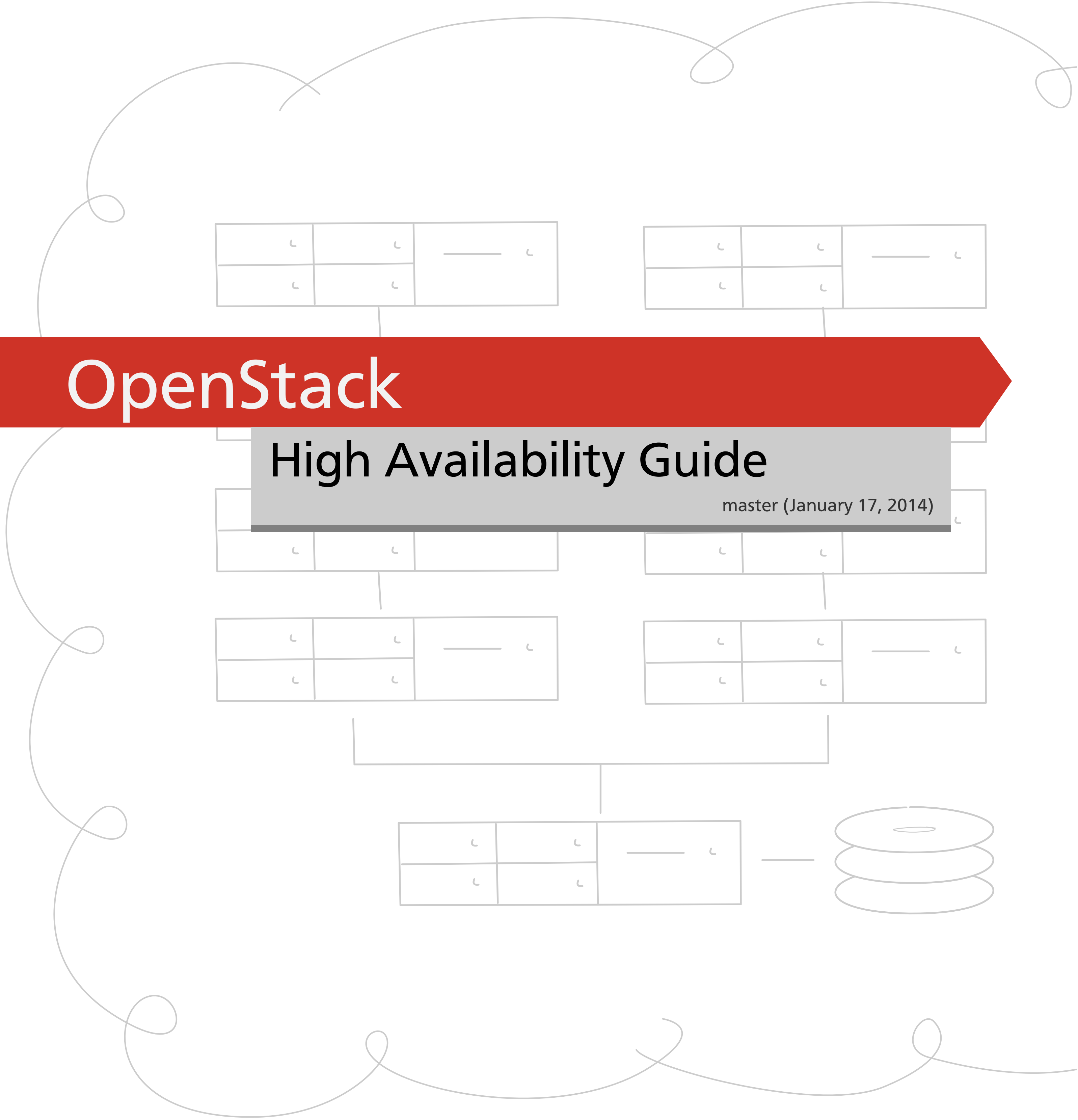
OpenStack

High Availability Guide

master (January 17, 2014)



docs.openstack.org

OpenStack High Availability Guide

Florian Haas

master (2014-01-17)

Table of Contents

1. Introduction to OpenStack High Availability

Table of Contents

High Availability systems seek to minimize two things:

- **System downtime** — occurs when a *user-facing* service is unavailable beyond a specified maximum amount of time, and
- **Data loss** — accidental deletion or destruction of data.

Most high availability systems guarantee protection against system downtime and data loss only in the event of a single failure. However, they are also expected to protect against cascading failures, where a single failure deteriorates into a series of consequential failures.

A crucial aspect of high availability is the elimination of single points of failure (SPOFs). A SPOF is an individual piece of equipment or software which will cause system downtime or data loss if it fails. In order to eliminate SPOFs, check that mechanisms exist for redundancy of:

- Network components, such as switches and routers
- Applications and automatic service migration
- Storage components
- Facility services such as power, air conditioning, and fire protection

Most high availability systems will fail in the event of multiple independent (non-consequential) failures. In this case, most systems will protect data over maintaining availability.

High-availability systems typically achieve uptime of 99.99% or more, which roughly equates to less than an hour of cumulative downtime per year. In order to achieve this, high availability systems should keep recovery times after a failure to about one to two minutes, sometimes significantly less.

OpenStack currently meets such availability requirements for its own infrastructure services, meaning that an uptime of 99.99% is feasible for the OpenStack infrastructure proper. However, OpenStack *does not* guarantee 99.99% availability for individual guest instances.

Preventing single points of failure can depend on whether or not a service is stateless.

Stateless vs. Stateful services

A stateless service is one that provides a response after your request, and then requires no further attention. To make a stateless service highly available, you need to provide redundant instances and load balance them. OpenStack services that are stateless include nova-api, nova-conductor, glance-api, keystone-api, neutron-api and nova-scheduler.

A stateful service is one where subsequent requests to the service depend on the results of the first request. Stateful services are more difficult to manage because a single action typically involves more than one request, so simply providing additional instances and load balancing will not solve the problem. For example, if the Horizon user interface reset itself every time you went to a new page, it wouldn't be very useful. OpenStack services that are stateful include the OpenStack database and message queue.

Making stateful services highly available can depend on whether you choose an active/passive or active/active configuration.

Active/Passive

In an active/passive configuration, systems are set up to bring additional resources online to replace those that have failed. For example, OpenStack would write to the main database while maintaining a disaster recovery database that can be brought online in the event that the main database fails.

Typically, an active/passive installation for a stateless service would maintain a redundant instance that can be brought online when required. Requests are load balanced using a virtual IP address and a load balancer such as HAProxy.

A typical active/passive installation for a stateful service maintains a replacement resource that can be brought online when required. A separate application (such as Pacemaker or Corosync) monitors these services, bringing the backup online as necessary.

Active/Active

In an active/active configuration, systems also use a backup but will manage both the main and redundant systems concurrently. This way, if there is a failure the user is unlikely to notice. The backup system is already online, and takes on increased load while the main system is fixed and brought back online.

Typically, an active/active installation for a stateless service would maintain a redundant instance, and requests are load balanced using a virtual IP address and a load balancer such as HAProxy.

A typical active/active installation for a stateful service would include redundant services with all instances having an identical state. For example, updates to one instance of a database would also update all other instances. This way a request to one instance is the same as a request to any other. A load balancer manages the traffic to these systems, ensuring that operational systems always handle the request.

These are some of the more common ways to implement these high availability architectures, but they are by no means the only ways to do it. The important thing is to make sure that your services are redundant, and available; how you achieve that is up to you. This document will cover some of the more common options for highly available systems.

2. HA Using Active/Passive

Table of Contents

The Pacemaker Cluster Stack

OpenStack infrastructure high availability relies on the Pacemaker cluster stack, the state-of-the-art high availability and load balancing stack for the Linux platform. Pacemaker is storage- and application-agnostic, and is in no way specific to OpenStack.

Pacemaker relies on the Corosync messaging layer for reliable cluster communications. Corosync implements the Totem single-ring ordering and membership protocol and provides UDP and InfiniBand based messaging, quorum, and cluster membership to Pacemaker.

Pacemaker interacts with applications through *resource agents* (RAs), of which it supports over 70 natively. Pacemaker can also easily use third-party RAs. An OpenStack high-availability configuration uses existing native Pacemaker RAs (such as those managing MySQL databases or virtual IP addresses), existing third-party RAs (such as for RabbitMQ), and native OpenStack RAs (such as those managing the OpenStack Identity and Image Services).

Installing Packages

On any host that is meant to be part of a Pacemaker cluster, you must first establish cluster communications through the Corosync messaging layer. This involves installing the following packages (and their dependencies, which your package manager will normally install automatically):

- `pacemaker`
- `corosync`
- `cluster-glue`
- `fence-agents` (Fedora only; all other distributions use fencing agents from `cluster-glue`)
- `resource-agents`

Setting up Corosync

Besides installing the `corosync` package, you will also have to create a configuration file, stored in `/etc/corosync/corosync.conf`. Most distributions ship an example configuration file (`corosync.conf.example`) as part of the documentation bundled with the `corosync` package. An example Corosync configuration file is shown below:

Corosync configuration file (`corosync.conf`).

```
totem {
        version: 2

        # Time (in ms) to wait for a token ❶
        token: 10000

        # How many token retransmits before forming a new
        # configuration
        token_retransmits_before_loss_const: 10

        # Turn off the virtual synchrony filter
        vsftype: none

        # Enable encryption ❷
        secauth: on

        # How many threads to use for encryption/decryption
        threads: 0

        # This specifies the redundant ring protocol, which may be
        # none, active, or passive. ❸
        rrp_mode: active

        # The following is a two-ring multicast configuration. ❹
        interface {
                ringnumber: 0
                bindnetaddr: 192.168.42.0
                mcastaddr: 239.255.42.1
                mcastport: 5405
        }
        interface {
                ringnumber: 1
                bindnetaddr: 10.0.42.0
                mcastaddr: 239.255.42.2
                mcastport: 5405
        }
}

amf {
        mode: disabled
}

service {
        # Load the Pacemaker Cluster Resource Manager ❺
        ver:       1
        name:      pacemaker
}

aisexec {
        user:   root
        group:  root
}

logging {
        fileline: off
        to_stderr: yes
        to_logfile: no
        to_syslog: yes
        syslog_facility: daemon
        debug: off
        timestamp: on
        logger_subsys {
                subsys: AMF
                debug: off
                tags: enter|leave|trace1|trace2|trace3|trace4|trace6
        }
}
```

❶ The `token` value specifies the time, in milliseconds, during which the Corosync token is expected to be transmitted around the ring. When this timeout expires, the token is declared lost, and after `token_retransmits_before_loss_const` lost tokens the non-responding *processor* (cluster node) is declared dead. In other words, `token` × `token_retransmits_before_loss_const` is the maximum time a node is allowed to not respond to cluster messages before being considered dead. The default for `token` is 1000 (1 second), with 4 allowed retransmits. These defaults are intended to minimize failover times, but can cause frequent "false alarms" and unintended failovers in case of short network interruptions. The values used here are safer, albeit with slightly extended failover times.

❷ With `secauth` enabled, Corosync nodes mutually authenticate using a 128-byte shared secret stored in `/etc/corosync/authkey`, which may be generated with the `corosync-keygen` utility. When using `secauth`, cluster communications are also encrypted.

❸ In Corosync configurations using redundant networking (with more than one `interface`), you must select a Redundant Ring Protocol (RRP) mode other than `none`. `active` is the recommended RRP mode.

❹ There are several things to note about the recommended interface configuration:

- The ringnumber must differ between all configured interfaces, starting with 0.
- The bindnetaddr is the *network* address of the interfaces to bind to. The example uses two network addresses of /24 IPv4 subnets.
- Multicast groups (mcastaddr) *must not* be reused across cluster boundaries. In other words, no two distinct clusters should ever use the same multicast group. Be sure to select multicast addresses compliant with RFC 2365, "Administratively Scoped IP Multicast".
- For firewall configurations, note that Corosync communicates over UDP only, and uses mcastport (for receives) and mcastport-1 (for sends).

❺ The service declaration for the pacemaker service may be placed in the corosync.conf file directly, or in its own separate file, /etc/corosync/service.d/pacemaker.

Once created, the corosync.conf file (and the authkey file if the secauth option is enabled) must be synchronized across all cluster nodes.

Starting Corosync

Corosync is started as a regular system service. Depending on your distribution, it may ship with a LSB (System V style) init script, an upstart job, or a systemd unit file. Either way, the service is usually named corosync:

- /etc/init.d/corosync start (LSB)
- service corosync start (LSB, alternate)
- start corosync (upstart)
- systemctl start corosync (systemd)

You can now check Corosync connectivity with two tools.

The corosync-cfgtool utility, when invoked with the -s option, gives a summary of the health of the communication rings:

```
# corosync-cfgtool -s
Printing ring status.
Local node ID 435324542
RING ID 0
        id      = 192.168.42.82
        status  = ring 0 active with no faults
RING ID 1
        id      = 10.0.42.100
        status  = ring 1 active with no faults
```

The corosync-objctl utility can be used to dump the Corosync cluster member list:

```
# corosync-objctl runtime.totem.pg.mrp.srp.members
runtime.totem.pg.mrp.srp.435324542.ip=r(0) ip(192.168.42.82) r(1) ip(10.0.42.
100)
runtime.totem.pg.mrp.srp.435324542.join_count=1
runtime.totem.pg.mrp.srp.435324542.status=joined
runtime.totem.pg.mrp.srp.983895584.ip=r(0) ip(192.168.42.87) r(1) ip(10.0.42.
254)
runtime.totem.pg.mrp.srp.983895584.join_count=1
runtime.totem.pg.mrp.srp.983895584.status=joined
```

You should see a `status=joined` entry for each of your constituent cluster nodes.

Starting Pacemaker

Once the Corosync services have been started, and you have established that the cluster is communicating properly, it is safe to start `pacemakerd`, the Pacemaker master control process:

- `/etc/init.d/pacemaker start` (LSB)
- `service pacemaker start` (LSB, alternate)
- `start pacemaker` (upstart)
- `systemctl start pacemaker` (systemd)

Once Pacemaker services have started, Pacemaker will create an default empty cluster configuration with no resources. You may observe Pacemaker’s status with the `crm_mon` utility:

```
============
Last updated: Sun Oct  7 21:07:52 2012
Last change: Sun Oct  7 20:46:00 2012 via cibadmin on node2
Stack: openais
Current DC: node2 - partition with quorum
Version: 1.1.6-9971ebba4494012a93c03b40a2c58ec0eb60f50c
2 Nodes configured, 2 expected votes
0 Resources configured.
============

Online: [ node2 node1 ]
```

Setting basic cluster properties

Once your Pacemaker cluster is set up, it is recommended to set a few basic cluster properties. To do so, start the `crm` shell and change into the configuration menu by entering `configure`. Alternatively. you may jump straight into the Pacemaker configuration menu by typing `crm configure` directly from a shell prompt.

Then, set the following properties:

```
property no-quorum-policy="ignore" \ # ❶
  pe-warn-series-max="1000" \        # ❷
  pe-input-series-max="1000" \
  pe-error-series-max="1000" \
  cluster-recheck-interval="5min"    # ❸
```

❶ Setting `no-quorum-policy="ignore"` is required in 2-node Pacemaker clusters for the following reason: if quorum enforcement is enabled, and one of the two nodes fails, then the remaining node can not establish a *majority* of quorum votes necessary to run services, and thus it is unable to take over any resources. The appropriate workaround is to ignore loss of quorum in the cluster. This is safe and necessary *only* in 2-node clusters. Do not set this property in Pacemaker clusters with more than two nodes.

❷ Setting `pe-warn-series-max`, `pe-input-series-max` and `pe-error-series-max` to 1000 instructs Pacemaker to keep a longer history of the inputs processed, and errors and warnings generated, by its Policy Engine. This history is typically useful in case cluster troubleshooting becomes necessary.

❸ Pacemaker uses an event-driven approach to cluster state processing. However, certain Pacemaker actions occur at a configurable interval, `cluster-recheck-interval`, which defaults to 15 minutes. It is usually prudent to reduce this to a shorter interval, such as 5 or 3 minutes.

Once you have made these changes, you may `commit` the updated configuration.

Cloud Controller Cluster Stack

The Cloud Controller sits on the management network and needs to talk to all other services.

Highly available MySQL

MySQL is the default database server used by many OpenStack services. Making the MySQL service highly available involves

- configuring a DRBD device for use by MySQL,
- configuring MySQL to use a data directory residing on that DRBD device,
- selecting and assigning a virtual IP address (VIP) that can freely float between cluster nodes,
- configuring MySQL to listen on that IP address,
- managing all resources, including the MySQL daemon itself, with the Pacemaker cluster manager.

Note

MySQL/Galera is an alternative method of configuring MySQL for high availability. It is likely to become the preferred method of achieving MySQL high availability once it has sufficiently matured. At the time of writing, however, the Pacemaker/DRBD based approach remains the recommended one for OpenStack environments.

Configuring DRBD

The Pacemaker based MySQL server requires a DRBD resource from which it mounts the `/var/lib/mysql` directory. In this example, the DRBD resource is simply named `mysql`:

`mysql` DRBD resource configuration (`/etc/drbd.d/mysql.res`).

```
resource mysql {
  device    minor 0;
  disk      "/dev/data/mysql";
  meta-disk internal;
  on node1 {
    address ipv4 10.0.42.100:7700;
  }
  on node2 {
    address ipv4 10.0.42.254:7700;
  }
}
```

This resource uses an underlying local disk (in DRBD terminology, a *backing device*) named `/dev/data/mysql` on both cluster nodes, `node1` and `node2`. Normally, this would be an LVM Logical Volume specifically set aside for this purpose. The DRBD `meta-disk` is `internal`, meaning DRBD-specific metadata is being stored at the end of the `disk` device itself. The device is configured to communicate between IPv4 addresses 10.0.42.100 and 10.0.42.254, using TCP port 7700. Once enabled, it will map to a local DRBD block device with the device minor number 0, that is, `/dev/drbd0`.

Enabling a DRBD resource is explained in detail in the DRBD User's Guide. In brief, the proper sequence of commands is this:

```
drbdadm create-md mysql ❶
drbdadm up mysql ❷
drbdadm -- --force primary mysql ❸
```

❶ Initializes DRBD metadata and writes the initial set of metadata to `/dev/data/mysql`. Must be completed on both nodes.

❷ Creates the `/dev/drbd0` device node, *attaches* the DRBD device to its backing store, and *connects* the DRBD node to its peer. Must be completed on both nodes.

❸ Kicks off the initial device synchronization, and puts the device into the `primary` (readable and writable) role. See Resource roles (from the DRBD User's Guide) for a more detailed description of the primary and secondary roles in DRBD. Must be completed *on one node only,* namely the one where you are about to continue with creating your filesystem.

Creating a file system

Once the DRBD resource is running and in the primary role (and potentially still in the process of running the initial device synchronization), you may proceed with creating the filesystem for MySQL data. XFS is the generally recommended filesystem:

```
mkfs -t xfs /dev/drbd1
```

You may also use the alternate device path for the DRBD device, which may be easier to remember as it includes the self-explanatory resource name:

```
mkfs -t xfs /dev/drbd/by-res/mysql
```

Once completed, you may safely return the device to the secondary role. Any ongoing device synchronization will continue in the background:

```
drbdadm secondary mysql
```

Preparing MySQL for Pacemaker high availability

In order for Pacemaker monitoring to function properly, you must ensure that MySQL's database files reside on the DRBD device. If you already have an existing MySQL database, the simplest approach is to just move the contents of the existing /var/lib/mysql directory into the newly created filesystem on the DRBD device.

Warning

You must complete the next step while the MySQL database server is shut down.

```
node1:# mount /dev/drbd/by-res/mysql /mnt
node1:# mv /var/lib/mysql/* /mnt
node1:# umount /mnt
```

For a new MySQL installation with no existing data, you may also run the mysql_install_db command:

```
node1:# mount /dev/drbd/by-res/mysql /mnt
node1:# mysql_install_db --datadir=/mnt
node1:# umount /mnt
```

Regardless of the approach, the steps outlined here must be completed on only one cluster node.

Adding MySQL resources to Pacemaker

You may now proceed with adding the Pacemaker configuration for MySQL resources. Connect to the Pacemaker cluster with crm configure, and add the following cluster resources:

```
primitive p_ip_mysql ocf:heartbeat:IPaddr2 \
  params ip="192.168.42.101" cidr_netmask="24" \
  op monitor interval="30s"
primitive p_drbd_mysql ocf:linbit:drbd \
  params drbd_resource="mysql" \
  op start timeout="90s" \
  op stop timeout="180s" \
  op promote timeout="180s" \
  op demote timeout="180s" \
  op monitor interval="30s" role="Slave" \
  op monitor interval="29s" role="Master"
primitive p_fs_mysql ocf:heartbeat:Filesystem \
  params device="/dev/drbd/by-res/mysql" \
    directory="/var/lib/mysql" \
    fstype="xfs" \
    options="relatime" \
  op start timeout="60s" \
  op stop timeout="180s" \
  op monitor interval="60s" timeout="60s"
primitive p_mysql ocf:heartbeat:mysql \
  params additional_parameters="--bind-address=50.56.179.138"
    config="/etc/mysql/my.cnf" \
    pid="/var/run/mysqld/mysqld.pid" \
    socket="/var/run/mysqld/mysqld.sock" \
    log="/var/log/mysql/mysqld.log" \
  op monitor interval="20s" timeout="10s" \
  op start timeout="120s" \
  op stop timeout="120s"
group g_mysql p_ip_mysql p_fs_mysql p_mysql
ms ms_drbd_mysql p_drbd_mysql \
  meta notify="true" clone-max="2"
colocation c_mysql_on_drbd inf: g_mysql ms_drbd_mysql:Master
order o_drbd_before_mysql inf: ms_drbd_mysql:promote g_mysql:start
```

This configuration creates

- `p_ip_mysql`, a virtual IP address for use by MySQL (192.168.42.101),
- `p_fs_mysql`, a Pacemaker managed filesystem mounted to `/var/lib/mysql` on whatever node currently runs the MySQL service,
- `ms_drbd_mysql`, the *master/slave set* managing the `mysql` DRBD resource,
- a service `group` and `order` and `colocation` constraints to ensure resources are started on the correct nodes, and in the correct sequence.

`crm configure` supports batch input, so you may copy and paste the above into your live pacemaker configuration, and then make changes as required. For example, you may enter `edit p_ip_mysql` from the `crm configure` menu and edit the resource to match your preferred virtual IP address.

Once completed, commit your configuration changes by entering `commit` from the `crm configure` menu. Pacemaker will then start the MySQL service, and its dependent resources, on one of your nodes.

Configuring OpenStack services for highly available MySQL

Your OpenStack services must now point their MySQL configuration to the highly available, virtual cluster IP address — rather than a MySQL server's physical IP address as you normally would.

For OpenStack Image, for example, if your MySQL service IP address is 192.168.42.101 as in the configuration explained here, you would use the following line in your OpenStack Image registry configuration file (`glance-registry.conf`):

```
sql_connection = mysql://glancedbadmin:<password>@192.168.42.101/glance
```

No other changes are necessary to your OpenStack configuration. If the node currently hosting your database experiences a problem necessitating service failover, your OpenStack services may experience a brief MySQL interruption, as they would in the event of a network hiccup, and then continue to run normally.

Highly available RabbitMQ

RabbitMQ is the default AMQP server used by many OpenStack services. Making the RabbitMQ service highly available involves

- configuring a DRBD device for use by RabbitMQ,
- configuring RabbitMQ to use a data directory residing on that DRBD device,
- selecting and assigning a virtual IP address (VIP) that can freely float between cluster nodes,
- configuring RabbitMQ to listen on that IP address,

- managing all resources, including the RabbitMQ daemon itself, with the Pacemaker cluster manager.

Note

There is an alternative method of configuring RabbitMQ for high availability. That approach, known as active-active mirrored queues, happens to be the one preferred by the RabbitMQ developers — however it has shown less than ideal consistency and reliability in OpenStack clusters. Thus, at the time of writing, the Pacemaker/DRBD based approach remains the recommended one for OpenStack environments, although this may change in the near future as RabbitMQ active-active mirrored queues mature.

Configuring DRBD

The Pacemaker based RabbitMQ server requires a DRBD resource from which it mounts the `/var/lib/rabbitmq` directory. In this example, the DRBD resource is simply named `rabbitmq`:

`rabbitmq` DRBD resource configuration (`/etc/drbd.d/rabbitmq.res`).

```
resource rabbitmq {
  device    minor 1;
  disk      "/dev/data/rabbitmq";
  meta-disk internal;
  on node1 {
    address ipv4 10.0.42.100:7701;
  }
  on node2 {
    address ipv4 10.0.42.254:7701;
  }
}
```

This resource uses an underlying local disk (in DRBD terminology, a *backing device*) named `/dev/data/rabbitmq` on both cluster nodes, `node1` and `node2`. Normally, this would be an LVM Logical Volume specifically set aside for this purpose. The DRBD `meta-disk` is `internal`, meaning DRBD-specific metadata is being stored at the end of the `disk` device itself. The device is configured to communicate between IPv4 addresses 10.0.42.100 and 10.0.42.254, using TCP port 7701. Once enabled, it will map to a local DRBD block device with the device minor number 1, that is, `/dev/drbd1`.

Enabling a DRBD resource is explained in detail in the DRBD User's Guide. In brief, the proper sequence of commands is this:

```
drbdadm create-md rabbitmq ❶
drbdadm up rabbitmq ❷
drbdadm -- --force primary rabbitmq ❸
```

❶ Initializes DRBD metadata and writes the initial set of metadata to `/dev/data/rabbitmq`. Must be completed on both nodes.

❷ Creates the `/dev/drbd1` device node, *attaches* the DRBD device to its backing store, and *connects* the DRBD node to its peer. Must be completed on both nodes.

❸ Kicks off the initial device synchronization, and puts the device into the `primary` (readable and writable) role. See Resource roles (from the DRBD User's Guide) for a more detailed description of the primary and secondary roles in DRBD. Must be completed *on one node only,* namely the one where you are about to continue with creating your filesystem.

Creating a file system

Once the DRBD resource is running and in the primary role (and potentially still in the process of running the initial device synchronization), you may proceed with creating the filesystem for RabbitMQ data. XFS is the generally recommended filesystem:

```
mkfs -t xfs /dev/drbd1
```

You may also use the alternate device path for the DRBD device, which may be easier to remember as it includes the self-explanatory resource name:

```
mkfs -t xfs /dev/drbd/by-res/rabbitmq
```

Once completed, you may safely return the device to the secondary role. Any ongoing device synchronization will continue in the background:

```
drbdadm secondary rabbitmq
```

Preparing RabbitMQ for Pacemaker high availability

In order for Pacemaker monitoring to function properly, you must ensure that RabbitMQ's `.erlang.cookie` files are identical on all nodes, regardless of whether DRBD is mounted there or not. The simplest way of doing so is to take an existing `.erlang.cookie` from one of your nodes, copying it to the RabbitMQ data directory on the other node, and also copying it to the DRBD-backed filesystem.

```
node1:# scp -a /var/lib/rabbitmq/.erlang.cookie node2:/var/lib/rabbitmq/
node1:# mount /dev/drbd/by-res/rabbitmq /mnt
node1:# cp -a /var/lib/rabbitmq/.erlang.cookie /mnt
node1:# umount /mnt
```

Adding RabbitMQ resources to Pacemaker

You may now proceed with adding the Pacemaker configuration for RabbitMQ resources. Connect to the Pacemaker cluster with `crm configure`, and add the following cluster resources:

```
primitive p_ip_rabbitmp ocf:heartbeat:IPaddr2 \
  params ip="192.168.42.100" cidr_netmask="24" \
  op monitor interval="10s"
primitive p_drbd_rabbitmq ocf:linbit:drbd \
  params drbd_resource="rabbitmq" \
  op start timeout="90s" \
  op stop timeout="180s" \
  op promote timeout="180s" \
  op demote timeout="180s" \
  op monitor interval="30s" role="Slave" \
  op monitor interval="29s" role="Master"
primitive p_fs_rabbitmq ocf:heartbeat:Filesystem \
  params device="/dev/drbd/by-res/rabbitmq" \
    directory="/var/lib/rabbitmq" \
    fstype="xfs" options="relatime" \
  op start timeout="60s" \
  op stop timeout="180s" \
  op monitor interval="60s" timeout="60s"
primitive p_rabbitmq ocf:rabbitmq:rabbitmq-server \
  params nodename="rabbit@localhost" \
    mnesia_base="/var/lib/rabbitmq" \
  op monitor interval="20s" timeout="10s"
group g_rabbitmq p_ip_rabbitmq p_fs_rabbitmq p_rabbitmq
ms ms_drbd_rabbitmq p_drbd_rabbitmq \
  meta notify="true" master-max="1" clone-max="2"
colocation c_rabbitmq_on_drbd inf: g_rabbitmq ms_drbd_rabbitmq:Master
order o_drbd_before_rabbitmq inf: ms_drbd_rabbitmq:promote g_rabbitmq:start
```

This configuration creates

- `p_ip_rabbitmp`, a virtual IP address for use by RabbitMQ (192.168.42.100),
- `p_fs_rabbitmq`, a Pacemaker managed filesystem mounted to `/var/lib/rabbitmq` on whatever node currently runs the RabbitMQ service,
- `ms_drbd_rabbitmq`, the *master/slave set* managing the `rabbitmq` DRBD resource,
- a service `group` and `order` and `colocation` constraints to ensure resources are started on the correct nodes, and in the correct sequence.

`crm configure` supports batch input, so you may copy and paste the above into your live pacemaker configuration, and then make changes as required. For example, you may enter `edit p_ip_rabbitmq` from the `crm configure` menu and edit the resource to match your preferred virtual IP address.

Once completed, commit your configuration changes by entering `commit` from the `crm configure` menu. Pacemaker will then start the RabbitMQ service, and its dependent resources, on one of your nodes.

Configuring OpenStack services for highly available RabbitMQ

Your OpenStack services must now point their RabbitMQ configuration to the highly available, virtual cluster IP address — rather than a RabbitMQ server's physical IP address as you normally would.

For OpenStack Image, for example, if your RabbitMQ service IP address is 192.168.42.100 as in the configuration explained here, you would use the following line in your OpenStack Image API configuration file (`glance-api.conf`):

```
rabbit_host = 192.168.42.100
```

No other changes are necessary to your OpenStack configuration. If the node currently hosting your RabbitMQ experiences a problem necessitating service failover, your OpenStack services may experience a brief RabbitMQ interruption, as they would in the event of a network hiccup, and then continue to run normally.

API Node Cluster Stack

The API node exposes OpenStack API endpoints onto external network (Internet). It needs to talk to the Cloud Controller on the management network.

Configure the VIP

First of all, we need to select and assign a virtual IP address (VIP) that can freely float between cluster nodes.

This configuration creates `p_ip_api`, a virtual IP address for use by the API node (192.168.42.103) :

```
primitive p_api-ip ocf:heartbeat:IPaddr2 \
    params ip="192.168.42.103" cidr_netmask="24" \
    op monitor interval="30s"
```

Highly available OpenStack Identity

OpenStack Identity is the Identity Service in OpenStack and used by many services. Making the OpenStack Identity service highly available in active / passive mode involves

- configuring OpenStack Identity to listen on the VIP address,
- managing OpenStack Identity daemon with the Pacemaker cluster manager,
- configuring OpenStack services to use this IP address.

Note

Here is the documentation for installing OpenStack Identity service.

Adding OpenStack Identity resource to Pacemaker

First of all, you need to download the resource agent to your system :

```
cd /usr/lib/ocf/resource.d
mkdir openstack
cd openstack
wget https://raw.github.com/madkiss/openstack-resource-agents/master/ocf/
keystone
chmod a+rx *
```

You may now proceed with adding the Pacemaker configuration for OpenStack Identity resource. Connect to the Pacemaker cluster with `crm configure`, and add the following cluster resources:

```
primitive p_keystone ocf:openstack:keystone \
  params config="/etc/keystone/keystone.conf" os_password="secret"
 os_username="admin" os_tenant_name="admin" os_auth_url="http://192.168.42.
103:5000/v2.0/" \
  op monitor interval="30s" timeout="30s"
```

This configuration creates `p_keystone`, a resource for manage OpenStack Identity service.

`crm configure` supports batch input, so you may copy and paste the above into your live pacemaker configuration, and then make changes as required. For example, you may enter `edit p_ip_keystone` from the `crm configure` menu and edit the resource to match your preferred virtual IP address.

Once completed, commit your configuration changes by entering `commit` from the `crm configure` menu. Pacemaker will then start the OpenStack Identity service, and its dependent resources, on one of your nodes.

Configuring OpenStack Identity service

You need to edit your OpenStack Identity configuration file (`keystone.conf`) and change the bind parameter :

```
bind_host = 192.168.42.103
```

To be sure all data will be high available, you should be sure that you store everything in the MySQL database (which is also high available) :

```
[catalog]
driver = keystone.catalog.backends.sql.Catalog
...
[identity]
driver = keystone.identity.backends.sql.Identity
...
```

Configuring OpenStack Services to use High Available OpenStack Identity

Your OpenStack services must now point their OpenStack Identity configuration to the highly available, virtual cluster IP address — rather than a OpenStack Identity server's physical IP address as you normally would.

For example with OpenStack Compute, if your OpenStack Identity service IP address is 192.168.42.103 as in the configuration explained here, you would use the following line in your API configuration file (`api-paste.ini`):

```
auth_host = 192.168.42.103
```

You need also to create the OpenStack Identity Endpoint with this IP.

NOTE : If you are using both private and public IP addresses, you should create two Virtual IP addresses and define your endpoint like this :

```
keystone endpoint-create --region $KEYSTONE_REGION --service-id $service-id
 --publicurl 'http://PUBLIC_VIP:5000/v2.0' --adminurl 'http://192.168.42.
103:35357/v2.0' --internalurl 'http://192.168.42.103:5000/v2.0'
```

If you are using the Horizon Dashboard, you should edit `local_settings.py` file :

```
OPENSTACK_HOST = 192.168.42.103
```

Highly available OpenStack Image API

OpenStack Image Service offers a service for discovering, registering, and retrieving virtual machine images. Making the OpenStack Image API service highly available in active / passive mode involves

- configuring OpenStack Image to listen on the VIP address,

- managing OpenStack Image API daemon with the Pacemaker cluster manager,
- configuring OpenStack services to use this IP address.

Note

Here is the documentation for installing OpenStack Image API service.

Adding OpenStack Image API resource to Pacemaker

First of all, you need to download the resource agent to your system :

```
cd /usr/lib/ocf/resource.d/openstack
wget https://raw.github.com/madkiss/openstack-resource-agents/master/ocf/
glance-api
chmod a+rx *
```

You may now proceed with adding the Pacemaker configuration for OpenStack Image API resource. Connect to the Pacemaker cluster with `crm configure`, and add the following cluster resources:

```
primitive p_glance-api ocf:openstack:glance-api \
    params config="/etc/glance/glance-api.conf" os_password="secrete"
 os_username="admin" os_tenant_name="admin" os_auth_url="http://192.168.42.
103:5000/v2.0/" \
    op monitor interval="30s" timeout="30s"
```

This configuration creates

- `p_glance-api`, a resource for manage OpenStack Image API service

`crm configure` supports batch input, so you may copy and paste the above into your live pacemaker configuration, and then make changes as required. For example, you may enter `edit p_ip_glance-api` from the `crm configure` menu and edit the resource to match your preferred virtual IP address.

Once completed, commit your configuration changes by entering `commit` from the `crm configure` menu. Pacemaker will then start the OpenStack Image API service, and its dependent resources, on one of your nodes.

Configuring OpenStack Image API service

Edit `/etc/glance/glance-api.conf` :

```
# We have to use MySQL connection to store datas :
sql_connection=mysql://glance:password@192.168.42.101/glance

# We bind OpenStack Image API to the VIP :
bind_host = 192.168.42.103

# Connect to OpenStack Image Registry service :
registry_host = 192.168.42.103

# We send notifications to High Available RabbitMQ :
notifier_strategy = rabbit
```

```
rabbit_host = 192.168.42.102
```

Configuring OpenStack Services to use High Available OpenStack Image API

Your OpenStack services must now point their OpenStack Image API configuration to the highly available, virtual cluster IP address — rather than a OpenStack Image API server's physical IP address as you normally would.

For OpenStack Compute, for example, if your OpenStack Image API service IP address is 192.168.42.104 as in the configuration explained here, you would use the following line in your `nova.conf` file :

```
glance_api_servers = 192.168.42.103
```

You need also to create the OpenStack Image API Endpoint with this IP.

NOTE : If you are using both private and public IP addresses, you should create two Virtual IP addresses and define your endpoint like this:

```
keystone endpoint-create --region $KEYSTONE_REGION --service-id $service-id --
publicurl 'http://PUBLIC_VIP:9292' --adminurl 'http://192.168.42.103:9292' --
internalurl 'http://192.168.42.103:9292'
```

Highly available Cinder API

Cinder is the block storage service in OpenStack. Making the Cinder API service highly available in active / passive mode involves

- configuring Cinder to listen on the VIP address,
- managing Cinder API daemon with the Pacemaker cluster manager,
- configuring OpenStack services to use this IP address.

Note

Here is the documentation for installing Cinder service.

Adding Cinder API resource to Pacemaker

First of all, you need to download the resource agent to your system :

```
cd /usr/lib/ocf/resource.d/openstack
wget https://raw.github.com/madkiss/openstack-resource-agents/master/ocf/
cinder-api
chmod a+rx *
```

You may now proceed with adding the Pacemaker configuration for Cinder API resource. Connect to the Pacemaker cluster with `crm configure`, and add the following cluster resources:

```
primitive p_cinder-api ocf:openstack:cinder-api \
    params config="/etc/cinder/cinder.conf" os_password="secrete" os_username=
"admin" \
    os_tenant_name="admin" keystone_get_token_url="http://192.168.42.103:5000/
v2.0/tokens" \
    op monitor interval="30s" timeout="30s"
```

This configuration creates

- p_cinder-api, a resource for manage Cinder API service

crm configure supports batch input, so you may copy and paste the above into your live pacemaker configuration, and then make changes as required. For example, you may enter edit p_ip_cinder-api from the crm configure menu and edit the resource to match your preferred virtual IP address.

Once completed, commit your configuration changes by entering commit from the crm configure menu. Pacemaker will then start the Cinder API service, and its dependent resources, on one of your nodes.

Configuring Cinder API service

Edit /etc/cinder/cinder.conf :

```
# We have to use MySQL connection to store datas :
sql_connection=mysql://cinder:password@192.168.42.101/cinder

# We bind Cinder API to the VIP :
osapi_volume_listen = 192.168.42.103

# We send notifications to High Available RabbitMQ :
notifier_strategy = rabbit
rabbit_host = 192.168.42.102
```

Configuring OpenStack Services to use High Available Cinder API

Your OpenStack services must now point their Cinder API configuration to the highly available, virtual cluster IP address — rather than a Cinder API server's physical IP address as you normally would.

You need to create the Cinder API Endpoint with this IP.

NOTE : If you are using both private and public IP, you should create two Virtual IP and define your endpoint like this :

```
keystone endpoint-create --region $KEYSTONE_REGION --service-id $service-id --
publicurl 'http://PUBLIC_VIP:8776/v1/%(tenant_id)s' --adminurl 'http://192.
168.42.103:8776/v1/%(tenant_id)s' --internalurl 'http://192.168.42.103:8776/
v1/%(tenant_id)s'
```

Highly available OpenStack Networking Server

OpenStack Networking is the network connectivity service in OpenStack. Making the OpenStack Networking Server service highly available in active / passive mode involves

- configuring OpenStack Networking to listen on the VIP address,

- managing OpenStack Networking API Server daemon with the Pacemaker cluster manager,
- configuring OpenStack services to use this IP address.

Note

Here is the documentation for installing OpenStack Networking service.

Adding OpenStack Networking Server resource to Pacemaker

First of all, you need to download the resource agent to your system :

```
cd /usr/lib/ocf/resource.d/openstack
wget https://raw.github.com/madkiss/openstack-resource-agents/master/ocf/
neutron-server
chmod a+rx *
```

You may now proceed with adding the Pacemaker configuration for OpenStack Networking Server resource. Connect to the Pacemaker cluster with `crm configure`, and add the following cluster resources:

```
primitive p_neutron-server ocf:openstack:neutron-server \
    params os_password="secrete" os_username="admin" os_tenant_name="admin" \
    keystone_get_token_url="http://192.168.42.103:5000/v2.0/tokens" \
    op monitor interval="30s" timeout="30s"
```

This configuration creates `p_neutron-server`, a resource for manage OpenStack Networking Server service

`crm configure` supports batch input, so you may copy and paste the above into your live pacemaker configuration, and then make changes as required. For example, you may enter `edit p_neutron-server` from the `crm configure` menu and edit the resource to match your preferred virtual IP address.

Once completed, commit your configuration changes by entering `commit` from the `crm configure` menu. Pacemaker will then start the OpenStack Networking API service, and its dependent resources, on one of your nodes.

Configuring OpenStack Networking Server

Edit `/etc/neutron/neutron.conf` :

```
# We bind the service to the VIP :
bind_host = 192.168.42.103

# We bind OpenStack Networking Server to the VIP :
bind_host = 192.168.42.103

# We send notifications to Highly available RabbitMQ :
notifier_strategy = rabbit
rabbit_host = 192.168.42.102

[database]
# We have to use MySQL connection to store datas :
connection = mysql://neutron:password@192.168.42.101/neutron
```

Configuring OpenStack Services to use Highly available OpenStack Networking Server

Your OpenStack services must now point their OpenStack Networking Server configuration to the highly available, virtual cluster IP address — rather than an OpenStack Networking server's physical IP address as you normally would.

For example, you should configure OpenStack Compute for using Highly Available OpenStack Networking Server in editing `nova.conf` file :

```
neutron_url = http://192.168.42.103:9696
```

You need to create the OpenStack Networking Server Endpoint with this IP.

NOTE : If you are using both private and public IP addresses, you should create two Virtual IP addresses and define your endpoint like this:

```
keystone endpoint-create --region $KEYSTONE_REGION --service-id $service-id --
publicurl 'http://PUBLIC_VIP:9696/' --adminurl 'http://192.168.42.103:9696/'
 --internalurl 'http://192.168.42.103:9696/'
```

Highly available Ceilometer Central Agent

Ceilometer is the metering service in OpenStack. Central Agent polls for resource utilization statistics for resources not tied to instances or compute nodes.

Note

Due to limitations of a polling model, a single instance of this agent can be polling a given list of meters. In this setup, we install this service on the API nodes also in the active / passive mode.

Making the Ceilometer Central Agent service highly available in active / passive mode involves managing its daemon with the Pacemaker cluster manager,

Note

You will find at this page the process to install Ceilometer Central Agent.

Adding Ceilometer Central Agent resource to Pacemaker

First of all, you need to download the resource agent to your system :

```
cd /usr/lib/ocf/resource.d/openstack
wget https://raw.github.com/madkiss/openstack-resource-agents/master/ocf/
ceilometer-agent-central
chmod a+rx *
```

You may then proceed with adding the Pacemaker configuration for Ceilometer Central Agent resource. Connect to the Pacemaker cluster with `crm configure`, and add the following cluster resources:

```
primitive p_ceilometer-agent-central ocf:openstack:ceilometer-agent-central \
    params config="/etc/ceilometer/ceilometer.conf" \
    op monitor interval="30s" timeout="30s"
```

This configuration creates

- `p_ceilometer-agent-central`, a resource for manage Ceilometer Central Agent service

`crm configure` supports batch input, so you may copy and paste the above into your live pacemaker configuration, and then make changes as required.

Once completed, commit your configuration changes by entering `commit` from the `crm configure` menu. Pacemaker will then start the Ceilometer Central Agent service, and its dependent resources, on one of your nodes.

Configuring Ceilometer Central Agent service

Edit `/etc/ceilometer/ceilometer.conf` :

```
# We use API VIP for Identity Service connection :
os_auth_url=http://192.168.42.103:5000/v2.0

# We send notifications to High Available RabbitMQ :
notifier_strategy = rabbit
rabbit_host = 192.168.42.102

[database]
# We have to use MySQL connection to store datas :
sql_connection=mysql://ceilometer:password@192.168.42.101/ceilometer
```

Configure Pacemaker Group

Finally, we need to create a service `group` to ensure that virtual IP is linked to the API services resources :

```
group g_services_api p_api-ip p_keystone p_glance-api p_cinder-api \
   p_neutron-server p_glance-registry p_ceilometer-agent-central
```

Network Controller Cluster Stack

The Network controller sits on the management and data network, and needs to be connected to the Internet if a VM needs access to it.

Note

Both nodes should have the same hostname since the Neutron scheduler will be aware of one node, for example a virtual router attached to a single L3 node.

Highly available Neutron L3 Agent

The Neutron L3 agent provides L3/NAT forwarding to ensure external network access for VMs on tenant networks. High Availability for the L3 agent is achieved by adopting Pacemaker.

Note

Here is the documentation for installing Neutron L3 Agent.

Adding Neutron L3 Agent resource to Pacemaker

First of all, you need to download the resource agent to your system:

```
cd /usr/lib/ocf/resource.d/openstack
wget https://raw.github.com/madkiss/openstack-resource-agents/master/ocf/
neutron-l3-agent
chmod a+rx neutron-l3-agent
```

You may now proceed with adding the Pacemaker configuration for Neutron L3 Agent resource. Connect to the Pacemaker cluster with `crm configure`, and add the following cluster resources:

```
primitive p_neutron-l3-agent ocf:openstack:neutron-l3-agent \
    params config="/etc/neutron/neutron.conf" \
    plugin_config="/etc/neutron/l3_agent.ini" \
    op monitor interval="30s" timeout="30s"
```

This configuration creates

- `p_neutron-l3-agent`, a resource for manage Neutron L3 Agent service

`crm configure` supports batch input, so you may copy and paste the above into your live pacemaker configuration, and then make changes as required.

Once completed, commit your configuration changes by entering `commit` from the `crm configure` menu. Pacemaker will then start the Neutron L3 Agent service, and its dependent resources, on one of your nodes.

Note

This method does not ensure a zero downtime since it has to recreate all the namespaces and virtual routers on the node.

Highly available Neutron DHCP Agent

Neutron DHCP agent distributes IP addresses to the VMs with dnsmasq (by default). High Availability for the DHCP agent is achieved by adopting Pacemaker.

Note

Here is the documentation for installing Neutron DHCP Agent.

Adding Neutron DHCP Agent resource to Pacemaker

First of all, you need to download the resource agent to your system :

```
cd /usr/lib/ocf/resource.d/openstack
wget https://raw.github.com/madkiss/openstack-resource-agents/master/ocf/
neutron-dhcp-agent
```

```
chmod a+rx neutron-dhcp-agent
```

You may now proceed with adding the Pacemaker configuration for Neutron DHCP Agent resource. Connect to the Pacemaker cluster with `crm configure`, and add the following cluster resources:

```
primitive p_neutron-dhcp-agent ocf:openstack:neutron-dhcp-agent \
    params config="/etc/neutron/neutron.conf" \
    plugin_config="/etc/neutron/dhcp_agent.ini" \
    op monitor interval="30s" timeout="30s"
```

This configuration creates

- `p_neutron-dhcp-agent`, a resource for manage Neutron DHCP Agent service

`crm configure` supports batch input, so you may copy and paste the above into your live pacemaker configuration, and then make changes as required.

Once completed, commit your configuration changes by entering `commit` from the `crm configure` menu. Pacemaker will then start the Neutron DHCP Agent service, and its dependent resources, on one of your nodes.

Highly available Neutron Metadata Agent

Neutron Metadata agent allows Nova API Metadata to be reachable by VMs on tenant networks. High Availability for the Metadata agent is achieved by adopting Pacemaker.

Note

Here is the documentation for installing Neutron Metadata Agent.

Adding Neutron Metadata Agent resource to Pacemaker

First of all, you need to download the resource agent to your system:

```
cd /usr/lib/ocf/resource.d/openstack
wget https://raw.github.com/madkiss/openstack-resource-agents/master/ocf/
neutron-metadata-agent
chmod a+rx neutron-metadata-agent
```

You may now proceed with adding the Pacemaker configuration for Neutron Metadata Agent resource. Connect to the Pacemaker cluster with `crm configure`, and add the following cluster resources:

```
primitive p_neutron-metadata-agent ocf:openstack:neutron-metadata-agent \
    params config="/etc/neutron/neutron.conf" \
    plugin_config="/etc/neutron/metadata_agent.ini" \
    op monitor interval="30s" timeout="30s"
```

This configuration creates

- `p_neutron-metadata-agent`, a resource for manage Neutron Metadata Agent service

`crm configure` supports batch input, so you may copy and paste the above into your live pacemaker configuration, and then make changes as required.

Once completed, commit your configuration changes by entering `commit` from the `crm configure` menu. Pacemaker will then start the Neutron Metadata Agent service, and its dependent resources, on one of your nodes.

Manage network resources

You may now proceed with adding the Pacemaker configuration for managing all network resources together with a group. Connect to the Pacemaker cluster with `crm configure`, and add the following cluster resources:

```
group g_services_network p_neutron-l3-agent p_neutron-dhcp-agent \
    p_neutron-metadata_agent
```

3. HA Using Active/Active

Table of Contents

Database

The first step is installing the database that sits at the heart of the cluster. When we're talking about High Availability, however, we're talking about not just one database, but several (for redundancy) and a means to keep them synchronized. In this case, we're going to choose the MySQL database, along with Galera for synchronous multi-master replication.

The choice of database isn't a foregone conclusion; you're not required to use MySQL. It is, however, a fairly common choice in OpenStack installations, so we'll cover it here.

MySQL with Galera

Rather than starting with a vanilla version of MySQL and then adding Galera to it, you will want to install a version of MySQL patched for wsrep (Write Set REPlication) from https://launchpad.net/codership-mysql/0.7. Note that the installation requirements are a bit touchy; you will want to make sure to read the README file so you don't miss any steps.

Next, download Galera itself from https://launchpad.net/galera/+download. Go ahead and install the *.rpms or *.debs, taking care of any dependencies that your system doesn't already have installed.

Once you've completed the installation, you'll need to make a few configuration changes:

In the system-wide `my.conf` file, make sure mysqld isn't bound to 127.0.0.1, and that `/etc/mysql/conf.d/` is included. Typically you can find this file at `/etc/my.cnf`:

```
[mysqld]
...
```

```
!includedir /etc/mysql/conf.d/
...
#bind-address = 127.0.0.1
```

When adding a new node, you must configure it with a MySQL account that can access the other nodes so that it can request a state snapshot from one of those existing nodes. First specify that account information in `/etc/mysql/conf.d/wsrep.cnf`:

```
wsrep_sst_auth=wsrep_sst:wspass
```

Next connect as root and grant privileges to that user:

```
$ mysql -e "SET wsrep_on=OFF; GRANT ALL ON *.* TO wsrep_sst@'%' IDENTIFIED BY
 'wspass'";
```

You'll also need to remove user accounts with empty usernames, as they cause problems:

```
$ mysql -e "SET wsrep_on=OFF; DELETE FROM mysql.user WHERE user='';"
```

You'll also need to set certain mandatory configuration options within MySQL itself. These include:

```
query_cache_size=0
binlog_format=ROW
default_storage_engine=innodb
innodb_autoinc_lock_mode=2
innodb_doublewrite=1
```

Finally, make sure that the nodes can access each other through the firewall. This might mean adjusting iptables, as in:

```
# iptables --insert RH-Firewall-1-INPUT 1 --proto tcp --source <my IP>/24 --
destination <my IP>/32 --dport 3306 -j ACCEPT
# iptables --insert RH-Firewall-1-INPUT 1 --proto tcp --source <my IP>/24 --
destination <my IP>/32 --dport 4567 -j ACCEPT
```

It might also mean configuring any NAT firewall between nodes to allow direct connections, or disabling SELinux or configuring it to allow mysqld to listen to sockets at unprivileged ports.

Now you're ready to actually create the cluster.

Creating the cluster

In creating a cluster, you first start a single instance, which creates the cluster. The rest of the MySQL instances then connect to that cluster. For example, if you started on `10.0.0.10` by executing the command:

```
# service mysql start wsrep_cluster_address=gcomm://
```

you could then connect to that cluster on the rest of the nodes by referencing the address of that node, as in:

```
# service mysql start wsrep_cluster_address=gcomm://10.0.0.10
```

You also have the option to set the `wsrep_cluster_address` in the `/etc/mysql/conf.d/wsrep.cnf` file, or within the client itself. (In fact, for some systems, such as MariaDB or Percona, this may be your only option.) For example, to check the status of the cluster, open the MySQL client and check the status of the various parameters:

```
mysql> SET GLOBAL wsrep_cluster_address='<cluster address string>';
mysql> SHOW STATUS LIKE 'wsrep%';
```

You should see a status that looks something like this:

```
mysql> show status like 'wsrep%';
+----------------------------+--------------------------------------+
| Variable_name              | Value                                |
+----------------------------+--------------------------------------+
| wsrep_local_state_uuid     | 111fc28b-1b05-11e1-0800-e00ec5a7c930 |
| wsrep_protocol_version     | 1                                    |
| wsrep_last_committed       | 0                                    |
| wsrep_replicated           | 0                                    |
| wsrep_replicated_bytes     | 0                                    |
| wsrep_received             | 2                                    |
| wsrep_received_bytes       | 134                                  |
| wsrep_local_commits        | 0                                    |
| wsrep_local_cert_failures  | 0                                    |
| wsrep_local_bf_aborts      | 0                                    |
| wsrep_local_replays        | 0                                    |
| wsrep_local_send_queue     | 0                                    |
| wsrep_local_send_queue_avg | 0.000000                             |
| wsrep_local_recv_queue     | 0                                    |
| wsrep_local_recv_queue_avg | 0.000000                             |
| wsrep_flow_control_paused  | 0.000000                             |
| wsrep_flow_control_sent    | 0                                    |
| wsrep_flow_control_recv    | 0                                    |
| wsrep_cert_deps_distance   | 0.000000                             |
| wsrep_apply_oooe           | 0.000000                             |
| wsrep_apply_oool           | 0.000000                             |
| wsrep_apply_window         | 0.000000                             |
| wsrep_commit_oooe          | 0.000000                             |
| wsrep_commit_oool          | 0.000000                             |
| wsrep_commit_window        | 0.000000                             |
| wsrep_local_state          | 4                                    |
| wsrep_local_state_comment  | Synced (6)                           |
| wsrep_cert_index_size      | 0                                    |
| wsrep_cluster_conf_id      | 1                                    |
| wsrep_cluster_size         | 1                                    |
| wsrep_cluster_state_uuid   | 111fc28b-1b05-11e1-0800-e00ec5a7c930 |
| wsrep_cluster_status       | Primary                              |
| wsrep_connected            | ON                                   |
| wsrep_local_index          | 0                                    |
| wsrep_provider_name        | Galera                               |
| wsrep_provider_vendor      | Codership Oy                         |
| wsrep_provider_version     | 21.1.0(r86)                          |
| wsrep_ready                | ON                                   |
+----------------------------+--------------------------------------+
38 rows in set (0.01 sec)
```

Galera Monitoring Scripts

(Coming soon)

Other ways to provide a Highly Available database

MySQL with Galera is by no means the only way to achieve database HA. MariaDB (https://mariadb.org/) and Percona (http://www.percona.com/) also work with Galera. You also have the option to use Postgres, which has its own replication, or some other database HA option.

RabbitMQ

RabbitMQ is the default AMQP server used by many OpenStack services. Making the RabbitMQ service highly available involves:

- Install RabbitMQ
- Configure RabbitMQ for HA queues
- Configure OpenStack services to use Rabbit HA queues

Install RabbitMQ

This setup has been tested with RabbitMQ 2.7.1.

On Ubuntu / Debian

RabbitMQ is packaged on both distros:

```
apt-get install rabbitmq-server rabbitmq-plugins
```

Official manual for installing RabbitMQ on Ubuntu / Debian

On Fedora / RHEL

RabbitMQ is packaged on both distros:

```
yum install erlang
```

Official manual for installing RabbitMQ on Fedora / RHEL

Configure RabbitMQ

Here we are building a cluster of RabbitMQ nodes to construct a RabbitMQ broker. Mirrored queues in RabbitMQ improve the availability of service since it will be resilient to failures, but we have to consider that while exchanges and bindings will survive the loss of individual nodes, queues and their messages will no because a queue and its contents is located on one node. So if we lose this node, we also lose the queue.

We consider that we run (at least) two RabbitMQ servers. To build a broker, we need to ensure that all nodes have the same erlang cookie file. To do so, stop RabbitMQ everywhere and copy the cookie from rabbit1 server to other server(s):

```
scp /var/lib/rabbitmq/.erlang.cookie \
root@rabbit2:/var/lib/rabbitmq/.erlang.cookie
```

Then, start RabbitMQ on nodes. If RabbitMQ fails to start, you can’t continue to the next step.

Now, we are building the HA cluster. From rabbit2, run these commands:

```
rabbitmqctl stop_app
rabbitmqctl cluster rabbit@rabbit1
rabbitmqctl start_app
```

To verify the cluster status :

```
rabbitmqctl cluster_status

Cluster status of node rabbit@rabbit2 ...
[{nodes,[{disc,[rabbit@rabbit1]},{ram,[rabbit@rabbit2]}]},{running_nodes,
[rabbit@rabbit2,rabbit@rabbit1]}]
```

If the cluster is working, you can now proceed to creating users and password for queues.

Note for RabbitMQ version 3

Queue mirroring is no longer controlled by the *x-ha-policy* argument when declaring a queue. OpenStack can continue to declare this argument, but it won't cause queues to be mirrored. We need to make sure that all queues (except those with auto-generated names) are mirrored across all nodes in running:

```
rabbitmqctl set_policy HA '^(?!amq\.).*' '{"ha-mode": "all"}'
```

More informations about High availability in RabbitMQ

Configure OpenStack Services to use RabbitMQ

Since Grizzly Release, most of OpenStack components using queuing has been supported the feature, we have to configure them to use at least two RabbitMQ nodes.

Do this configuration on all services using RabbitMQ:

RabbitMQ HA cluster host:port pairs:

```
rabbit_hosts=rabbit1:5672,rabbit2:5672
```

How frequently to retry connecting with RabbitMQ:

```
rabbit_retry_interval=1
```

How long to backoff for between retries when connecting to RabbitMQ:

```
rabbit_retry_backoff=2
```

Maximum retries with trying to connect to RabbitMQ (infinite by default):

```
rabbit_max_retries=0
```

Use durable queues in RabbitMQ:

```
rabbit_durable_queues=false
```

Use H/A queues in RabbitMQ (x-ha-policy: all):

```
rabbit_ha_queues=true
```

If you change the configuration from an old setup which did not use HA queues, you should interrupt the service :

```
rabbitmqctl stop_app
rabbitmqctl reset
rabbitmqctl start_app
```

Services currently working with HA queues : OpenStack Compute, Cinder, OpenStack Networking, Ceilometer.

HAproxy Nodes

HAProxy is a very fast and reliable solution offering high availability, load balancing, and proxying for TCP and HTTP-based applications. It is particularly suited for web sites crawling under very high loads while needing persistence or Layer 7 processing. Supporting tens of thousands of connections is clearly realistic with today’s hardware.

For installing HAproxy on your nodes, you should consider its official documentation. Also, you have to consider that this service should not be a single point of failure, so you need at least two nodes running HAproxy.

Here is an example for HAproxy configuration file:

```
global
  chroot  /var/lib/haproxy
  daemon
  group  haproxy
  maxconn  4000
  pidfile  /var/run/haproxy.pid
  user  haproxy

defaults
  log  global
  maxconn  8000
  option  redispatch
  retries  3
  timeout  http-request 10s
  timeout  queue 1m
  timeout  connect 10s
  timeout  client 1m
  timeout  server 1m
  timeout  check 10s

listen dashboard_cluster
  bind <Virtual IP>:443
  balance  source
  option  tcpka
  option  httpchk
  option  tcplog
  server controller1 10.0.0.1:443 check inter 2000 rise 2 fall 5
  server controller2 10.0.0.2:443 check inter 2000 rise 2 fall 5

listen galera_cluster
  bind <Virtual IP>:3306
  balance  source
  option  httpchk
  server controller1 10.0.0.4:3306 check port 9200 inter 2000 rise 2 fall 5
  server controller2 10.0.0.5:3306 check port 9200 inter 2000 rise 2 fall 5
  server controller3 10.0.0.6:3306 check port 9200 inter 2000 rise 2 fall 5

listen glance_api_cluster
  bind <Virtual IP>:9292
  balance  source
  option  tcpka
  option  httpchk
  option  tcplog
  server controller1 10.0.0.1:9292 check inter 2000 rise 2 fall 5
  server controller2 10.0.0.2:9292 check inter 2000 rise 2 fall 5
```

```
listen glance_registry_cluster
  bind <Virtual IP>:9191
  balance  source
  option  tcpka
  option  tcplog
  server controller1 10.0.0.1:9191 check inter 2000 rise 2 fall 5
  server controller2 10.0.0.2:9191 check inter 2000 rise 2 fall 5

listen keystone_admin_cluster
  bind <Virtual IP>:35357
  balance  source
  option  tcpka
  option  httpchk
  option  tcplog
  server controller1 10.0.0.1:35357 check inter 2000 rise 2 fall 5
  server controller2 10.0.0.2.42:35357 check inter 2000 rise 2 fall 5

listen keystone_public_internal_cluster
  bind <Virtual IP>:5000
  balance  source
  option  tcpka
  option  httpchk
  option  tcplog
  server controller1 10.0.0.1:5000 check inter 2000 rise 2 fall 5
  server controller2 10.0.0.2:5000 check inter 2000 rise 2 fall 5

listen nova_ec2_api_cluster
  bind <Virtual IP>:8773
  balance  source
  option  tcpka
  option  tcplog
  server controller1 10.0.0.1:8773 check inter 2000 rise 2 fall 5
  server controller2 10.0.0.2:8773 check inter 2000 rise 2 fall 5

listen nova_compute_api_cluster
  bind <Virtual IP>:8774
  balance  source
  option  tcpka
  option  httpchk
  option  tcplog
  server controller1 10.0.0.1:8774 check inter 2000 rise 2 fall 5
  server controller2 10.0.0.2:8774 check inter 2000 rise 2 fall 5

listen nova_metadata_api_cluster
  bind <Virtual IP>:8775
  balance  source
  option  tcpka
  option  tcplog
  server controller1 10.0.0.1:8775 check inter 2000 rise 2 fall 5
  server controller2 10.0.0.2:8775 check inter 2000 rise 2 fall 5

listen cinder_api_cluster
  bind <Virtual IP>:8776
  balance  source
  option  tcpka
  option  httpchk
  option  tcplog
  server controller1 10.0.0.1:8776 check inter 2000 rise 2 fall 5
  server controller2 10.0.0.2:8776 check inter 2000 rise 2 fall 5
```

```
listen ceilometer_api_cluster
  bind <Virtual IP>:8777
  balance  source
  option  tcpka
  option  httpchk
  option  tcplog
  server controller1 10.0.0.1:8774 check inter 2000 rise 2 fall 5
  server controller2 10.0.0.2:8774 check inter 2000 rise 2 fall 5

listen spice_cluster
  bind <Virtual IP>:6082
  balance  source
  option  tcpka
  option  tcplog
  server controller1 10.0.0.1:6080 check inter 2000 rise 2 fall 5
  server controller2 10.0.0.2:6080 check inter 2000 rise 2 fall 5

listen neutron_api_cluster
  bind <Virtual IP>:9696
  balance  source
  option  tcpka
  option  httpchk
  option  tcplog
  server controller1 10.0.0.1:9696 check inter 2000 rise 2 fall 5
  server controller2 10.0.0.2:9696 check inter 2000 rise 2 fall 5

listen swift_proxy_cluster
  bind <Virtual IP>:8080
  balance  source
  option  tcplog
  option  tcpka
  server controller1 10.0.0.1:8080 check inter 2000 rise 2 fall 5
  server controller2 10.0.0.2:8080 check inter 2000 rise 2 fall 5
```

After each change of this file, you should restart HAproxy.

OpenStack Controller Nodes

OpenStack Controller Nodes contains:

- All OpenStack API services
- All OpenStack schedulers
- Memcached service

Running OpenStack API & schedulers

API Services

All OpenStack projects have an API service for controlling all the resources in the Cloud. In Active / Active mode, the most common setup is to scale-out these services on at least two nodes and use load-balancing and virtual IP (with HAproxy & Keepalived in this setup).

Configuring API OpenStack services

To configure our Cloud using Highly available and scalable API services, we need to ensure that:

- Using Virtual IP when configuring OpenStack Identity Endpoints.
- All OpenStack configuration files should refer to Virtual IP.

In case of failure

The monitor check is quite simple since it just establishes a TCP connection to the API port. Comparing to the Active / Passive mode using Corosync & Resources Agents, we don’t check if the service is actually running). That’s why all OpenStack API should be monitored by another tool (i.e. Nagios) with the goal to detect failures in the Cloud Framework infrastructure.

Schedulers

OpenStack schedulers are used to determine how to dispatch compute, network and volume requests. The most common setup is to use RabbitMQ as messaging system already documented in this guide. Those services are connected to the messaging backend and can scale-out :

- nova-scheduler
- nova-conductor
- cinder-scheduler
- neutron-server
- ceilometer-collector
- heat-engine

Please refer to the RabbitMQ section for configure these services with multiple messaging servers.

Memcached

Most of OpenStack services use an application to offer persistence and store ephemeral datas (like tokens). Memcached is one of them and can scale-out easily without specific trick.

To install and configure it, you can read the official documentation.

Memory caching is managed by Oslo-incubator for so the way to use multiple memcached servers is the same for all projects.

Example with two hosts:

```
memcached_servers = controller1:11211,controller2:11211
```

By default, controller1 will handle the caching service but if the host goes down, controller2 will do the job. More informations about memcached installation are in the OpenStack Compute Manual.

OpenStack Network Nodes

OpenStack Network Nodes contains:

- Neutron DHCP Agent
- Neutron L2 Agent
- Neutron L3 Agent
- Neutron Metadata Agent
- Neutron LBaaS Agent

Note

The Neutron L2 Agent does not need to be highly available. It has to be installed on each Data Forwarding Node and controls the virtual networking drivers as Open-vSwitch or Linux Bridge. One L2 agent runs per node and controls its virtual interfaces. That’s why it cannot be distributed and highly available.

Running Neutron DHCP Agent

Since the Grizzly release, OpenStack Networking service has a scheduler that lets you run multiple agents across nodes. Also, the DHCP agent can be natively highly available. For details, see OpenStack Configuration Reference.

Running Neutron L3 Agent

Since the Grizzly release, the Neutron L3 Agent is scalable thanks to the scheduler that allows distribution of virtual routers across multiple nodes. But there is no native feature to make these routers highly available. At this time, the Active / Passive solution exists to run the Neutron L3 agent in failover mode with Pacemaker. See the Active / Passive section of this guide.

Running Neutron Metadata Agent

There is no native feature to make this service highly available. At this time, the Active / Passive solution exists to run the Neutron Metadata agent in failover mode with Pacemaker. See the Active / Passive section of this guide.