Advanced Openstack Administration

Student guide

Release L rev39

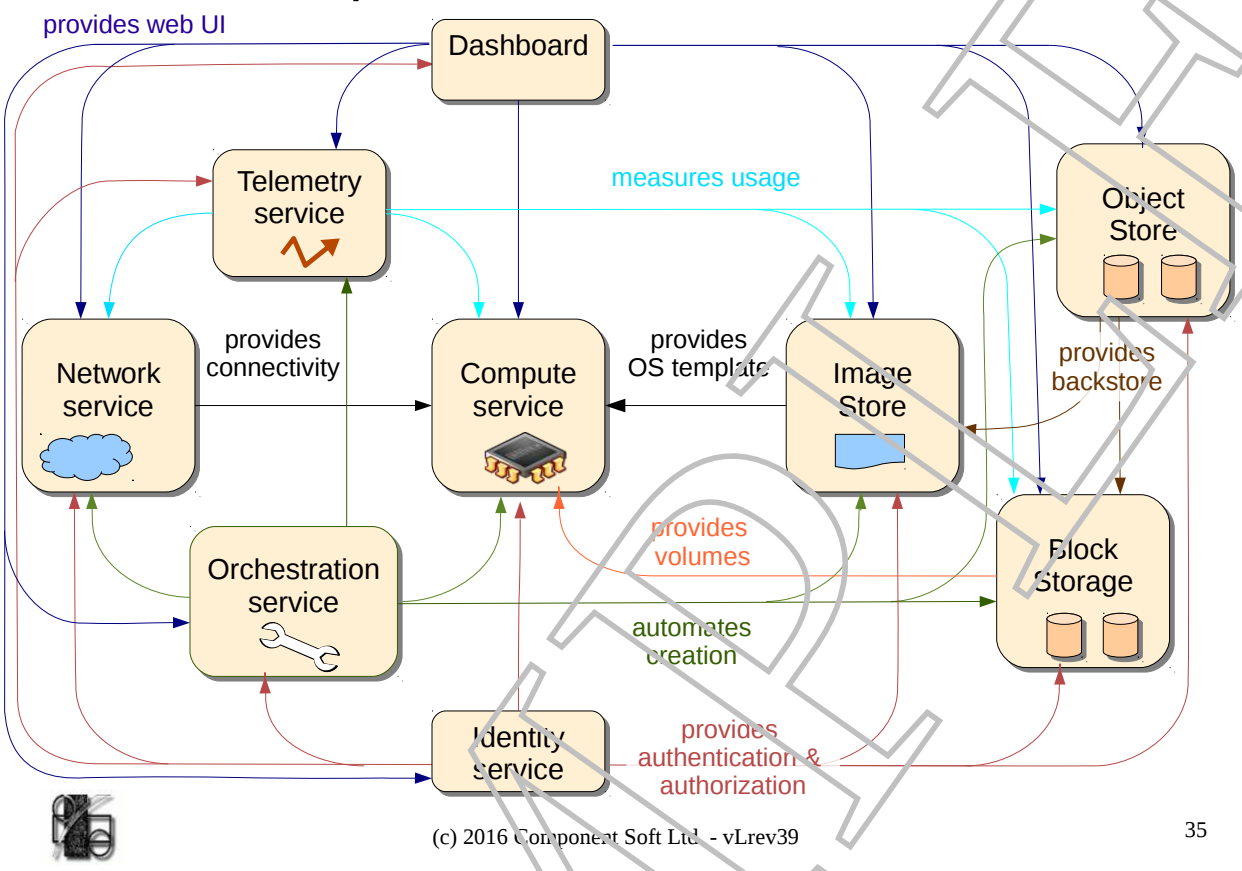
Component Soft Ltd.

November 20, 2016

Contents

Module 2: Introduction

- Overview of Openstack
- Lab environment
- OpenStack HA solutions
  - Active/Passive clustered services
  - Active/Active clustered services



This slide gives a short summary of architecture of OpenStack. Learn more about the OpenStack basics at the training OpenStack Administation Workshop (OST-104), which is the prerequisite of the current training.

OpenStack is defined as a collection of independently developed software projects, glued together. Each service has an HTTP REST interface to communicate with OpenStack clients. To get the full list of services in OpenStack, please visit the OpenStack Project Navigator

At first glance, all services use the identity service, since every single service has to validate the authentication *token* received from its clients. On the other hand, the Identity service also provides a catalog of REST endpoints, so every service has to register itself into the catalog to make itself known to OpenStack clients.

The other general service is the Dashboard, nicknamed **Horizon**, which is a Python/Django based Web GUI, with plugins for many of the projects in OpenStack.

The heart of the whole OpenStack installation is the Compute service (code name **Nova**). This service is responsible for the creation of virtual machines on the top of a hypervisor (such as KVM, Xen, VMWare, etc.).

The Image service (code name **Glance**) is a simple storage for pre-installed OS images. Images can either be stored locally on the server where Glance is running, or uploaded to remote storage, such as the Object Store.

The Block Storage service (code name **Cinder**) provides additional volumes besides the OS images provided by Glance. This service can also use the Object store to store backups of data volumes.

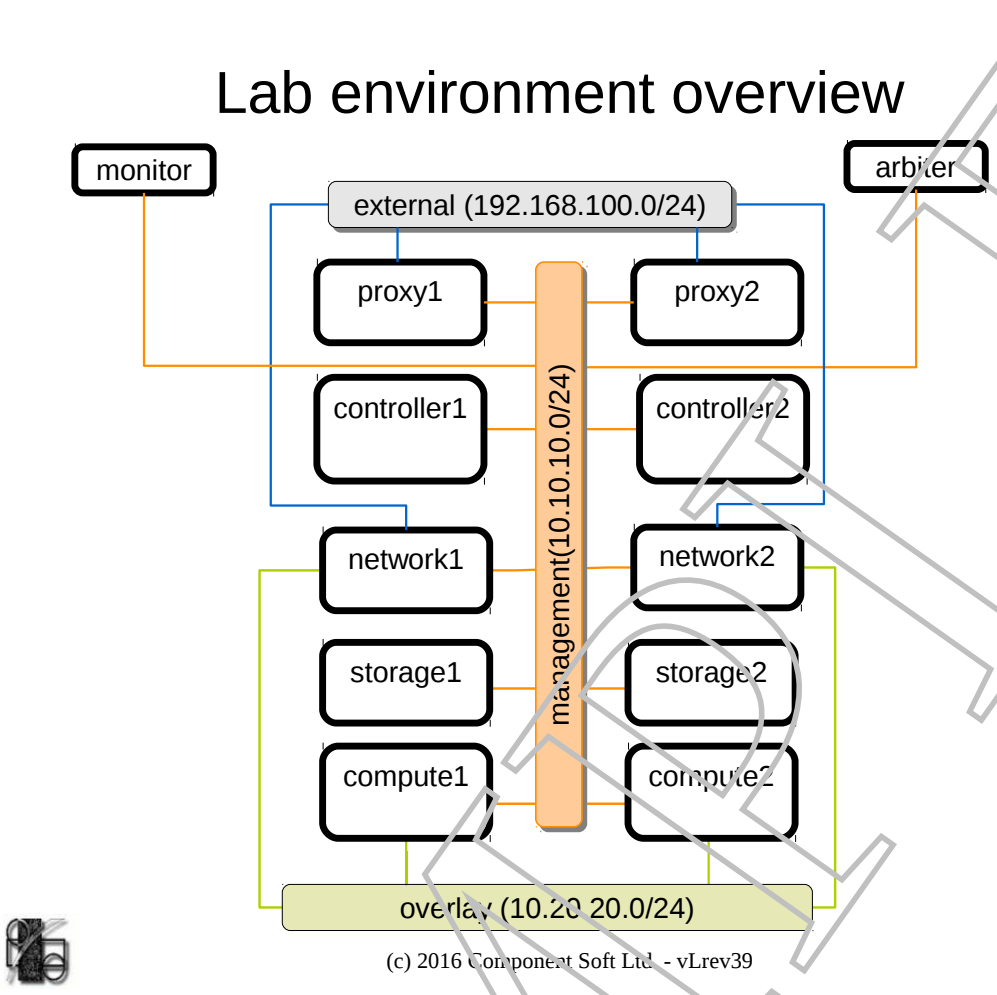
The Object Store (code name **Swift**) is a petabyte scale storage built from hundreds of commodity servers and cheap disks. Files are up/downloaded via a simple REST/HTTP interface.

The Network service (code name **Neutron**) provides Software Defined Networking (SDN) functionality for OpenStack. With Neutron, users can build up complex network topologies to connect their VMs.

The telemetry service (called **Ceilometer**) collects resource usage information such as CPU cycles for virtual machines, storage space used out of the Block Storage, etc. . This telemetry data can be used to create bills for customers. This service is also capable of raising alarms in case of a resource overuse (such high CPU usage in a VM).

The orchestration service, nicknamed **Heat** makes it easier to provision a larger set of OpenStack resources. With Heat you can describe your resources in plain text documents (templates), and simply launch 'stacks' from those templates. A stack can start, for example, multiple VMs provisioned with the same software stack. In collaboration with the telemetry service, it could also scale up/down a stack on demand.

Please check the conceptual architecture and the logical architecture document for a more detailed overview.



Each student desktop is running a complete Openstack environment. The host machine runs KVM instances as OpenStack nodes. The below table give a summary of the resources assigned to these KVM instances.

Node	Memory	CPU
controller1	3G	2
controller2	3G	2
network1	1G	1
network2	1G	1
compute1	1.5G	1
compute2	1.5G	1
storage1	1G	1
storage2	1G	1
arbiter	1G	1
monitor	2G	1
proxy1	512M	1
proxy2	512M	1

Our lab environment uses 3 networks.

- br_management (`10.10.10.0/24`)

  This network is used to log into the nodes. Internal signaling and REST API calls also happen on that network.

- br_external (`192.168.100.0/24`)

  This is the so called *external* network.

  - Floating IPs are allocated from the IP range of this network.
  - The externally accessible REST API endpoints use the *external* IP of the controller.
  - The Horizon dashboard can also be accessed via the *external* IP of the controller.

- br_internal (`10.20.20.0/24`)

  This *overlay* network is between the network and the compute nodes, and hosts tenant networks.

The IP addresses used by KVM instances are summarized in the following table.

Node	br_management	br_internal	br_external
lab machine	10.10.10.1	10.20.20.1	192.168.100.1
controller1	10.10.10.51	—	192.168.100.51
network1	10.10.10.52	10.20.20.52	192.168.100.52
compute1	10.10.10.53	10.20.20.53	—
compute2	10.10.10.54	10.20.20.54	—
storage1	10.10.10.55	—	—
storage2	10.10.10.56	—	—
controller2	10.10.10.61	—	192.168.100.61
network2	10.10.10.62	10.20.20.62	192.168.100.62
arbiter	10.10.10.71	—	—
monitor	10.10.10.110	—	—
proxy1	10.10.10.101	—	—
proxy2	10.10.10.102	—	—

Openstack HA

- Most services are stateless/fault tolerant
  - REST API endpoints (like nova-api)
  - Intermediate services (like nova-scheduler)
- Processes bound to external systems
  - Keystone (like with LDAP)
  - Glance/file backend
  - Cinder/cinder-volume/LVM backend
- Neutron: no full HA
  - HA: dhcp, L3-HA, DVR
  - Not HA: lbaas
- Some of the components needs extra care
  - Database: MySQL/Galera
  - AMQ service: RabbitMQ cluster
  - Ceilometer:
    - MongoDB
    - Redis/ceilometer-central-agent

Here we talk about the fault tolerance and scalability of OpenStack services.

- Most OpenStack services are fault tolerant and scalable by design.
  - REST API based services (such as `nova-api`) can have multiple instances, an those can accessed through a plain HTTP load-balancer.
  - AMQP based services (such as `nova-scheduler`) can also have multiple instances as well, that consume/produce messages (jobs) from/to the same message queue.
- Some services have external dependencies
  - Keystone is generally fault tolerant, but if given a dependency to an external system (such as an LDAP server), one should guarantee the reliability of that system as well.
  - Service Glance is susceptible of node failures if using the default backend `glance.store.filesystem.Store`. Using a distributed backend such as `glance.store.swift.Store`, or `glance.store.rbd.Store` solves this problem.

  - Node failures can be tolerated by Cinder regarding `cinder-api` and `cinder-scheduler`, but the process `cinder-volume` has problems:
    * Backend drivers like `cinder.volume.drivers.lvm.LVMVolumeDriver` work for a single node, so a node failure makes all the locally hosted volumes unavailable.
    * Volumes provisioned by `cinder-volume` on `node1` cannot be accessed if `node1` fails, even if another `cinder-volume` is active on `node2`, and the chosen backend is distributed (like `cinder.volume.drivers.rbd.RBDDriver`). The best practice is to run `cinder-volume` as a fail-over service. Learn more aboud this topic at the Cinder HA part of this book.

- The network service is not fully fault tolerant
  - The DHCP service for tenant networks can be served from several network nodes, so it is fault tolerant
  - Tenant routers can be configured to be fault tolerant (L3 HA, DVR)
  - Tenant load-balancers (LBaaS) does not implement high availability (yet) with the default backend `haproxy`, but with external drivers a fault tolerant LBaaS can be achieved.

  Learn more at part Neutron HA.

- Some key services of OpenStack needs extra maintenance to achieve HA.
  - Database

  The internal database server (MySQL by default) is a fundamental part, as most of the OpenStack services (except for Ceilometer) has a local database there.

  Turning your single MySQL instance into a Galera Cluster solves both the reliability and the scalability problem of the database service.

  Learn more at the Galera module of this training.

  - AMQ server

  The message queue server (broker) also plays an important role in OpenStack as a transport layer for internal signaling. A failure to this service renders much of the OpenStack services unusable.

  A multi-node RabbitMQ cluster can provide a reliable AMQ service of OpenStack.

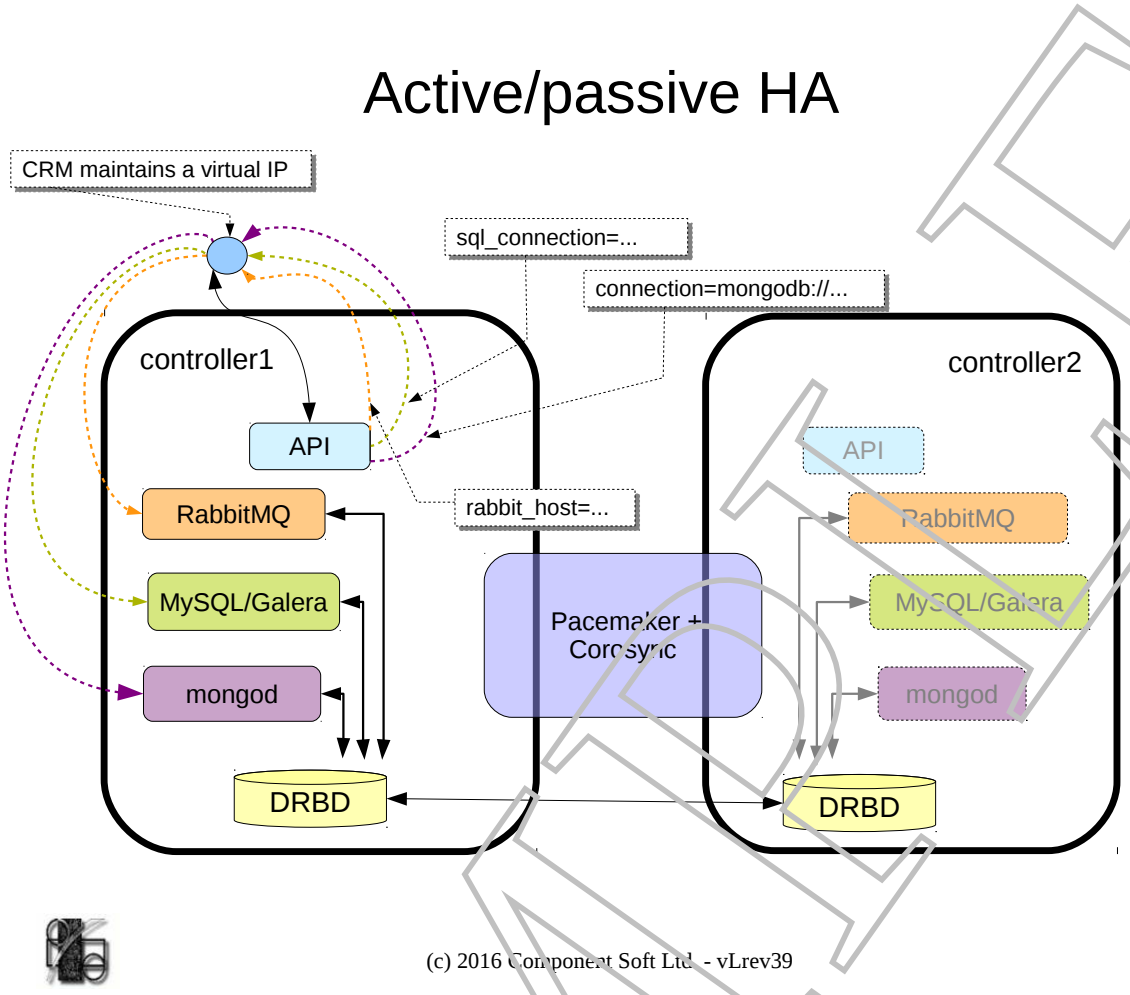
  Learn more at the Rabbit module of this training.

  - Ceilometer related backend services

Ceilometer uses MongoDB as database. One can protect telemetry data by creating MongoDB replicaset. If you also need scalability, you can combine replicasets with the sharding feature of MongoDB.

Ceilometer also uses Redis as a backend for its coordination library Tooz. To make Ceilometer highly available, one should set up a Redis replication.

Ceilometer HA is out of the scope of this training.



Forming an active-passive cluster out of the controller nodes is the first possible approach to make OpenStack services fault tolerant.

In this case, services like MySQL, RabbitMQ or even MongoDB run on one controller node only, and fail over to another node in case of an error. However, stateless processes (like processes implementing REST API endpoints) can run parallel on all controller nodes.

There are several commercial solutions for this problem, but a free one is Pacemaker.

Pacemaker is cluster resource manager (CRM)

- Based on a cluster infrastructure service (like CoroSync), it keeps track of the available cluster nodes (cluster membership), and keeps the cluster wide configuration in sync.
- It has resource agents, scripts that start, stop or monitor a cluster unaware process. End users can define a cluster resource based on that resource agent, and Pacemaker keeps that resource alive through the cluster.

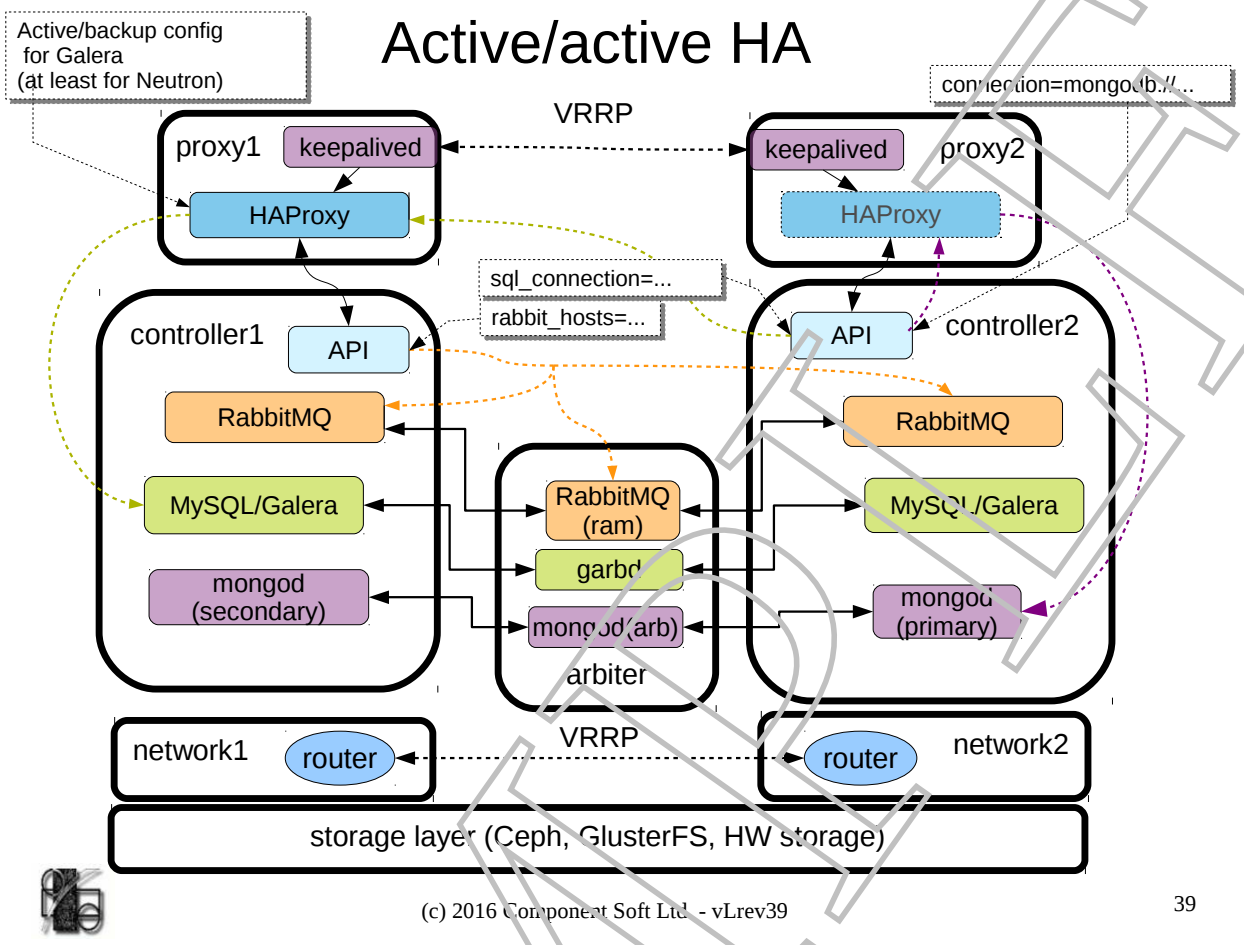
As key feature, in this solution services listen on a so called *virtual IP*, that switches from one node to

another along with the service itself. Other processes (like API processes) should also use this virtual IP to reach the desired service (like MySQL).

The content (like database files) should be shared between nodes, which requires a shared disk (like a volume on a shared fibre channel storage) or shared filesystem. If shared devices are not available, we might use a mirrored block device via DRBD.

The active/passive approach has some caveats:

- If MySQL fails over to another node after a node crash, its startup may be delayed by the InnoDB crash recovery procedure, causing minutes of downtime for OpenStack.
- Data replication with DRBD can slow down operations for MySQL and RabbitMQ.
- MySQL, RabbitMQ and MongoDB do not utilize the resources of the second (or maybe third) node of the cluster.



This approach provides a scalable solution.

Here we use the native clustering of MySQL/Galera, RabbitMQ and MongoDB, so we utilize all controller nodes. A common side effect of this approach, that it requires a third node, which can be either a fully functioning one, or just an *arbiter*. The third node is required by the quorum mechanism inside the cluster.

An additional requirement is a load-balancer: either a Linux/HAProxy based one, or a vendor appliance. In either case it is important to make the LB to handle several virtual IPs

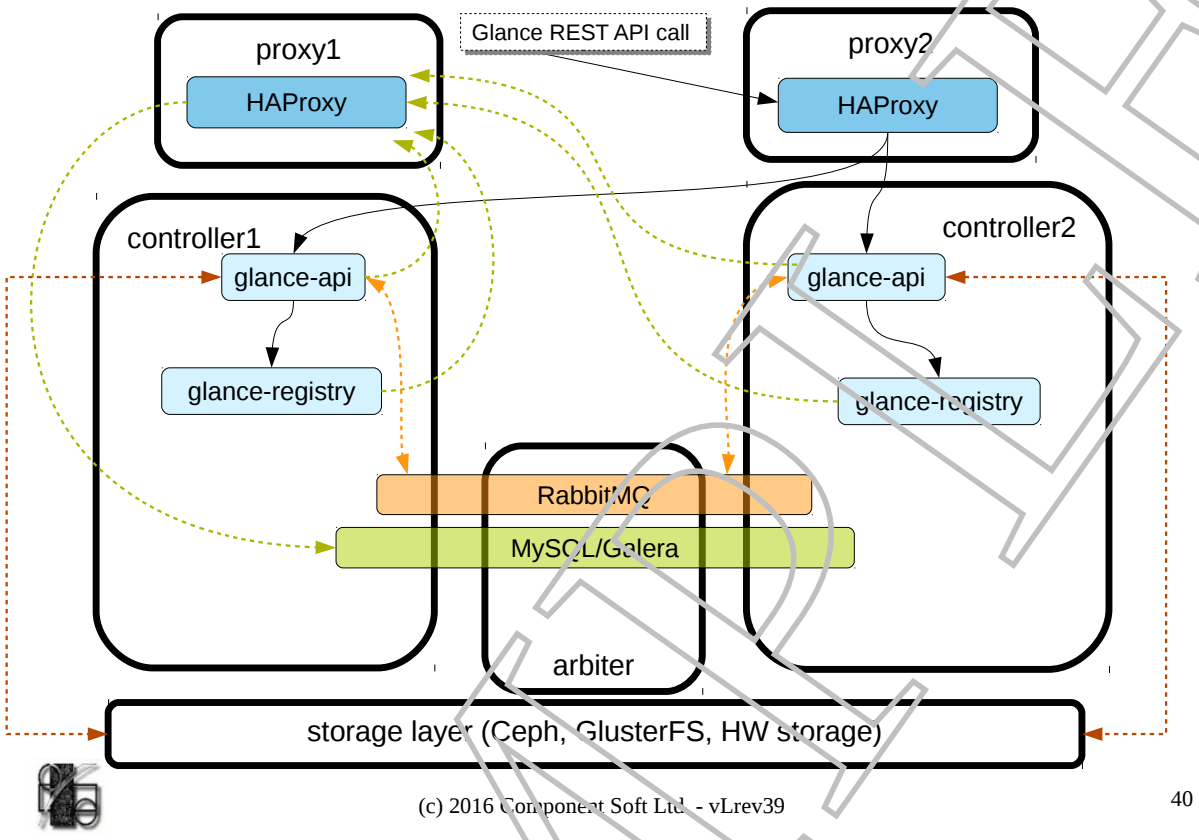
- A VIP which is published to end users via the Keystone catalog. The traffic arriving on the VIP a forwarded to the local API processes.
- Another VIP is configured as an access point for the Galera and MongoDB service. This VIP is used by internal services only.

The above load-balancer should also be fault-tolerant. In the HAProxy case, this can be implemented via keepalived . Keepalived processes talk Virtual Router Redundancy Protocol (VRRP) to elect exactly one active node. The active node can be configured to run a script (for instance to start a service), or it can plumb VIPs on an interface.

The third feature of this approach is the shared (and possibly redundant) storage layer. This layer could be implemented via Ceph , GlusterFS , or even via a vendor storage appliance. In all cases, OpenStack services Glance, Cinder and Nova should use this unified storage layer to store the provisioned resources (images, block volumes, VM ephemeral disks).

The implementation of a Ceph based storage layer is discussed in Module 4.

The following pages describe the details of an active/active HA solution, from the perspective of each OpenStack service.



Glance is probably the simplest service of OpenStack. It is composed of two processes: `glance-api` and `glance-registry`. The later one is required for version 1 API calls only.

A typical API call (both public and internal) first hits the load-balancer, which forwards to one of the controller nodes where a `glance-api` process finally processes the call.

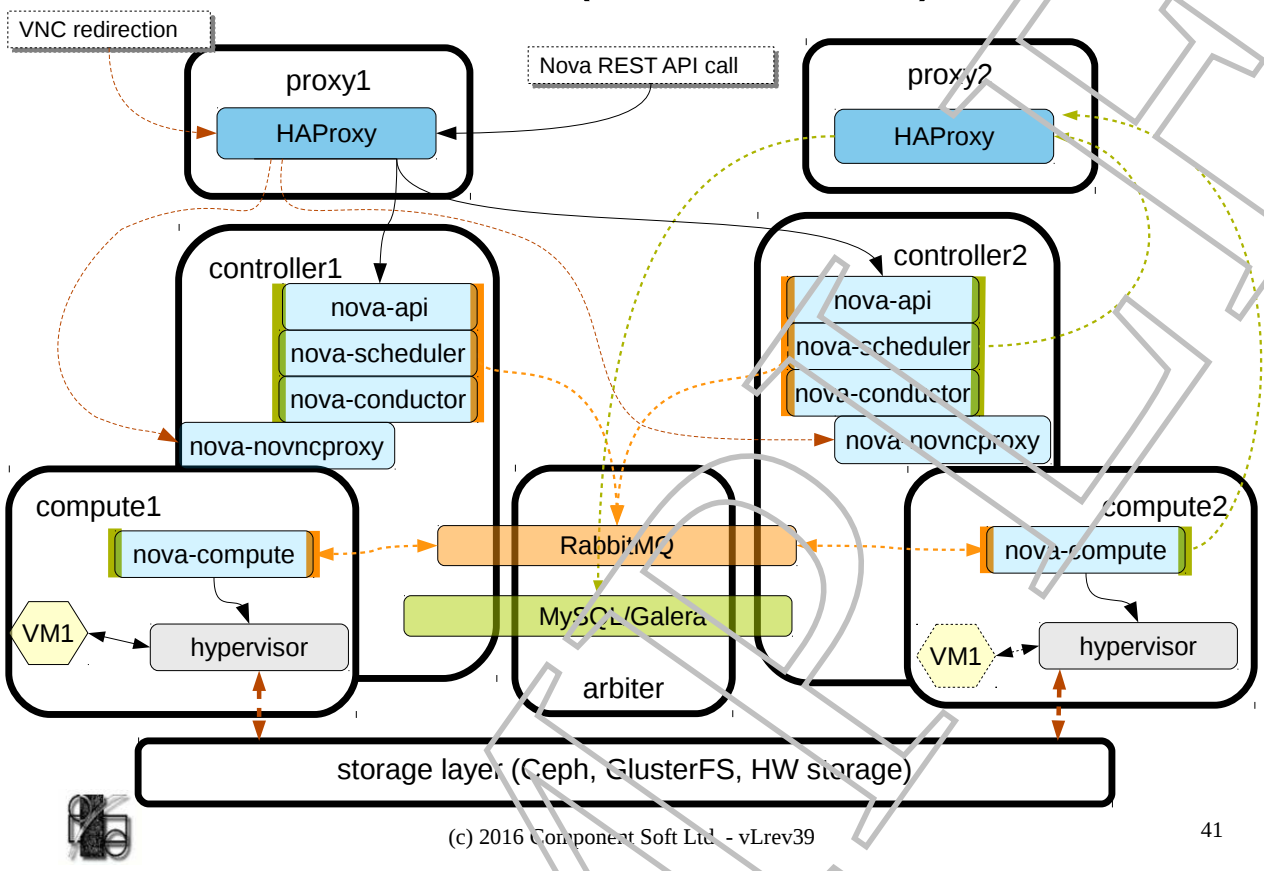
- In case of a APIv1 call, the `glance-api` need the `glance-registry` to read or write the database, so a REST call is sent to `glance-registry`. The Glance database is accessed through the load-balancer, and the LB can forward database connections equally to both Galera nodes.
- In case of APIv2 call, the `glance-api` directly communicates with the database through the LB.

Glance uses the message queue to send notifications to Ceilometer, but the Glance project does not use AMQP for internal signaling. The later makes the message service less important for Glance.

To achieve a reliable image service

- we need to run both `glance-api` and `glance-registry` on all controller nodes.

- a distributed storage backend such as `glance.store.swift.Store`, or `glance.store.rbd.Store` should be used



The compute service is far more complicated than the previously mentioned Glance.

We have two processes needing external connectivity: the REST API process `nova-api`, and the VNC tunnelling appication `nova-novncproxy`. These processes can be accessed via the load-balancer.

Internally, every process (except for `nova-novncproxy`) requires a database and a message queue connection. As usual, the database is accessed through the load-balancer, in order to achieve a fault tolerant database service.

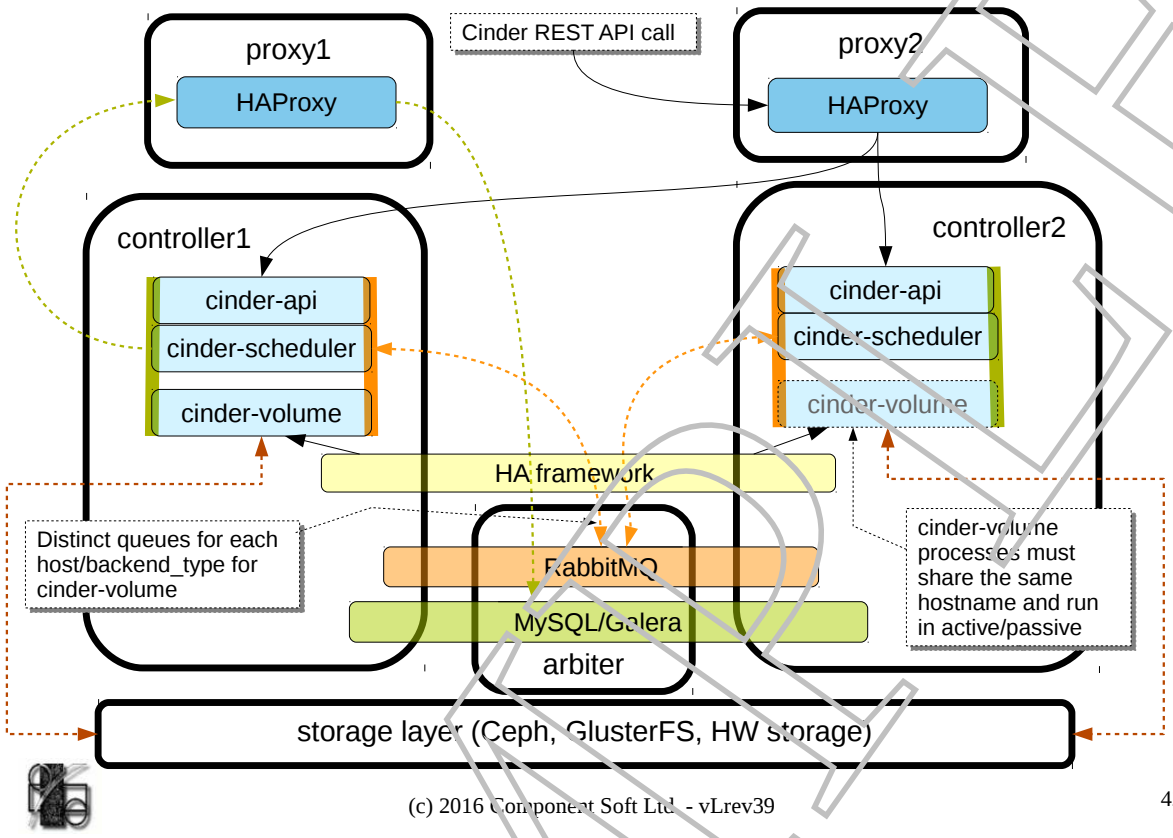
All compute service processes are stateless, so we can easily deploy many instances of them. For example, we can run two `nova-scheduler` processes on different nodes: both will use the same message queue to listen for new request, and also signal to other processes (like to the `nova-compute`) using the same target queue.

Compute node should place both ephemeral and persistent storage devices on a shared storage, to make prompt evacuation possible even in case of a complete node failure.

Note, that the hypervisors, and the created VM instances are not fault tolerant by default. If a compute node fails, the Compute service does not evacuate the VMs hosted by the failed node automatically. This

is partially because there is no standard way to determine reliably whether the give compute node is dead or not. The only thing, we can detect right now is lack or heartbeats from the `nova-compute` process, which does not necessarily means the loss of the compute node or the error of the hypervisor itself.

The problem itself is more complex than just a proper detection of hypervisor node failure, but we also have to be sure, that the corresponding node is fenced out of the OpenStack cluster. The later is required to avoid conflicting writes on shared storage resources (VM disks), as well as duplicated use of network resources (IP/MAC address). At the time of writing, there are proposals but no standard solution for this problem.



Cinder has common features with other services:

- It requires a load-balancer for both the REST API and for the database access.
- Many of it processes (`cinder-api` and `cinder-scheduler`) are stateless, so they scale out nicely to many machines.

In the other hand, `cinder-volume` is a bit problematic when trying create a fault tolerant Cinder service. Cinder registers every `cinder-volume` process plus backend driver pair as distinct service.

```
root@controller1 (admin) $> cinder service-list
+------------------+-------------------------------------+------+---------+-------//
|      Binary      |                Host                 | Zone |  Status | State //
+------------------+-------------------------------------+------+---------+-------//
| cinder-scheduler |   controller1.openstack.local       | nova | enabled |   up  //
| cinder-scheduler |   controller2.openstack.local       | nova | enabled |   up  //
|  cinder-volume   | controller1.openstack.local@lvm     | nova | enabled |   up  //
|  cinder-volume   | controller2.openstack.local@lvm     | nova | enabled |   up  //
+------------------+-------------------------------------+------+---------+-------//
```

When a new volume is created, `cinder-volume` registers the host and the backend name to a volume attribute.

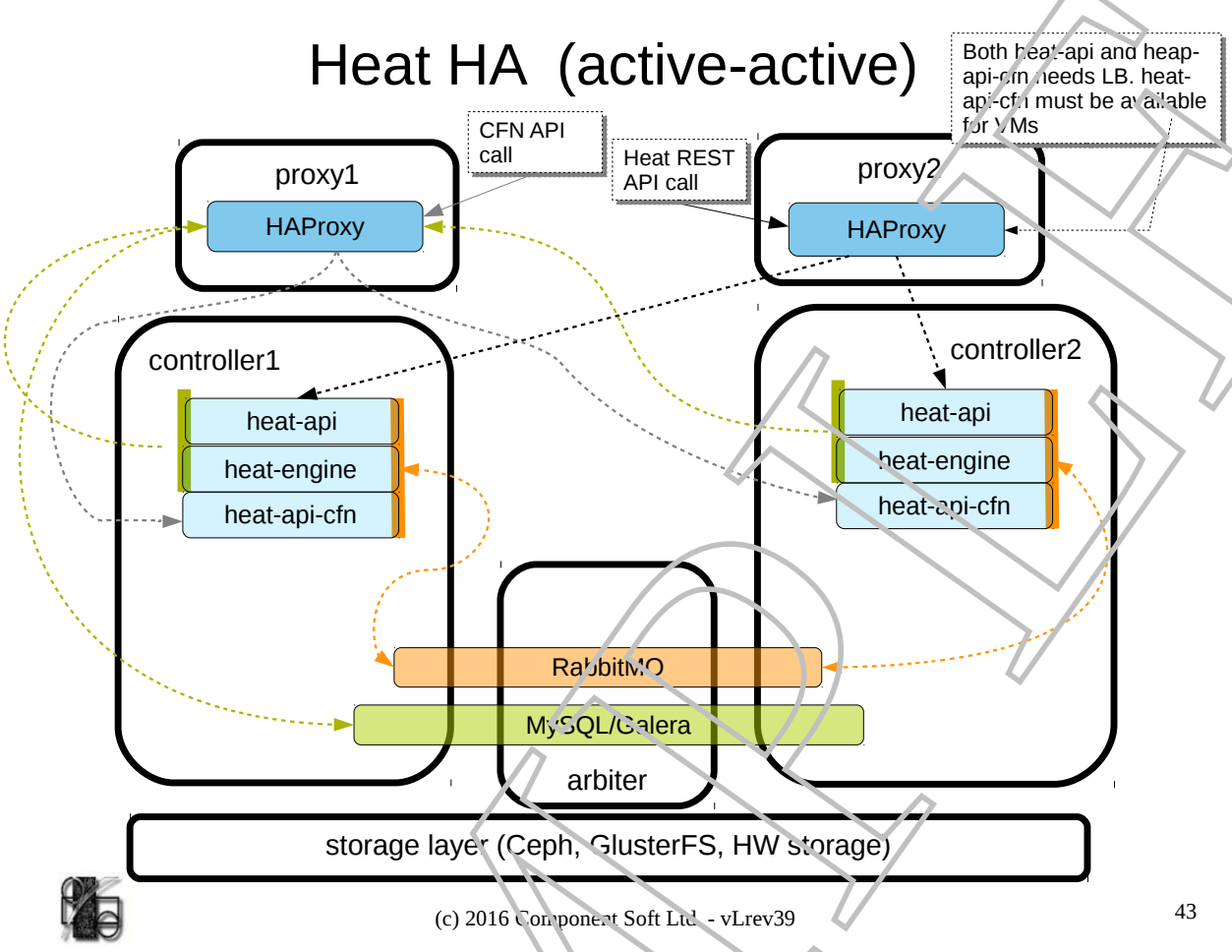
```
root@controller1 (admin) $> cinder list
//-----------+--------------+------+-------------+----------+-------------+
//   Status  | Display Name | Size | Volume Type | Bootable | Attached to |
//-----------+--------------+------+-------------+----------+-------------+
// available |     vol1     |  1   |      -      |  false   |             |
//-----------+--------------+------+-------------+----------+-------------+
root@controller1 (admin) $> cinder show vol1 |grep host
|         os-vol-host-attr:host         | controller1.openstack.local@lvm#lvm   |
```

Later on, only that `cinder-volume` process on that registered host is allowed to process requests regarding this volume. This means you loose your ability to change/attach/detach/delete this volume if the parent host is lost, even if the volume was put on a shared storage layer, like when you used Ceph with driver `cinder.volume.drivers.rbd.RBDDriver`.

As of February 2016, the best practice is to run `cinder-volume` as a fail-over service

- An arbitrary hostame should be registered to volumes.
- Process `cinder-volume` has to have the same configuration on each node (including this arbitrary hostname)
- We should run only one instance of `cinder-volume`
  - One can use either Pacemaker or Keepalived to keep this only instance alive cross the set of controller nodes.

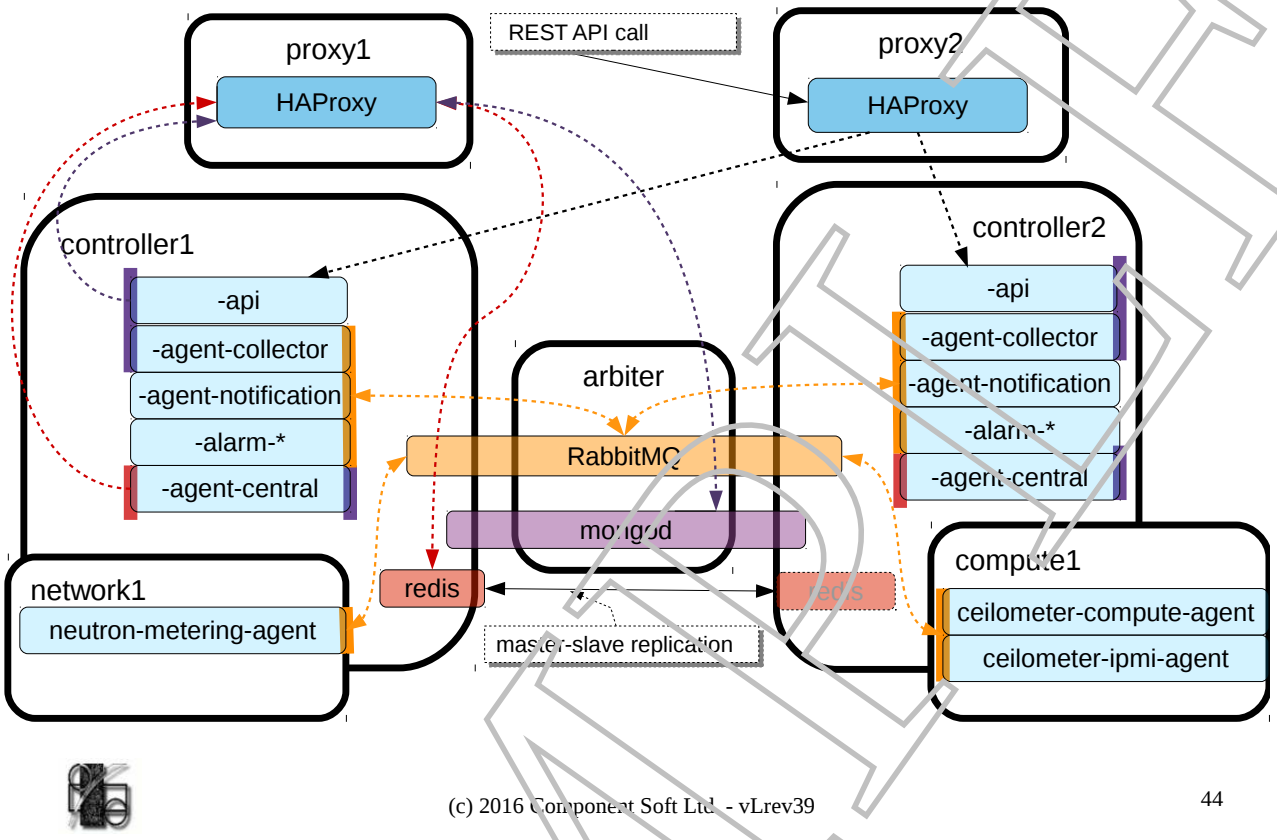
There is an ongoing development to solve this problem. The corresponding proposal suggests to replace local filesystem locks with Tooz library based ones. This change is scheduled to arrive to the Mitaka release.



As many other OpenStack services, Heat also has dependency on both RabbitMQ and MySQL.

It has two services needing a load-balancer.

- Connections to the API process (`heap-api`) should be balanced, and needs to be accessed from the public and from the internal networks as well.
- The `heat-api-cfn` process implements an endpoint for the *wait condition* feature (OS::Heat::WaitCondition resource). This endpoint should point to the internal interface of the LB, and it should forward the request to `heat-api-cfn` processes running on the controller nodes. A special request for this endpoint, that is should be accessible for processes running inside a VM, even if the VM does not hold a floating IP.



Ceilometer uses RabbitMQ extensively as a transport channel for samples, and as well for internal signaling.

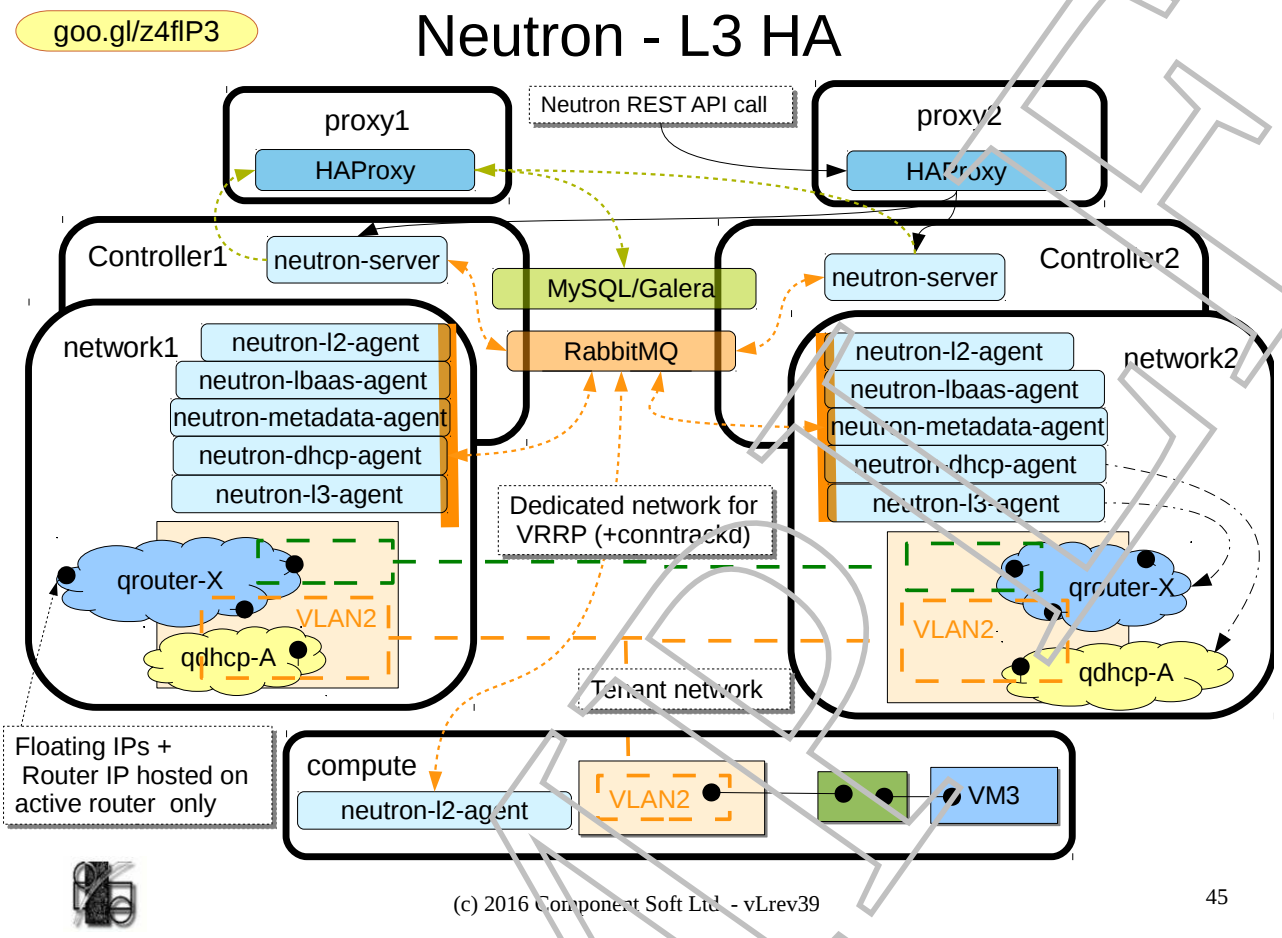
It uses MongoDB instead of MySQL to store both samples and internal data (like alarms). In order to protect Ceilometer data, one should set up a MongoDB replicaset. This replicaset needs to have at least 3 active nodes, but one of them can be an arbiter only. Arbiter nodes do not hold data, but vote in the primary node election process.

If you also need scalability, you can combine replicasets with the sharding feature of MongoDB. In this case, besides the data replicasets, one should set up a configuration replicaset, and a least two mongos instances, to route traffic to the mongod instances implementing the replicasets.

Setting up a MongoDB cluster is beyond the scope of this training.

Ceilometer also uses Redis as a backend for its coordination library Tooz. This is required by the `ceilometer-central-agent` to coordinate which resources and services are monitored by which agent instance.

To make `ceilometer-central-agent` highly available, one should also make Redis redundant, which can be achieved by Redis replication .



Presumably, service Neutron is the most complex of OpenStack services. It has a lot of moving parts distributed all over the OpenStack nodes.

Most of the components need AMQ server connection only. The only process with MySQL dependency is the API process `neutron-server`.

In Neutron, each process has its high availability solution. For some, just another process instance on a second node is enough to achieve a redundant service, but others need more complex setup.

- `neutron-dhcp-agent`

Neutron can configure many DHCP agents to support the same tenant network. This is controlled by the `dhcp_agents_per_network` parameter in `/etc/neutron/neutron.conf`. If `dhcp_agents_per_network` is set to `2`, two DHCP agents on two different network nodes has to set up the corresponding `qdhcp-` IP namespace, and has to run a `dnsmasq` process so serve the tenant network.

```
root@controller1 (admin) $> neutron net-list  |grep gre
| 4f268471-3d2f-4ae5-9fab-11c9e9d573a4| | int_gre1       >>
2ffb9ecb-7a03-4cfc-8c4f-a0329616cc03 10.30.30.0/24

root@controller1 (admin) $> for i in network{1,2};do
> echo "===$i==="; ssh $i ip netns |grep 4f268471;done
===network1===
qdhcp-4f268471-3d2f-4ae5-9fab-11c9e9d573a4
===network2===
qdhcp-4f268471-3d2f-4ae5-9fab-11c9e9d573a4
```

It is generally a bad idea to run multiple DHCP servers on the same network, but in this case, both servers have identical configuration, so they offer the same IP for each VM in the network.

- `neutron-l3-agent`

This daemon is responsible of `qrouter-` IP namespace, which implements a tenant router on the network node. Naturally, we has a `neutron-l3-agent` running in each and every network node, but this, however, does not guarantee high availability for the router service. To make a router prone to network node failures, one should create a *ha* router.

```
root@controller1 (admin) $> neutron router-create --ha=True int_gre1_router
root@controller1 (admin) $> neutron router-list
+--------------------------------------+-----------------//-------------+------+
| id                                   | name            // distributed | ha   |
+--------------------------------------+-----------------//-------------+------+
| acebe6db-b8ea-44a4-beb8-f5355097e03a | int_gre1_router // False       | True |
+--------------------------------------+-----------------//-------------+------+
```

Note:
The `ha` flag of the router cannot be updated.

For an *ha* router, a `qrouter-` namespace is created on a couple of network nodes.

```
root@controller1 (admin) $> for i in network{1,2};do
> echo "===$i==="; ssh $i ip netns |grep f0e8b170;done
===network1===
qrouter-acebe6db-b8ea-44a4-beb8-f5355097e03a
===network2===
qrouter-acebe6db-b8ea-44a4-beb8-f5355097e03a
```

The level of redundancy is controlled by a set of parameters in the `neutron.conf`. The below config example defines maximum 3 and minimum 2 namespace/router instances.

```
### /etc/neutron/neutron.conf ###
l3_ha = True
max_l3_agents_per_router = 3
```



```
min_l3_agents_per_router = 2
```

These routers, however, serve the network in an active/passive manner, so only one of them is allowed to configure the IP addresses of the served tenant network (gateway IP, floating IPs). The details are discussed on the next slide.

- `neutron-metadata-agent`

This daemon starts a `neutron-ns-metadata-proxy` process for each router hosted by the given network node. This process runs inside the router namespace, and serves configuration meta-data for the VMs.

```
root@network1 $> ps -ef |grep metadata-proxy
neutron   5742     1  0 15:16 ? 00:00:00
/usr/bin/python2 /bin/neutron-ns-metadata-proxy
--pid_file=/var/lib/neutron/external/pids/aceb//--//7e03a.pid
--metadata_proxy_socket=/var/lib/neutron/metadata_proxy
--router_id=acebe6db-b8ea-44a4-beb8-f5355097e03a
--state_path=/var/lib/neutron --metadata_port=9697
...
```

The metadata service is compatible with L3-HA: the metadata proxy is running in the active router namespace only.

```
root@controller1 (admin) $> for i in network{1,2};do echo "===$i==="; \
> ssh $i ip netns exec qrouter-acebe6db-b8ea-44a4-beb8-f5355097e03a ss -antp|\
> grep meta ;done
===network1===
LISTEN 0 128  *:9697 *:*   users:(("neutron-ns-meta",5742,6))
===network2===
```

- `neutron-lbaas-agent`

This agent is responsible implementing a hardware or software-based load-balancer. The default implementation uses HAProxy.

In case of `haproxy`, the load-balancer is running in a separated namespace `qlbaas-X`.

```
root@controller1 (admin) $> neutron lb-pool-list
+--------------------------------------+-----------------------+>>
| id                                   | name                  >>
+--------------------------------------+-----------------------+>>
| 36add235-50e2-485b-98ca-b30ddbe78f4c | auto-pool-wvxga2fcnb7n >>
+--------------------------------------+-----------------------+>>
+----------+-------------+----------+----------------+--------+
| provider | lb_method   | protocol | admin_state_up | status |
+----------+-------------+----------+----------------+--------+
| haproxy  | ROUND_ROBIN | HTTP     | True           | ACTIVE |
+----------+-------------+----------+----------------+--------+
```



```
root@controller1 (admin) $> neutron lb-agent-hosting-pool  \
> 36add235-50e2-485b-98ca-b30ddbe78f4c
//--------------------------+----------------+-------+
// host                     | admin_state_up | alive |
//--------------------------+----------------+-------+
// network2.openstack.local | True           | :-)   |
//--------------------------+----------------+-------+
```

```
root@network2 $> ip netns |grep qlbaas
qlbaas-36add235-50e2-485b-98ca-b30ddbe78f4c
```

The load-balancer is active on one network node only, it is not rescheduled to an other node in case of a network node failure. For example, if we shut down `network2`, the previously discussed load-balancer remains as failed on that node.

```
root@controller1 (admin) $> neutron lb-agent-hosting-pool \
> 36add235-50e2-485b-98ca-b30ddbe78f4c
//--------------------------+----------------+-------+
// host                     | admin_state_up | alive |
//--------------------------+----------------+-------+
// network2.openstack.local | True           | xxx   |
//--------------------------+----------------+-------+
```

Recently a new project called Octavia aims to replace the namespace haproxy driver as a default implementation. Project Octavia is accepted for the Liberty release, and promises to provide HA and better scalability than the original implementation.

L3 HA details

- Network nodes run keepalived in qrouter-X
  - keepalived talks VRRP on a separated network
    - one per tenant
  - VRRP election process
    - hello packet goes to multicast 224.0.0.18
    - One byte for VRID in VRRP headers
      only 255 virtual routers per tenant
    - 3 missed hellos start an election process
      the higher priority node wins
      a failback can done if the original active comes back
    - new active instance sends a gratuitous ARP request
      switches learn the new port for MAC/IP
- Incompatibility with l2population
- Other *aaS
  - HA for LBaaS is implemented in Liberty via Octavia (lbaasv2)
  - OS::Neutron::Router supports option 'ha' since Kilo

Let us see how L3 HA is implemented.

A `keepalived` daemon is running in each namespace, and those daemons elect a master via the VRRP protocol. Only the elected master is allowed to configure the IP addresses related to the tenant network (gateway address, floating IPs)

```
root@controller1 (admin) $> for i in network{1,2};do echo "###$i###";\
> ssh $i ip netns exec qrouter-acebe6db-b8ea-44a4-beb8-f5355097e03a ss -anpl|\
> grep keep;done
###network1###
p_raw  UNCONN     52480  0 ipv6:*          *      users:(("keepalived",5549,9))
p_raw  UNCONN     0      0 rarp:*          *      users:(("keepalived",5549,8))
u_dgr  UNCONN     0      0    * 44023838    * 7081  users:(("keepalived",5549,3),
                                                    ("keepalived",5548,3))
raw    UNCONN     0      0    *:112       *:*      users:(("keepalived",5549,11))
raw    UNCONN     0      0    *:112       *:*      users:(("keepalived",5549,10))
###network2###
p_raw  UNCONN     0      0 ipv6:*          *      users:(("keepalived",14002,9))
p_raw  UNCONN     0      0 rarp:*          *      users:(("keepalived",14002,8))
u_dgr  UNCONN     0      0    * 45347796    * 7081  users:(("keepalived",14002,3),
                                                    ("keepalived",14001,3))
raw    UNCONN     0      0    *:112       *:*      users:(("keepalived",14002,11))
```

```
raw    UNCONN     0      0     *:112      *:*      users:(("keepalived",14002,10))
```

```
root@controller1 (admin) $> for i in network{1,2};do echo "===$i==="; \
> ssh $i ip netns exec qrouter-acebe6db-b8ea-44a4-beb8-f5355097e03a \
> ip a s|grep -A2 ': [hq][rga]-';done
===network1===
15: ha-1f914259-39: <BROADCAST,MULTICAST,UP,LOWER_UP>
       mtu 1500 qdisc noqueue state UNKNOWN
    link/ether fa:16:3e:52:a0:44 brd ff:ff:ff:ff:ff:ff
    inet 169.254.192.10/18 brd 169.254.255.255 scope global ha-1f914259-39
--
17: qg-7dba9dc9-1c: <BROADCAST,MULTICAST,UP,LOWER_UP>
       mtu 1500 qdisc noqueue state UNKNOWN
    link/ether fa:16:3e:87:90:db brd ff:ff:ff:ff:ff:ff
    inet 192.168.100.100/24 scope global qg-7dba9dc9-1c
--
18: qr-ce25fad0-2f: <BROADCAST,MULTICAST,UP,LOWER_UP>
       mtu 1500 qdisc noqueue state UNKNOWN
    link/ether fa:16:3e:d1:2e:d3 brd ff:ff:ff:ff:ff:ff
    inet 10.30.30.254/24 scope global qr-ce25fad0-2f
===network2===
23: ha-dd65610b-4c: <BROADCAST,MULTICAST,UP,LOWER_UP>
       mtu 1500 qdisc noqueue state UNKNOWN
    link/ether fa:16:3e:5b:9e:22 brd ff:ff:ff:ff:ff:ff
    inet 169.254.192.11/18 brd 169.254.255.255 scope global ha-dd65610b-4c
--
24: qg-7dba9dc9-1c: <BROADCAST,MULTICAST,UP,LOWER_UP>
       mtu 1500 qdisc noqueue state UNKNOWN
    link/ether fa:16:3e:87:90:db brd ff:ff:ff:ff:ff:ff
25: qr-ce25fad0-2f: <BROADCAST,MULTICAST,UP,LOWER_UP>
       mtu 1500 qdisc noqueue state UNKNOWN
    link/ether fa:16:3e:d1:2e:d3 brd ff:ff:ff:ff:ff:ff
```

The `keepalived` daemons are connected to the `ha-XXX` interface, which is connected to a tenant network dedicated for the VRRP signaling.

```
root@controller1 (admin) $> neutron net-list

+--------------------------------------------------->>
| name                                               >>
+--------------------------------------------------->>
| int_gre1                                           >>
| HA network tenant eb77b585a9184304a0248537431c723f >>

+-----------------------------------------------------+
| subnets                                             |
+-----------------------------------------------------+
| dfd52314-d841-47b1-80e8-58dbdb8cf45a 10.30.30.0/24  |
| c1422ac0-14f1-4e19-9155-b17e1fb3ab79 169.254.192.0/18 |
```


Neutron - Distributed Virtual Routing

- Motivation
  - Too many network nodes
    - North-South traffic (SNAT/DNAT) has to go through the a network node
      - We might want to utilize the Compute for DNAT
      - SNAT is still done by the Network node
    - East-West traffic (between two tenant networks)
      - Router/firewall can forward between private nets
        VM1 access VM2 via a floating IP
      - We want direct Compute-Compute communication
- Implementation
  - compute needs access to the external net (br-ex)
  - l3-agent on compute
    - agent-mode=dvr-snat on Network
    - agent-mode=dvr on Compute
    - new namespaces (snat-X on Network, fip-X, qrouter-X on Compute)

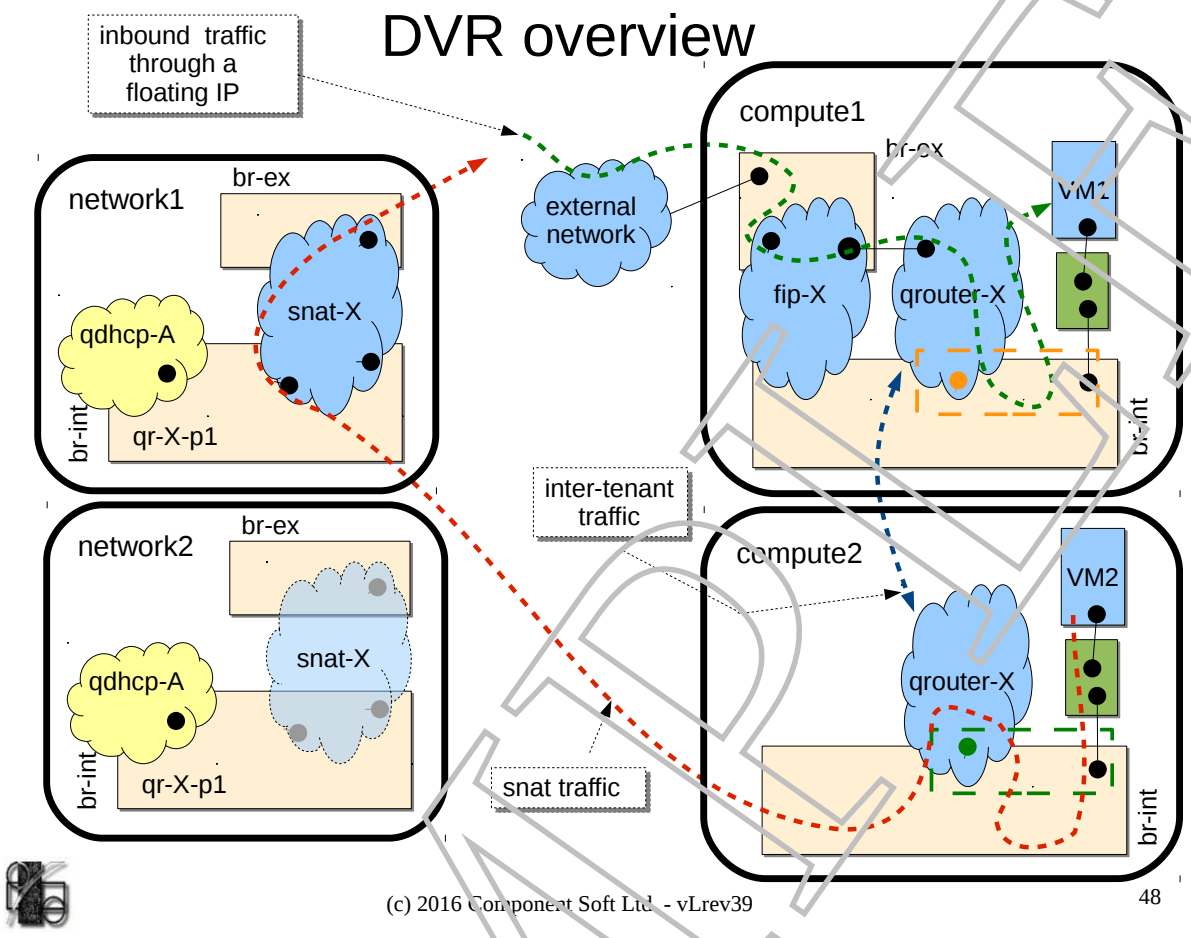
An other way to solve the scalability/redundancy issues of Neutron routers is to use the Distributed Virtual Router (DVR) feature.

The original solution with network nodes has scalability problems

- All traffic to/from the external network (North - South traffic) has to pass through the network node. When most of VMs has floating IPs, and when there is a high bandwidth demand towards the external network, in worst case, we might need as many network nodes as many compute nodes we have.
- Traffic between tenant networks (East - West traffic) has also pass through the network node, increasing the demand for additional network nodes.

Option DVR solves the problem by making compute nodes implement tenant routers and host floating IPs. However, we still need a network nodes:

- DHCP service for tenant networks is still hosted by network nodes
- The network nodes provide external connectivity for VM without a floating IP.



The above slide gives an overview of the DVR feature.

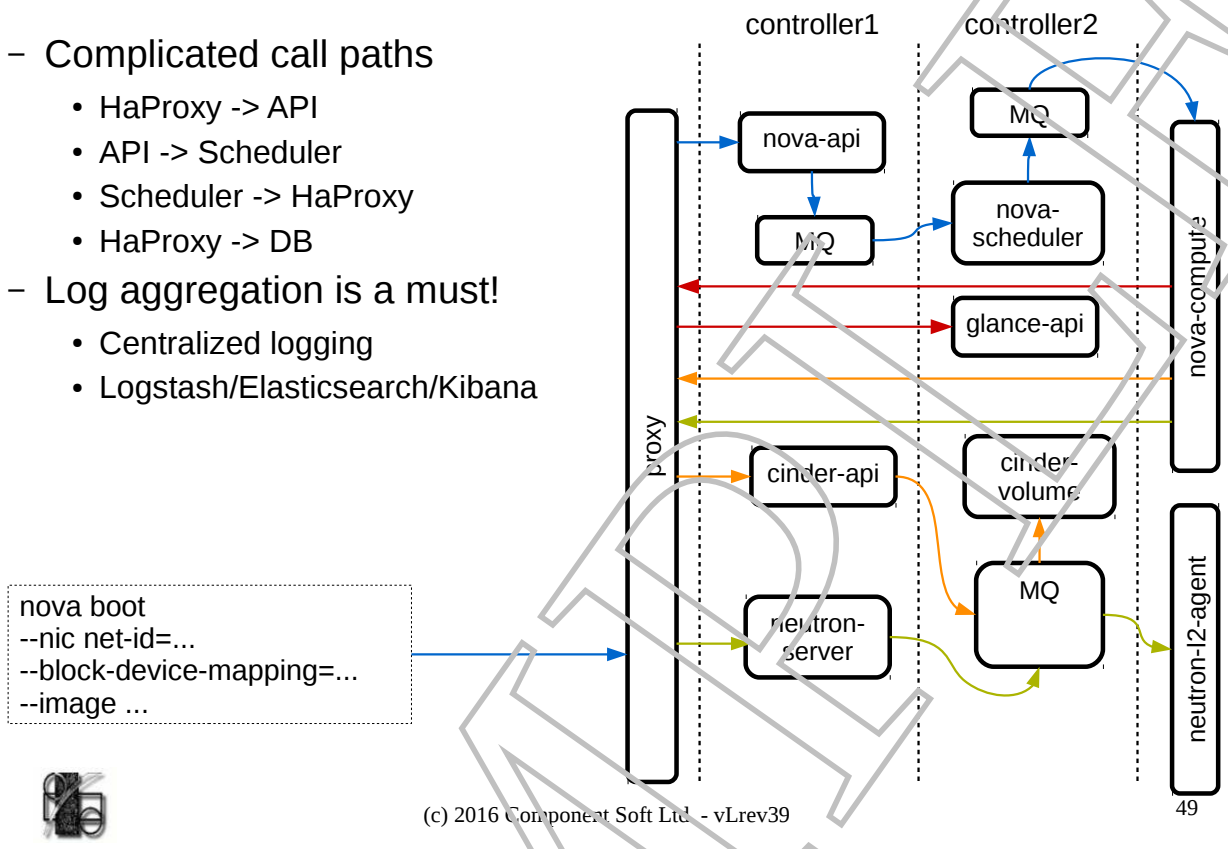
In order to make the DVR work, one should change the configuration of nodes

- On **compute nodes**, one should enable `l3-agent` with `agent-mode=dvr` in its `l3-agent.ini`.
- The **compute nodes** need external connectivity.
- On **network nodes**, one should run `l3-agent` with `agent-mode=dvr-snat` in its `l3-agent.ini`.
- Tenant routers have to be created with flag `--distributed=True`.

Each compute node creates a `qrouter` namespace, and setup up complex policy based routing rules to direct East/West traffic towards another compute node.

Compute nodes hosting a VM with a floating IP also create a `fip` namespace.

Traffic from VM without a floating IP is routed towards the network nodes, where the packet source address is translated to the external IP of the `snat` namespace of the network node.



The last part of this chapter describes a centralized logging solution for OpenStack.

Even for a medium sized deployment of OpenStack (3 controller nodes, 2 network nodes, ~10 compute nodes) it is a challenge to find the root of a problem from log files. For instance, when starting a simple VM, a lot of processes on many nodes are involved in the provisioning process.

- The request is processed by one `nova-api` process on one of the controller nodes. This process forwards the request to the message queue, but its message broker might run on a different node.
- The message is processed by a `nova-scheduler` on one of the controller nodes, and then forwarded to a `nova-compute` process on one of the compute nodes.
- The `nova-compute` then makes REST calls to Glance,Cinder and Neutron to gather the necessary resources for the VM. These REST calls are handled by API processes on different controller nodes.

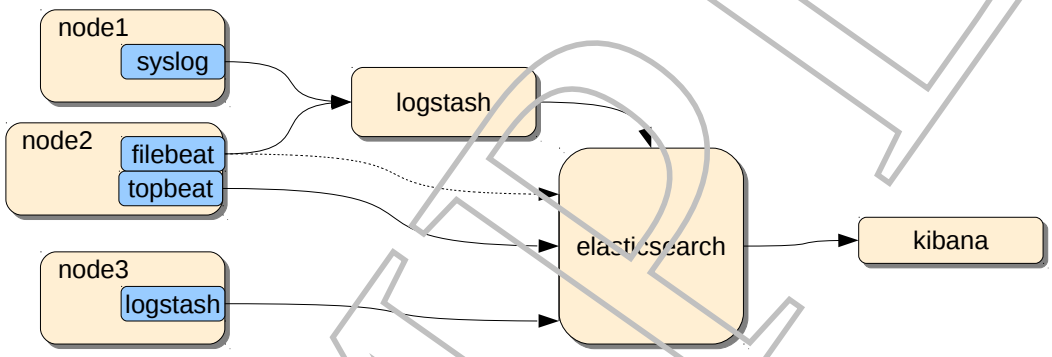
To follow the provisioning process we might need to check all log files in all OpenStack nodes. For the proper maintenance of OpenStack we must setup up centralized logging. This can be achieved by

- Centralized syslog server:
  - This case all OpenStack daemons are configured to send logs into syslog instead of plain files.
  - Local syslog servers (like rsyslogd) can forward these log entries to a centralized log server.
  - The centralized log server can direct messages from one server/daemon to a separate directories/files.
- Elasticsearch, Kibana, Logstash:

  In this case logs are processed by Logstash, stored in the search engine ElasticSearch, and finally logs can be retrieved by the Kibana web UI.

Deployment scenarios for LS/ES/Kibana

- Filebeat/Syslog > LogStash -> ES -> Kibana
  - Filebeat is quite small (11MB), and efficient
  - Limited local processing
    - static tags, fields
- LogStash on nodes -> ES -> Kibana
  - Full power of LogStash for local log files
  - Big footprint (130MB + 100M java) for each node



The ELK stack can be used in different ways:

- Log can be sent directly into ElasticSearch .

  In this case the local agent `filebeat` is installed on each node. Filebeat is configured to check changes to the related OpenStack log files, and new lines can be sent to ES. This solution, however, has a problem. We cannot do pre-processing on log entries, like

  – Drop line matching regular expressions

  – Merge lines representing a single event to one message

- Use Logstash to process log lines first and the results are sent to ES

  This solve the above problem, but installing LogStash on every node might be an overkill. It is a better solution, to make Filebeat (or configure syslog) to send log lines to LogStash first, than logs are filtered/merged and sent to ElasticSearch.

The final visualization or search on log entries can be done by the Kibana GUI .

Logstash configuration example

logstash.conf

```
input { beats { port => 5044 }  }
filter {
    multiline { negate => false
      pattern => "^%{TIMESTAMP_ISO8601}\s+\d+\s+(TRACE|ERROR)"
      what => "previous"
      stream_identity => "%{beat.hostname}.%{source}"
    }
    grok {
      match => { "message" => "(?m)^%{TIMESTAMP_ISO8601:date}\s+\d+\s+
     (?<loglevel>AUDIT|CRITICAL|DEBUG|INFO|TRACE|WARNING|ERROR)\s(?<module>\S+).*$" }
    }
    if [module] == "iso8601.iso8601" { drop {} }
}
output {
        elasticsearch {
          hosts => "10.10.10.110:9200"
          index => "%{[@metadata][beat]}-%{+YYYY.MM.dd}"
          document_type => "%{[@metadata][type]}"
        }
    }
```

This is an example configuration for LogStash.

- Messages arriving from FileBeat on port `5044`.
- First we try to join lines matching the same event into a single record.
- Second, we extract additional information from records (Oslo message format)
  - The timestamp of the record
  - The loglevel
  - The Python module the log entry is originated from.
- Finally we send the message to ElasticSearch
  - A new index is opened each day. Later we could drop old indexes via the Curator.

Lab2

- Initial health check
- Configure central logging
  - Start the monitor node
    - with logstash/ES/Kibana preinstalled
  - Install filebeat, topbeat on nodes.