MIRANTIS

Mirantis OpenStack Reference Architecture

version 9.2

Contents

Mirantis OpenStack Reference Architecture **1**
Ceph Monitors 1
Advanced Network Configuration using Open VSwitch 2
Reference Network Model in Neutron 2
Adjust the Network Configuration via CLI 2
The "Transformations" Section 3
Network aggregation 5
Types of network bonding 5
An Example of NIC Aggregation using Fuel CLI tools 6
How the Operating System Role is provisioned 7
Image Based Provisioning 8
Fuel Agent 8
Image building 9
Operating system provisioning 10
Viewing the control files on the Fuel Master node 11
Troubleshooting image-based provisioning 11
Fuel Repository Mirroring 12
Corosync Settings 13
Pacemaker Settings 14
How Fuel Deploys HA 14
HA deployment for Networking 14
Virtual IP addresses deployment details 14
TCP keepalive configuration details 15
Router 15
Switches 15

Ceph Monitors

Ceph monitors (MON) manage various maps like MON map, CRUSH map, and others. The CRUSH map is used by clients to deterministically select the storage devices (OSDs) to receive copies of the data. Ceph monitor nodes manage where the data should be stored and maintain data consistency with the Ceph OSD nodes that store the actual data.

Ceph monitors implement HA using a master-master model:

- One Ceph monitor node is designated the "leader." This is the node that first received the most recent cluster map replica.
- Each other monitor node must sync its cluster map with the current leader.
- Each monitor node that is already sync'ed with the leader becomes a provider; the leader knows which nodes are currently providers. The leader tells the other nodes which provider they should use to sync their data.

Ceph Monitors use the Paxos algorithm to determine all updates to the data they manage. All monitors that are in quorum have consistent up-to-date data because of this.

You can read more in Ceph documentation.

Advanced Network Configuration using Open VSwitch

The Neutron networking model uses Open VSwitch (OVS) bridges and the Linux namespaces to create a flexible network setup and to isolate tenants from each other on L2 and L3 layers. Mirantis OpenStack also provides a flexible network setup model based on Open VSwitch primitives, which you can use to customize your nodes. Its most popular feature is link aggregation. While the FuelWeb UI uses a hardcoded per-node network model, the Fuel CLI tool allows you to modify it in your own way.

Note

When using encapsulation protocols for network segmentation, take header overhead into account to avoid guest network slowdowns from packet fragmentation or packet rejection. With a physical host MTU of 1500 the maximum instance (guest) MTU is 1430 for GRE and 1392 for VXLAN. When possible, increase MTU on the network infrastructure using jumbo frames. The default OpenVSwitch behavior in Mirantis OpenStack 6.0 and newer is to fragment packets larger than the MTU. In prior versions OpenVSwitch discards packets exceeding MTU. See the Official OpenStack documentation for more information.

This section includes the following topics:

Reference Network Model in Neutron

The FuelWeb UI uses the following per-node network model:

- Create an OVS bridge for each NIC except for the NIC with Admin network (for example, br-eth0 bridge for eth0 NIC) and put NICs into their bridges
- Create a separate bridge for each OpenStack network:
    - br-ex for the Public network
    - br-prv for the Private network
    - br-mgmt for the Management network
    - br-storage for the Storage network
- Connect each network's bridge with an appropriate NIC bridge using an OVS patch with an appropriate VLAN tag.
- Assign network IP addresses to the corresponding bridges.

Note that the Admin network IP address is assigned to its NIC directly.

This network model allows the cluster administrator to manipulate cluster network entities and NICs separately, easily, and on the fly during the cluster lifecycle.

Adjust the Network Configuration via CLI

On a basic level, this network configuration is part of a data structure that provides instructions to the Puppet modules to set up a network on the current node. You can examine and modify this data using the Fuel CLI tool. Just download (then modify and upload if needed) the environment's 'deployment default' configuration:

```
[root@fuel ~]# fuel --env 1 deployment default
directory /root/deployment_1 was created
Created /root/deployment_1/compute_1.yaml
Created /root/deployment_1/controller_2.yaml
[root@fuel ~]# vi ./deployment_1/compute_1.yaml
[root@fuel ~]# fuel --env 1 deployment --upload
```

Note

Read Fuel CLI reference in Fuel User Guide.

The part of this data structure that describes how to apply the network configuration is the 'network_scheme' key in the top-level hash of the YAML file. Let's take a closer look at this substructure. The value of the 'network_scheme' key is a hash with the following keys:

- interfaces - A hash of NICs and their low-level/physical parameters. You can set an MTU feature here.
- provider - Set to 'ovs' for Neutron.
- endpoints - A hash of network ports (OVS ports or NICs) and their IP settings.
- roles - A hash that specifies the mappings between the endpoints and internally-used roles in Puppet manifests ('management', 'storage', and so on).
- transformations - An ordered list of OVS network primitives.

See the example of a network_scheme section in a node's configuration, showing how to change MTU parameters: the MTU parameters configuration example.

The "Transformations" Section

You can use four OVS primitives:

- add-br - To add an OVS bridge to the system
- add-port - To add a port to an existent OVS bridge
- add-bond - To create a port in OVS bridge and add aggregated NICs to it
- add-patch - To create an OVS patch between two existing OVS bridges

The primitives will be applied in the order they are listed.

Here are the the available options:

```
{
  "action": "add-br",         # type of primitive
  "name": "xxx",              # unique name of the new bridge
  "provider": "ovs"           # type of provider `linux` or `ovs`
},
{
  "action": "add-port",       # type of primitive
  "name": "xxx-port",         # unique name of the new port
  "bridge": "xxx",           # name of the bridge where the port should be created
  "type": "internal",        # [optional; default: "internal"] a type of OVS
                             # interface # for the port (see OVS documentation);
                             # possible values:
                             # "system", "internal", "tap", "gre", "null"
  "tag": 0,                  # [optional; default: 0] a 802.1q tag of traffic that
                             # should be captured from an OVS bridge;
                             # possible values: 0 (means port is a trunk),
                             # 1-4094 (means port is an access)
  "trunks": [],              # [optional; default: []] a set of 802.1q tags
                             # (integers from 0 to 4095) that are allowed to
                             # pass through if "tag" option equals 0;
                             # possible values: an empty list (all traffic passes),
                             # 0 (untagged traffic only), 1 (strange behaviour;
                             # shouldn't be used), 2-4095 (traffic with this
                             # tag passes); e.g. [0,10,20]
  "port_properties": [],     # [optional; default: []] a list of additional
                             # OVS port properties to modify them in OVS DB
  "interface_properties": [], # [optional; default: []] a list of additional
                             # OVS interface properties to modify them in OVS DB
},
{
  "action": "add-bond",      # type of primitive
  "name": "xxx-port",        # unique name of the new bond
  "interfaces": [],          # a set of two or more bonded interfaces' names;
                             # e.g. ['eth1','eth2']
  "bridge": "xxx",           # name of the bridge where the bond should be created
  "tag": 0,                  # [optional; default: 0] a 802.1q tag of traffic which
                             # should be catched from an OVS bridge;
                             # possible values: 0 (means port is a trunk),
                             # 1-4094 (means port is an access)
  "trunks": [],              # [optional; default: []] a set of 802.1q tags
                             # (integers from 0 to 4095) which are allowed to
                             # pass through if "tag" option equals 0;
                             # possible values: an empty list (all traffic passes),
                             # 0 (untagged traffic only), 1 (strange behaviour;
                             # shouldn't be used), 2-4095 (traffic with this
                             # tag passes); e.g. [0,10,20]
  "properties": [],          # [optional; default: []] a list of additional
                             # OVS bonded port properties to modify them in OVS DB;
                             # you can use it to set the aggregation mode and
                             # balancing # strategy, to configure LACP, and so on
                             # (see the OVS documentation)
},
{
  "action": "add-patch",     # type of primitive
  "bridges": ["br0", "br1"], # a pair of different bridges that will be connected
  "peers": ["p1", "p2"],     # [optional] abstract names for each end of the patch
  "tags": [0, 0] ,           # [optional; default: [0,0]] a pair of integers that
                             # represent an 802.1q tag of traffic that is
                             # captured from an appropriate OVS bridge; possible
                             # values: 0 (means port is a trunk), 1-4094 (means
                             # port is an access)
  "trunks": [],               # [optional; default: []] a set of 802.1q tags
                             # (integers from 0 to 4095) which are allowed to
                             # pass through each bridge if "tag" option equals 0;
                             # possible values: an empty list (all traffic passes),
                             # 0 (untagged traffic only), 1 (strange behavior;
                             # shouldn't be used), 2-4095 (traffic with this
                             # tag passes); e.g., [0,10,20]
}
```

A combination of these primitives allows you to make custom and complex network configurations.

Network aggregation

Network aggregation or NIC bonding enables you to aggregate multiple physical links to one link to increase speed and provide fault tolerance in a systems that require maximum uptime.

This section includes the following topics:

Types of network bonding

Open VSwitch supports the same bonding features as the Linux kernel. Fuel supports bonding either via Open VSwitch or by using Linux native bonding interfaces. By default, Fuel uses the Linux native bonding provider.

Linux supports two types of bonding:

- IEEE 802.1AX (formerly known as 802.3ad) Link Aggregation Control Protocol (LACP). Devices on both sides of links must communicate using LACP to set up an aggregated link. So both devices must support LACP, enable and configure it on these links.
- One side bonding does not require any special feature support from the switch side. Linux handles it using a set of traffic balancing algorithms.

One Side Bonding Policies:

- Balance-rr - Round-robin policy. This mode provides load balancing and fault tolerance.
- Active-backup - Active-backup policy: Only one slave in the bond is active.This mode provides fault tolerance.
- Balance-xor - XOR policy: Transmit based on the selected transmit hash policy. This mode provides load balancing and fault tolerance.
- Broadcast - Broadcast policy: transmits everything on all slave interfaces. This mode provides fault tolerance.
- balance-tlb - Adaptive transmit load balancing based on a current links' utilization. This mode provides load balancing and fault tolerance.
- balance-alb - Adaptive transmit and receive load balancing based on the current links' utilization. This mode provides load balancing and fault tolerance.
- balance-slb - Modification of balance-alb mode. SLB bonding allows a limited form of load balancing without the remote switch's knowledge or cooperation. SLB assigns each source MAC+VLAN pair to a link and transmits all packets from that MAC+VLAN through that link.

Learning in the remote switch causes it to send packets to that MAC+VLAN through the same link.

- balance-tcp - Adaptive transmit load balancing among interfaces.

LACP Policies:

- Layer2 - Uses XOR of hardware MAC addresses to generate the hash.
- Layer2+3 - uses a combination of layer2 and layer3 protocol information to generate the hash.
- Layer3+4 - uses upper layer protocol information, when available, to generate the hash.
- Encap2+3 - uses the same formula as layer2+3 but it relies on skb_flow_dissect to obtain the header fields which might result in the use of inner headers if an encapsulation protocol is used. For example, this will improve the performance for tunnel users because the packets will be distributed according to the encapsulated flows.
- Encap3+4 - Similar to Encap2+3 but uses layer3+4.

Policies Supported by Fuel

Fuel supports the following policies: 802.3ad (LACP), balance-rr, active-backup. In addition, balance-xor, broadcast, balance-tlb, and balance-alb are supported in experimental mode.

Note

LACP rate can be set in Fuel web UI for the 802.3ad (LACP) mode. The possible values of the LACP rate are: slow and fast.

These interfaces can be configured in Fuel UI when nodes are being added to the environment or by using Fuel CLI and editing YAML configuration manually.

You can specify the transmit hash policy using Fuel web UI. You can choose from layer2, layer2+3, layer3+4, encap2+3, encap3+4 values for 802.3ad, balance-xor, balance-tlb, and balance-alb modes.

Note

The balance-xor, balance-tlb, and balance-alb modes are supported in the experimental mode.

Fuel has limited network verification capabilities when working with bonds. Network connectivity can be checked for the new cluster only (not for deployed one) so check is done when nodes are in bootstrap and no bonds are up. Connectivity between slave interfaces can be checked but not bonds themselves.

An Example of NIC Aggregation using Fuel CLI tools

Suppose you have a node with 4 NICs and you want to bond two of them with LACP enabled ("eth2" and "eth3" here) and then assign Private and Storage networks to them. The Admin network uses a dedicated NIC ("eth0"). The Management and Public networks use the last NIC ("eth1").

To create bonding interface using Open vSwitch, do the following:

- Create a separate OVS bridge "br-bondnew" instead of "br-eth2" and "br-eth3".
- Connect "eth2" and "eth3" to "br-bondnew" as a bonded port with property "lacp=active".
- Connect "br-prv" and "br-storage" bridges to "br-bondnew" by OVS patches.
- Leave all of the other things unchanged.

See the example of OVS network scheme section in the node configuration.

If you are going to use Linux native bonding, follow these steps:

- Create a new interface "bondnew" instead of "br-eth2" and "br-eth3".
- Connect "eth2" and "eth3" to "bondnew" as a bonded port.
- Add 'provider': 'lnx' to choose Linux native mode.
- Add properties as a hash instead of an array used in ovs mode. Properties are same as options used during the bonding kernel modules loading. You should provide which mode this bonding interface should use. Any other options are not mandatory. You can find all these options in the Linux kernel documentation.

  'properties':

  'mode': 1
- Connect "br-prv" and "br-storage" bridges to "br-bondnew" by OVS patches.
- Leave all of the other things unchanged.

See the example Linux network scheme.

How the Operating System Role is provisioned

Fuel provisions the Operating System role with either the CentOS or Ubuntu operating system that was selected for the environment but Puppet does not deploy other packages on this node or provision the node in any way.

The Operating System role is defined in the openstack.yaml file; the internal name is base-os. Fuel installs a standard set of operating system packages similar to what it installs on other roles; use the dpkg -l command on Ubuntu or the rpm -qa command on CentOS to see the exact list of packages that are installed.

A few configurations are applied to an Operating System role. For environments provisioned with the traditional tools, these configurations are applied by Cobbler snippets that run during the provisioning phase. When using image-based provisioning, cloud init applies these configurations. These include:

- Disk partitioning. The default partitioning allocates a small partition (about 15GB) on the first disk for the root partition and leaves the rest of the space unallocated; users can manually allocate the remaining space.
- The public key that is assigned to all target nodes in the environment
- The kernel parameters that are applied to all target nodes
- Network settings. Configure the Admin logical networks with a static IP address. No other networking is configured.

The following configurations that are set in the Fuel web UI have no effect on the Operating System role:

- Mapping of logical networks to physical interfaces. All connections for the logical networks that connect this node to the rest of the environment need to be defined.

Image Based Provisioning

Operating systems are usually distributed with their own installers (e.g. Anaconda or Debian-installer). Fuel 7.0 does not use these installers. Instead, Fuel 7.0 uses image based provisioning, which is a faster and enterprise-ready method.

Whereas installers like Anaconda or Debian-installer were used in older Fuel versions to build the operating system from scratch on each node using online or local repositories, with image based provisioning a base image is created and copied to each node to be used to deploy the operating system on the local disks.

Image based provisioning significantly reduces the time required for provisioning and it is more reliable to copy the same image on all nodes instead of building an operating system from scratch on each node.

Image based provisioning is implemented using the Fuel Agent. The image based provisioning process consists of two independent steps, which are:

1. Operating system image building.

This step assumes that we build an operating system image from a set of repositories in a directory which is then packed into the operating system image. The build script is run once no matter how many nodes one is going to deploy.

Ubuntu images are built on the master node, one operating system image per environment. We need to build different images for each environment because each environment has its own set of repositories. In order to deal with package differences between repository sets, we create an operating system image for each environment. When the user clicks the "Deploy changes" button, we check if the operating system package is already available for a particular environment, and if it is not, we build a new one just before starting the actual provisioning.

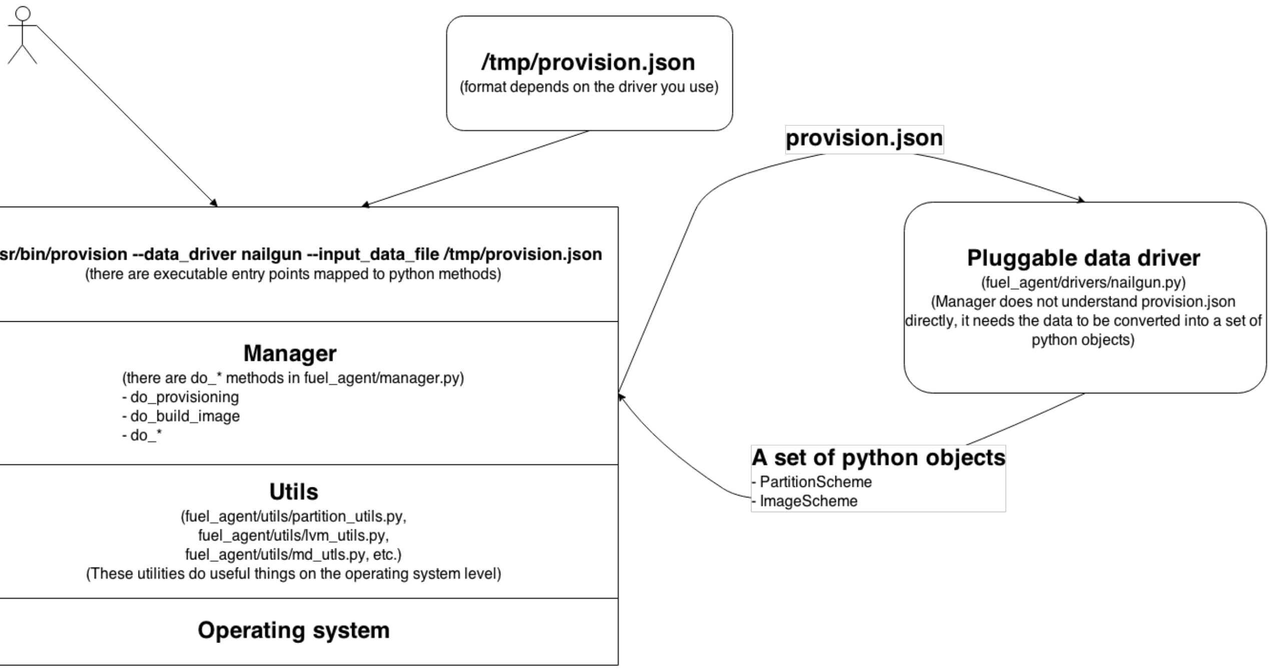
2. Copying of operating system image to nodes.

Operating system images that have been built can be downloaded via HTTP from the Fuel Master node. So, when a node is booted into the so called Bootstrap operating system, we can run an executable script to download the necessary operating system image and put it on a hard drive. We don't need to reboot the node into the installer OS like we do when we use an Anaconda or Debian-installer. Our executable script in this case plays the same role. We just need it to be installed into the Bootstrap operating system.

For both of these steps we have a special program component which is called Fuel Agent. Fuel Agent is nothing more than just a set of data driven executable scripts. One of these scripts is used for building operating system images and we run this script on the master node passing a set of repository URIs and a set of package names to it. Another script is used for the actual provisioning. We run it on each node and pass provisioning data to it. These data contain information about disk partitions, initial node configuration, operating system image location, etc. So, this script being run on a node, prepares disk partitions, downloads operating system images and puts these images on partitions. It is necessary to note that when we say operating system image we actually mean a set of images, one per file system. If, for example, we want / and /boot be two separate file systems, then this means we need to separate the operating system images, one for / and another for /boot. Images in this case are binary copies of corresponding file systems.

Fuel Agent

Fuel Agent is a set of data driven executable scripts. It is written in Python. Its high level architecture is depicted below:


/tmp/provision.json
(format depends on the driver you use)
provision.json
/usr/bin/provision --data_driver nailgun --input_data_file /tmp/provision.json
(there are executable entry points mapped to python methods)
Pluggable data driver
(fuel_agent/drivers/nailgun.py)
(Manager does not understand provision.json directly, it needs the data to be converted into a set of python objects)
Manager
(there are do_* methods in fuel_agent/manager.py)
- do_provisioning
- do_build_image
- do_*
A set of python objects
- PartitionScheme
- ImageScheme
Utils
(fuel_agent/utils/partition_utils.py, fuel_agent/utils/lvm_utils.py, fuel_agent/utils/md_utls.py, etc.)
(These utilities do useful things on the operating system level)
Operating system

When we run one of its executable entry, we pass the input data to it where it is written what needs to be done and how. We also point out which data driver it needs to use in order to parse these input data. For example:

```
/usr/bin/provision --input_data_file /tmp/provision.json --data_driver nailgun
```

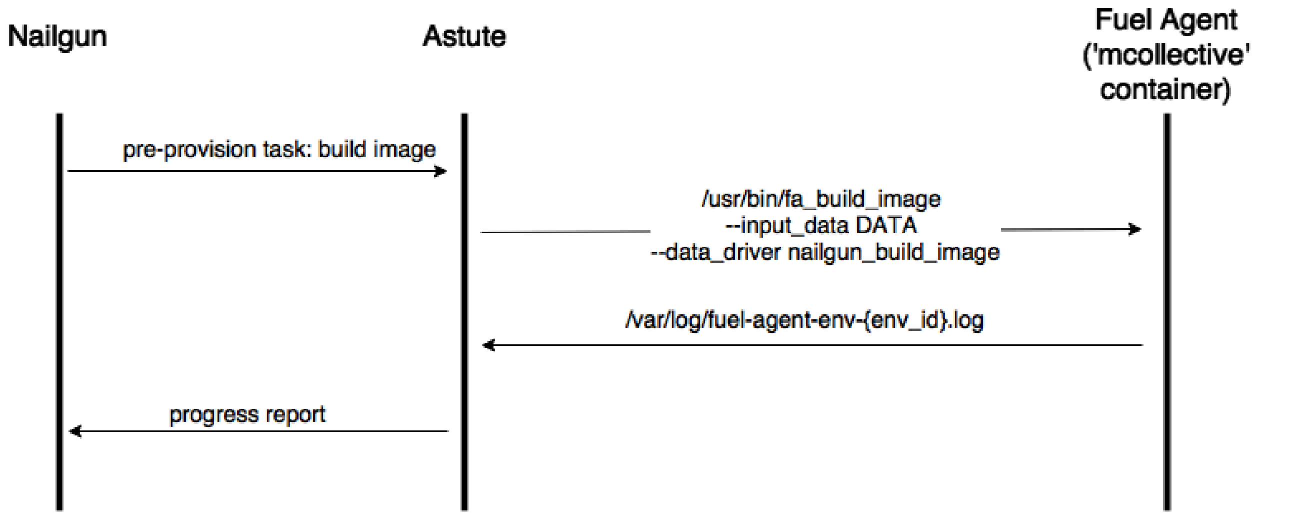
The heart of Fuel Agent is the manager fuel_agent/manager.py, which does not directly understand input data, but it does understand sets of Python objects defined in fuel_agent/objects. Data driver is the place where raw input data are converted into a set of objects. Using this set of objects manager then does something useful like creating partitions, building operating system images, etc. But the manager implements only high-level logic for all these cases and uses a low-level utility layer which is defined in fuel_agent/utils to perform real actions like launching parted or mkfs commands.

The Fuel Agent config file is located in /etc/fuel-agent/fuel-agent.conf. There are plenty of configuration parameters that can be set and all these parameters have default values which are defined in the source code. All configuration parameters are well commented.

The Fuel Agent leverages cloud-init for the Image based deployment process. It also creates a cloud-init drive which allows for post-provisioning configuration. The config drive uses jinja2 templates which can be found in /usr/share/fuel-agent/cloud-init-templates. These templates are filled with values given from the input data.

Image building

When Ubuntu based environment is being provisioned, there is a pre-provisioning task which runs the /usr/bin/fa_build_image script. This script is one of the executable Fuel Agent entry points. This script is installed in the 'mcollective' docker container on the Fuel master node. As input data we pass a list of Ubuntu repositories from which an operating system image is built and some other metadata. When launched, Fuel Agent checks if there is a Ubuntu image available for this environment and if there is not, it builds an operating system image and puts this image in a directory defined in the input data so as to make it available via HTTP. See the sequence diagram below:


Nailgun
Astute
Fuel Agent ('mcollective' container)
pre-provision task: build image
/usr/bin/fa_build_image --input_data DATA --data_driver nailgun_build_image
/var/log/fuel-agent-env-{env_id}.log
progress report

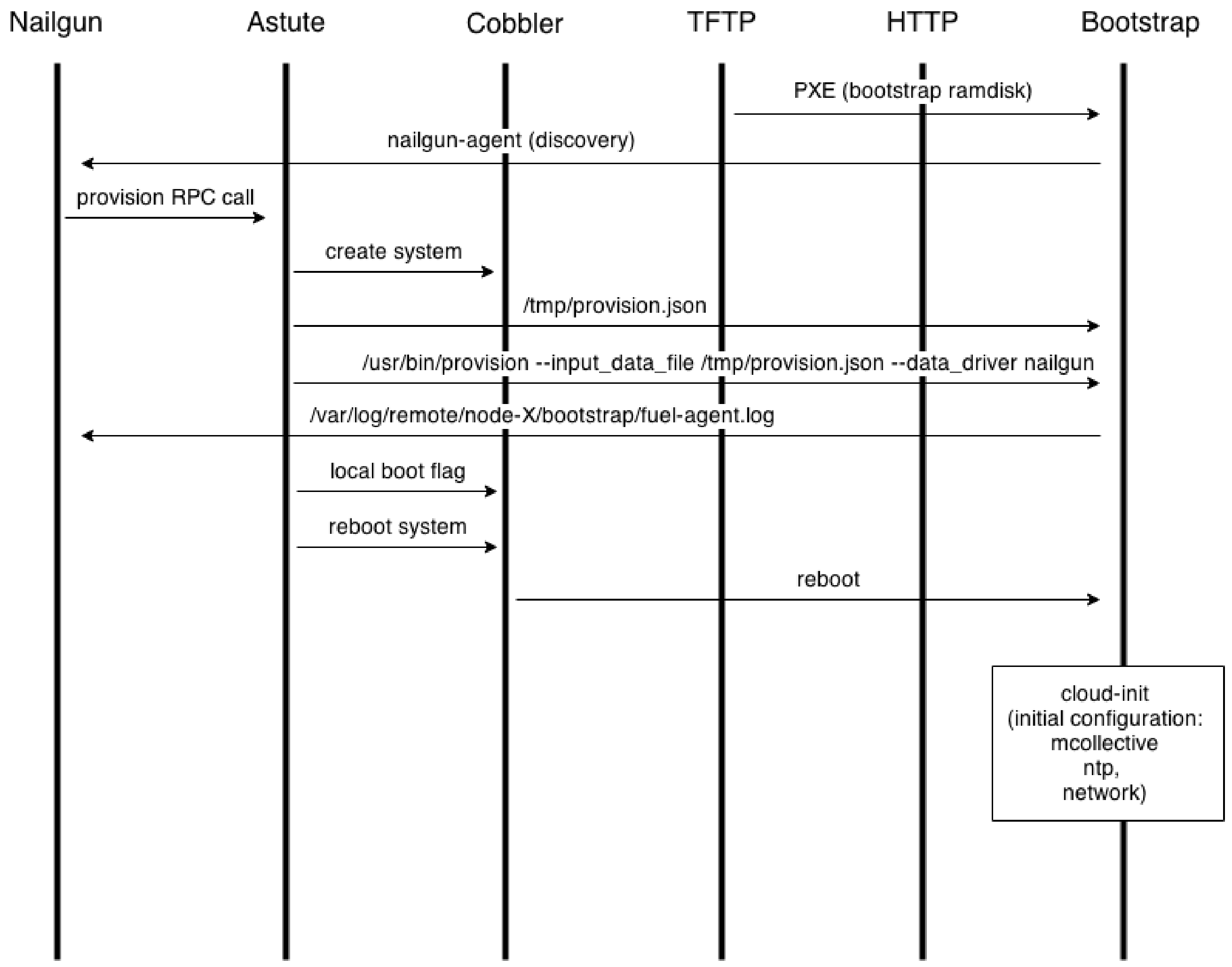
Operating system provisioning

The Fuel Agent is installed into a bootstrap ramdisk. An operating system can easily be installed on a node if the node has been booted with this ramdisk. We can simply run the /usr/bin/provision executable with the required input data to start provisioning. This allows provisioning to occur without a reboot.

The input data need to contain at least the following information:

- Partitioning scheme for the node. This scheme needs to contain information about the necessary partitions and on which disks we need to create these partitions, information about the necessary LVM groups and volumes, about software raid devices. This scheme contains also information about on which disk a bootloader needs to be installed and about the necessary file systems and their mount points. On some block devices we are assumed to put operating system images (one image per file system), while on other block devices we need to create file systems using the mkfs command.
- Operating system images URIs. Fuel Agent needs to know where to download the images and which protocol to use for this (by default, HTTP is used).
- Data for initial node configuration. Currently, we use cloud-init for the initial configuration and Fuel Agent prepares the cloud-init config drive which is put on a small partition at the end of the first hard drive. Config drive is created using jinja2 templates which are to be filled with values given from the input data. After the first reboot, cloud-init is run by upstart or similar. It then finds this config drive and configures services like NTP, MCollective, etc. It also performs an initial network configuration to make it possible for Fuel to access this particular node via SSH or MCollective and run Puppet to perform the final deployment.

The sequence diagram is below:



Viewing the control files on the Fuel Master node

To view the contents of the bootstrap ramdisk, run the following commands on the Fuel Master node:

```
cd /var/www/nailgun/bootstrap
mkdir initramfs
cd initramfs
gunzip -c ../initramfs.img | cpio -idv
```

You are now in the root file system of the ramdisk and can view the files that are included in the bootstrap node. For example:

```
cat /etc/fuel-agent/fuel-agent.conf
```

Troubleshooting image-based provisioning

The following files provide information for analyzing problems with the Fuel Agent provisioning.

- Bootstrap
  - etc/fuel-agent/fuel-agent.conf -- main configuration file for the Fuel Agent, defines the location of the provision data file, data format and log output, whether debugging is on or off, and so forth.
  - tmp/provision.json -- Astute puts this file on a node (on the in-memory file system) just before running the provision script.
  - usr/bin/provision -- executable entry point for provisioning. Astute runs this; it can also be run manually.
- Master
  - var/log/remote/node-N.domain.tld/bootstrap/fuel-agent.log -- this is where Fuel Agent log messages are recorded when the provision script is run; <N> is the node ID of the provisioned node.

Fuel Repository Mirroring

Starting in Mirantis OpenStack 6.1, the location of repositories now extends beyond just being local to the Fuel Master. It is now assumed that a given user will have Internet access and can download content from Mirantis and upstream mirrors. This impacts users with limited Internet access or unreliable connections.

Internet-based mirrors can be broken down into three categories:

- Ubuntu
- MOS DEBs
- MOS RPMs

There are two command-line utilities, fuel-createmirror and fuel-package-updates, which can replicate the mirrors.

Use fuel-createmirror for Ubuntu and MOS DEBs packages.

Use fuel-package-updates for MOS RPMs packages.

fuel-createmirror is a utility that can be used as a backend to replicate part or all of an APT repository. It can replicate Ubuntu and MOS DEBs repositories. It uses rsync as a backend.

fuel-package-updates is a utility written in Python that can pull entire APT and YUM repositories via recursive wget or rsync. Additionally, it can update Fuel environment configurations to use a given set of configuration.

Issue the following command to check the fuel-package-updates options:

```
fuel-package-updates -h
```

Note

If you change the default password (admin) in Fuel web UI, you will need to run the utility with the --password switch, or it will fail.

Seealso

Documentation on MOS RPMs mirror structure.

Corosync Settings

Corosync uses Totem protocol, which is an implementation of Virtual Synchrony protocol. It uses it in order to provide connectivity between cluster nodes, decide if cluster is quorate to provide services, to provide data layer for services that want to use features of Virtual Synchrony.

Corosync functions in Fuel as the communication and quorum service via Pacemaker cluster resource manager (crm). It's main configuration file is located in /etc/corosync/corosync.conf.

The main Corosync section is the totem section which describes how cluster nodes should communicate:

```
totem {
  version:                             2
  token:                              3000
  token_retransmits_before_loss_const: 10
  join:                              60
  consensus:                            3600
  vsftype:                             none
  max_messages:                         20
  clear_node_high_bit:                  yes
  rrp_mode:                            none
  secauth:                             off
  threads:                             0
  interface {
    ringnumber:  0
    bindnetaddr: 10.107.0.8
    mcastaddr:   239.1.1.2
    mcastport:   5405
  }
}
```

Corosync usually uses multicast UDP transport and sets up a "redundant ring" for communication. Currently Fuel deploys controllers with one redundant ring. Each ring has it’s own multicast address and bind net address that specifies on which interface Corosync should join corresponding multicast group. Fuel uses default Corosync configuration, which can also be altered in Fuel manifests.

Seealso

man corosync.conf or Corosync documentation at http://clusterlabs.org/doc/ if you want to know how to tune installation completely

Pacemaker Settings

Pacemaker is the cluster resource manager used by Fuel to manage Neutron resources, HAProxy, virtual IP addresses and MySQL Galera cluster. It is done by use of Open Cluster Framework (see http://linux-ha.org/wiki/OCF_Resource_Agents) agent scripts which are deployed in order to start/stop/monitor Neutron services, to manage HAProxy, virtual IP addresses and MySQL replication. These are located at /usr/lib/ocf/resource.d/mirantis/ocf-neutron-[metadata|ovs|dhcp|l3]-agent, /usr/lib/ocf/resource.d/fuel/mysql, /usr/lib/ocf/resource.d/ocf/haproxy. Firstly, MySQL agent is started, HAproxy and virtual IP addresses are set up. Open vSwitch, metadata, L3, and DHCP agents are started as Pacemaker clones on all the nodes.

Seealso

Using Rules to Determine Resource Location

MySQL HA script primarily targets to the cluster rebuild after power failure or equal type of disaster - it needs working Corosync in which it forms quorum of an epochs of replication and then electing master from node with newest epoch.

Be aware of default five minute interval in which every cluster member should be booted to participate in such election. Every node is a self-aware, that means if nobody pushes higher epoch that it retrieved from Corosync (neither no one did), it will just elect itself as a master.

How Fuel Deploys HA

Fuel installs Corosync service, configures corosync.conf, and includes the Pacemaker service plugin into /etc/corosync/service.d. Then Corosync service starts and spawns corresponding Pacemaker processes. Fuel configures the cluster properties of Pacemaker and then injects resource configurations for virtual IPs, HAProxy, MySQL and Neutron agent resources.

The running configuration can be retrieved from an OpenStack controller node by running:

```
# crm configure show
```

HA deployment for Networking

Fuel leverages Pacemaker resource agents in order to deploy highly avaiable networking for OpenStack environments.

Virtual IP addresses deployment details

Starting from the Fuel 5.0 release, HAProxy service and network interfaces running virtual IP addresses reside in separate haproxy network namespace. Using a separate namespace forces Linux kernel to treat connections from OpenStack services to HAProxy as remote ones, this ensures reliable failover of established connections when the management IP address migrates to another node. In order to achieve this, resource agent scripts for ocf:fuel:ns_haproxy and ocf:fuel:ns_IPaddr2 were hardened with network namespaces support.

Successfull failover of public VIP address requires controller nodes to perform active checking of the public gateway. Fuel configures the Pacemaker resource clone_ping_vip__public that makes public VIP to migrate in case the controller can't ping its public gateway.

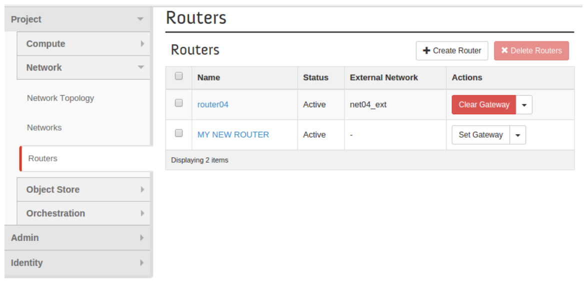
TCP keepalive configuration details

Failover sometimes ends up with dead connections. The detection of such connections requires additional assistance from the Linux kernel. To speed up the detection process from the default of two hours to a more acceptable 3 minutes, Fuel adjusts kernel parameters for net.ipv4.tcp_keepalive_time, net.ipv4.tcp_keepalive_intvl, net.ipv4.tcp_keepalive_probes and net.ipv4.tcp_retries2.

Router

Your network must have an IP address in the Public network set on a router port as an "External Gateway". Without this, your VMs are unable to access the outside world. For the purpose of example, that IP is set to 12.0.0.1 in VLAN 101.

If you add a new router, be sure to set its gateway IP:

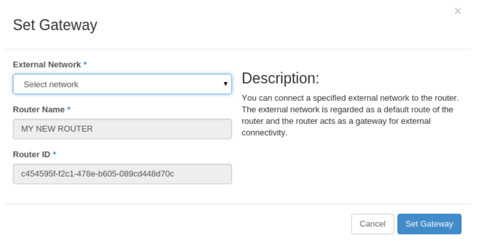


Routers

Routers

	Name	Status	External Network	Actions
	router04	Active	net04_ext	Clear Gateway
	MY NEW ROUTER	Active	-	Set Gateway

Displaying 2 items



Set Gateway

External Network *: Select network

Router Name *: MY NEW ROUTER

Router ID *: c454595f-f2c1-478e-b605-089cd448d70c

Description:

You can connect a specified external network to the router. The external network is regarded as a default route of the router and the router acts as a gateway for external connectivity.

The Fuel UI includes a field on the networking tab for the gateway address. When OpenStack deployment starts, the network on each node is reconfigured to use this gateway IP address as the default gateway.

If Floating addresses are from another L3 network, then you must configure the IP address (or multiple IPs if Floating addresses are from more than one L3 network) for them on the router as well. Otherwise, Floating IPs on nodes will be inaccessible.

Consider the following routing recommendations when you configure your network:

- Use the default routing via a router in the Public network
- Use the the management network to access your management infrastructure (L3 connectivity if necessary)
- The Storage and VM networks should be configured without access to other networks (no L3 connectivity)

Switches

You must manually configure your switches before deploying your OpenStack environment. Unfortunately the set of configuration steps, and even the terminology used, is different for different vendors; this section provides some vendor-agnostic information about how traffic should flow.

To configure your switches:

- Configure all access ports to allow non-tagged PXE booting connections from each slave node to the Fuel Master node. This network is referred to as the Fuel network.
- By default, the Fuel Master node uses the eth0 interface to serve PXE requests. However, you can modify this setting during the installation of the Fuel Master node.
- If you use the eth0 interface for PXE requests, you must set the switch port for eth0 on the Fuel Master node to access mode.
- We recommend that you use the eth0 interfaces of all other nodes for PXE booting as well. Corresponding ports must also be in access mode.
- Taking into account that this is the network for PXE booting, do not mix this L2 segment with any other network segments. Fuel runs a DHCP server, and, if there is another DHCP on the same L2 network segment, both the company's infrastructure and Fuel's are unable to function properly.
- You must also configure each of the switch's ports connected to nodes as an "STP Edge port" (or a "spanning-tree port fast trunk", according to Cisco terminology). If you do not do that, DHCP timeout issues may occur.

As soon as the Fuel network is configured, Fuel can operate. Other networks are required for OpenStack environments, and currently all of these networks live in VLANs over the one or multiple physical interfaces on a node. This means that the switch should pass tagged traffic, and untagging is done on the Linux hosts.

Note

For the sake of simplicity, all the VLANs specified on the networks tab of the Fuel UI should be configured on switch ports, pointing to Slave nodes, as tagged.

Of course, it is possible to specify as tagged only certain ports for certain nodes. However, in the current version, all existing networks are automatically allocated for each node, with any role. And network check also checks if tagged traffic can pass, even if some nodes do not require this check. for example, Cinder nodes do not need fixed network traffic)

This is enough to deploy the OpenStack environment. However, from a practical standpoint, it is still not really usable because there is no connection to other corporate networks yet. To make that possible, you must configure uplink port(s).

One of the VLANs may carry the office network. To provide access to the Fuel Master node from your network, any other free physical network interface on the Fuel Master node can be used and configured according to your network rules (static IP or DHCP). The same network segment can be used for Public and Floating ranges. In this case, you must provide the corresponding VLAN ID and IP ranges in the UI. One Public IP per node is used to SNAT traffic out of the VMs network, and one or more floating addresses per VM instance are used to get access to the VM from your network, or even the global Internet. To have a VM visible from the Internet is similar to having it visible from corporate network - corresponding IP ranges and VLAN IDs must be specified for the Floating and Public networks. One current limitation of Fuel is that the user must use the same L2 segment for both Public and Floating networks.

Example configuration for one of the ports on a Cisco switch:

```
interface GigabitEthernet0/6                # switch port
description s0_eth0 jv                      # description
switchport trunk encapsulation dot1q        # enables VLANs
switchport trunk native vlan 262            # access port, untags VLAN 262
switchport trunk allowed vlan 100,102,104   # 100,102,104 VLANs are passed with tags
switchport mode trunk                       # To allow more than 1 VLAN on the port
spanning-tree portfast trunk                # STP Edge port to skip network loop
                                            # checks (to prevent DHCP timeout issues)
vlan 262,100,102,104                        # Might be needed for enabling VLANs
```