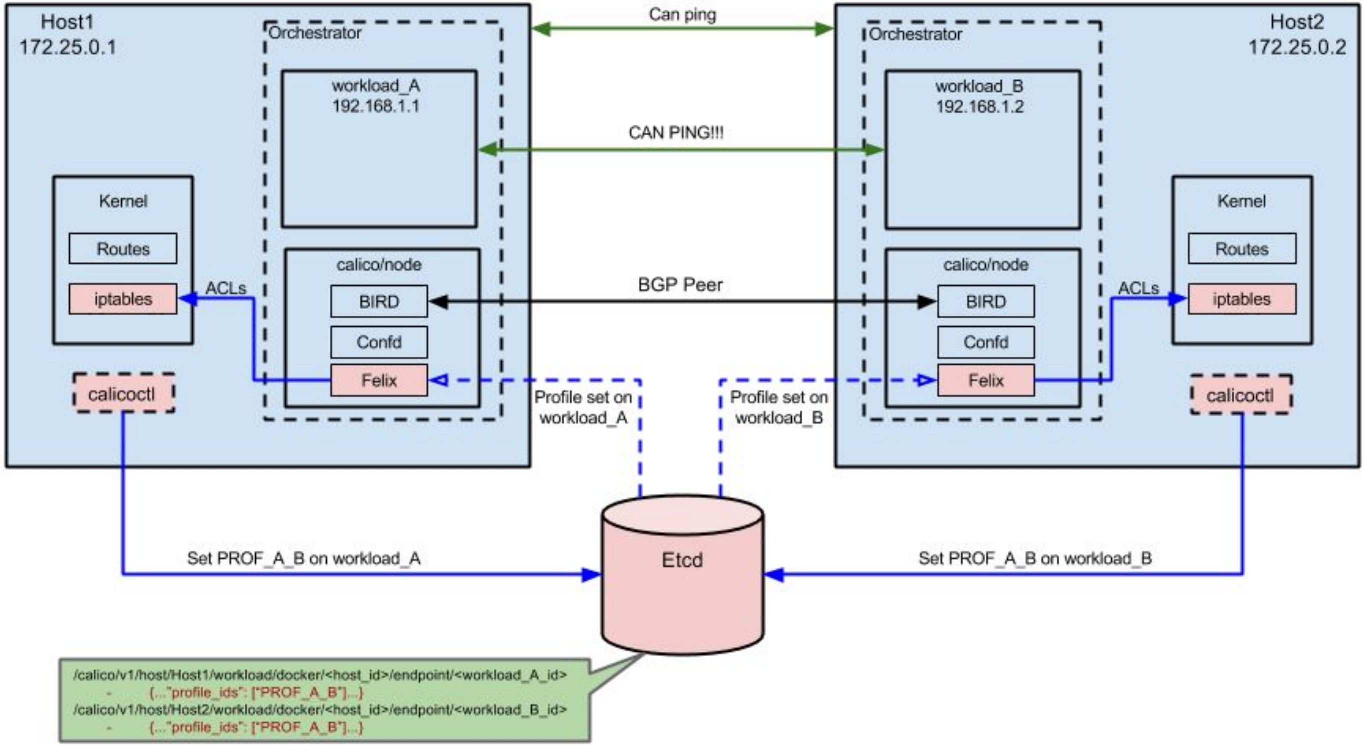


Kubernetes (kubelet) calls Calico CNI plugins. IPAM plugin is optional. Plugins call Calico API (etcd) to create configuration.





Confd and Felix see config changes on Etcd, and they configure BGP daemon, veth interfaces and iptables (optional policy)
Host1
172.25.0.
Orchestrator
workload_A
192.168.1.1
Assign 192.168.1.1
Kernel
Routes
iptables
New route to 192.168.1.1
Add route to 192.168.1.1 via veth
Modify ACLs
calico/node
BIRD
Confd
Felix
calicoctl
Can ping
Cannot ping
BGP exchange workload routes
Host2 IP
Orchestrator
workload_B
192.168.1.2
calico/node
BIRD
Confd
Felix
...
Host2
172.25.0.2
Workload ID + Endpoint ID + IP Address
Etcd
(Same process on Host2)
/calico/v1/ipam/v4/assignment/192.168.0.0-16/192.168.1.1
/calico/v1/host/Host1/workload/docker/<workload_id>/endpoint/<endpoint_id>


Host1
172.25.0.1
Orchestrator
workload_A
192.168.1.1
Kernel
Routes
iptables
ACLs
calico/node
BIRD
Confd
Felix
calicoctl
Can ping
CAN PING!!!
BGP Peer
Profile set on workload_A
Profile set on workload_B
Host2
172.25.0.2
Orchestrator
workload_B
192.168.1.2
calico/node
BIRD
Confd
Felix
Kernel
Routes
iptables
ACLs
calicoctl
Etcd
Set PROF_A_B on workload_A
Set PROF_A_B on workload_B
/calico/v1/host/Host1/workload/docker/<host_id>/endpoint/<workload_A_id>
- {..."profile_ids": ["PROF_A_B"]...}
/calico/v1/host/Host2/workload/docker/<host_id>/endpoint/<workload_B_id>
- {..."profile_ids": ["PROF_A_B"]...}

Calico deployment with MCP

- Docker container 'calico/node' runs on every Kubernetes node
  - it contains Calico control and management plane daemons
  - it is started using host network (node IP), so there is no egg chicken problem
- Since kubelet runs as a container, directory with CNI plugins and config file must be mounted as volume on kubelet container so they can be called/executed by kubelet
- Salt formula for Kubernetes does the magic
  - https://github.com/salt-formulas/salt-formula-kubernetes