Mastering OpenStack

This comprehensive guide will help you to choose the right practical option and make strategic decisions about the OpenStack cloud environment to fit your infrastructure in production.

At the start, this book will explain the OpenStack core architecture. You will soon be shown how to create your own OpenStack private cloud.

Next, you will move on to cover the key security layer and network troubleshooting skills, along with some advanced networking features. Finally, you will gain experience of centralizing and logging OpenStack. The book will show you how to carry out performance tuning based on OpenStack service logs.

By the end of this book, you will be ready to take steps to deploy and manage an OpenStack cloud with the latest open source technologies.

Who this book is written for
This book is intended for system administrators, cloud engineers, and system architects who want to deploy a cloud based on OpenStack in a mid- to large-sized IT infrastructure. If you have a fundamental understanding of cloud computing and OpenStack and want to expand your knowledge, then this book is an excellent checkpoint to move forward.

What you will learn from this book
- Explore the main architecture design of OpenStack components, core-by-core services, and how they work together
- Learn how to distribute OpenStack services among cluster setup
- Compare different storage solutions and driver extensions
- Design different high availability scenarios and how to plan for a no Single Point Of Failure environment
- Set up a multinode environment in production using orchestration tools
- Boost OpenStack performance with advanced configuration
- Establish a distributed monitoring solution and keep track of resource consumption

$ 49.99 US
£ 31.99 UK
Prices do not include local sales tax or VAT

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 6 'OpenStack HA and Failover'
- A synopsis of the book’s content
- More information on Mastering OpenStack
Omar Khedher is a network engineer and cloud computing researcher. Based in the Netherlands, he has worked in a cloud computing solution project that turned into an OpenStack deployment and became integrated with it. Leveraging his skills as a system administrator in virtualization, storage, and networking, he is currently pursuing a PhD on performance research preparation in the cloud computing paradigm and architecture patterns in which OpenStack is taking an active part. He has recently authored a few academic publications based on new researches for the cloud performance improvement.
Since its first official release in 2010, OpenStack has distinguished itself as the ultimate open source cloud operating system. Today, more than 200 companies worldwide have joined the development of the OpenStack project, which makes it an attractive cloud computing solution for thousands of organizations. The main reason behind the success of OpenStack is not the overwhelming number of features that it has implemented, but rather its good modularity. Thanks to its vast community around the world, OpenStack is growing very fast. Each release exposes new modules and administrative facilities that offer on-demand computing resources by provisioning a large set of networks of virtual machines. If you are looking for a cloud computing solution that scales out well, OpenStack is an ideal option. Nowadays, it is considered to be a mature cloud computing operating system. Several big, medium, and small enterprises have adopted it as a solution in their infrastructure. The nirvana of OpenStack comes from its architecture. Designing your cloud becomes much easier with more flexibility. It is an ideal solution if you intend either to design a start up cloud environment or to integrate it into your existing infrastructure. As you build your cloud using OpenStack, you will be able to integrate with legacy systems and third-party technologies by eliminating vendor lock-in as much as possible.

This book is designed to discuss what is new in OpenStack with regards to the new features and incubated projects. You will be guided through this book from design to deployment and implementation with the help of a set of best practices in every phase. Each topic is elaborated so that you can see the big and complete picture of a true production environment that runs OpenStack at scale. It will help you decide upon the ways of deploying OpenStack by determining the best outfit for your private cloud, such as the computer, storage, and network components.
If you are ready to start a real private cloud running OpenStack, master the OpenStack design, and deploy and manage a scalable OpenStack infrastructure, this book will prove to be a clear guide that exposes the latest features of the OpenStack technology and helps you leverage its power to design and manage any medium or large OpenStack infrastructure.

**What this book covers**

*Chapter 1, Designing OpenStack Cloud Architecture*, will focus on discussing the several components of the architecture of OpenStack. It will provide the basis that is needed to start with the first design of your OpenStack private cloud environment. The chapter will discuss the different models’ designs, which will help you begin your first deployment of OpenStack from scratch. The chapter will contain practical examples and calculations that are framed in a theoretical approach to give you an idea about how you can choose the right hardware capacity for your first OpenStack environment and adapt such information to real-world deployments.

*Chapter 2, Deploying OpenStack – DevOps and OpenStack Dual Deal*, will introduce you to the first installation of the OpenStack environment using automation tools. You will learn how to get the entire infrastructure installed and customized using Chef. The chapter will highlight the adoption of the DevOps approach and cover several advantages of how you can conduct your first OpenStack deployment from a test to production environment with more flexibility. It will provide instructions on how to install and use the Chef cookbooks to install the first test environment and get ready for the production stage.

*Chapter 3, Learning OpenStack Clustering – Cloud Controllers and Compute Nodes*, will decompose the big parts of your deployment by further refining your design, which was elaborated on in the previous chapter. It will cover some best practices regarding the art of clustering. Next, you will learn how to distribute the main OpenStack services between the cloud controllers and the compute nodes and construct an efficient OpenStack cluster. It will put under the microscope the choice of the hypervisor and hardware specifications. A sample design of the Chef cookbooks will be implemented to help you learn how to automate a cloud controller and install the compute nodes. The chapter will also explore how to plan the backup of an OpenStack cluster.
Chapter 4, Learning OpenStack Storage – Deploying the Hybrid Storage Model, will cover the subject of storage in OpenStack. The chapter will start by focusing on the storage types and their use cases. You will learn about an object storage code named Swift and how it works in OpenStack. A real Swift deployment will be shown to help you calculate the hardware requirements. The chapter will also talk about the block storage code named Cinder in OpenStack. You will learn how to decide which storage type will fulfill your needs. It will also explore Ceph and its main architectural design. It will help you integrate it and install in your test OpenStack environment using Vagrant and Chef.

Chapter 5, Implementing OpenStack Networking and Security, will focus mainly on the networking security features in OpenStack. It will cover the concept of namespaces and security groups in OpenStack and how you can manage them using the Neutron and Nova APIs. In addition, it will explore the new networking security feature, Firewall as a Service. A case study will help you understand another networking feature in Neutron called VPN as a Service.

Chapter 6, OpenStack HA and Failover, will cover the topics of high availability and failover. For each component of the OpenStack infrastructure, this chapter will expose several HA options. The chapter will be replete with HA concepts and best practices, which will help you define the best HA OpenStack environment. It serves as a good complementary chapter for the previous chapters by bringing a geared, distributed, and fault-tolerant OpenStack architecture design. Numerous open source solutions, such as HAProxy, Keepalived, Pacemaker, and Corosync, will be discussed through a step-by-step instruction guide.

Chapter 7, OpenStack Multinode Deployment – Bringing in Production, will be your “first production day” guide. It will focus on how you can deploy a complete multinode OpenStack setup. A sample setup will be explained and described in detail by exposing the different nodes and their roles, the network topology, and the deployment approach. The chapter will contain a practical guide to OpenStack deployment using bare metal provision tools xCAT together with the Chef server. It will demonstrate the first run of a new OpenStack tenant.

Chapter 8, Extending OpenStack – Advanced Networking Features and Deploying Multi-tier Applications, will delve into the advanced OpenStack networking features. It will explain in depth the Neutron plugins such as Linux Bridge and Open vSwitch, how they differ from the architectural perspective, and how instances can be connected to networks with the Neutron plugins. The chapter will also cover Load Balancing as a Service, which is used to load balance the traffic between instances by exploring their fundamental components. In addition, an orchestration module named Heat will be introduced in this chapter and will be used to build a complete stack to show how a real load balancer is deployed in OpenStack.
Chapter 9, Monitoring OpenStack – Ceilometer and Zabbix, will explore another new incubated project called Ceilometer as a new telemetry module for OpenStack. The chapter will discuss briefly the architecture of Ceilometer and how you can install and integrate it into the existing OpenStack environment. The discussion on Heat will be resumed, and it will be used to expand a stack installation including Ceilometer. The purpose of this is to discover the capabilities of heat with regard to supporting the Ceilometer functions, such as alarms and notifications. This section will also make sure that the OpenStack environment is well-monitored using some external monitoring tools such as Zabbix for advanced triggering capabilities.

Chapter 10, Keeping Track for Logs – Centralizing Logs with Logstash, will talk about the problem of logging in OpenStack. The chapter will present a very sophisticated logging solution called Logstash. It will go beyond the tailing and grepping of single log lines to tackle complex log filtering. The chapter will provide instructions on how to install Logstash and forward the OpenStack log files to a central logging server. Furthermore, a few snippets will be be provided to demonstrate the transformation of the OpenStack data logs and events into elegant graphs that are easy to understand.

Chapter 11, Tuning OpenStack Performance – Advanced Configuration, will wrap things up by talking about how you can make the OpenStack infrastructure run better with respect to its performance. Different topics, such as the advanced configuration in the exiting OpenStack environment, will be discussed. The chapter will put under the microscope the performance enhancement of MySQL by means of hardware upgrade and software layering such as memcached. You will learn how to tune the OpenStack infrastructure component-by-component using a new incubated OpenStack project called Rally.
OpenStack HA and Failover

"Once we accept our limits, we go beyond them."

–Albert Einstein

So far, you have gained a good knowledge of all the components needed to provide a functional OpenStack infrastructure. In Chapter 1, Designing OpenStack Cloud Architecture, we saw one of the many ways to design a complete OpenStack environment. Chapter 3, Learning OpenStack Clustering – Cloud Controllers and Compute Nodes, looked at one of the most important logical and physical designs of OpenStack clustering in depth by iterating through cloud controller and compute nodes. Distributing services through the mentioned nodes after considering the standalone storage cluster, as seen in Chapter 4, Learning OpenStack Storage – Deploying the Hybrid Storage Model, aims to reduce the downtime for a given service. Many design approaches can fulfill such high-availability goals in OpenStack. On the other hand, HA may not be as simple as the name suggests: it's the effort to eliminate any Single Point Of Failure (SPOF) on every layer in your architecture. OpenStack components can be brought and distributed in different nodes while maintaining a sense of teamwork, which OpenStack is good at—again, thanks to our messaging service. In this chapter, we will:

• Understand how HA and failover mechanisms can guarantee OpenStack business continuity
• Look for a workaround on how to make different OpenStack components configured in HA
• Check out different ways to validate a complete HA setup
HA under the scope

On a daily basis, system and network administrators are faced with a new challenge by hitting the same point: *we are aiming to make our infrastructure highly available!*

Meanwhile, the IT manager sticks to his chair, drinking his tea and claims: *our IT system works just fine and our customers are satisfied.* Surprisingly, you get that phone call from the help desk with a struggling voice: *well, the browser said "page not found". Is the application server down?* Obviously, the infrastructure was not as highly available as it should have been. Despite your extra time spent configuring clusters to be in uptime, more often than not, servers might not be reachable and you then face a few special events, and you raise this question: *why does it not fail over?* To make sense of an HA infrastructure, on one hand, you should know what HA offers to your environment and how. On the other hand, you should stay close to test scenarios of failing over as exemplified in the following real-life show. Many system administrators feel lucky when they have bought a storage box that is not supposed to fail, and even has this written: *the solution that never shouts I am offline.* They claim that the new NAS box is highly available. Sadly, this is never realized. A power outage takes place and it takes the fancy cluster out of service for a few hours so that it can be restarted. If you realized that you need an extra battery, then you can prevent this physical event failure. Later, you update its software package by clicking on *Update the NAS.* Unfortunately, the developers of the NAS appliance have included a new feature in its HA package that makes the software unstable, but you are not able to not figure that out, as it is a new release and nobody had complained about it previously. After a while, a failover happens but the server is unreachable. It should have worked as intended. But in vain, by checking in the racks, you figured out that eventually, the slave node is becoming the master according to the shining LED light, which gets stuck while blinking! The failover is on its way, but the system is not responsive. There was a software bug in the last release. At this point, the downtime increases again while the bug waits to be fixed. Unluckily, you were the first NAS box client to complain about the new features, which you might have to wait to fix. This might take some time. A real-long unplanned failure could lead to a bigger problem!

The storage system is not highly available anymore. Downtime is the exact *enemy* of HA. Friendly downtime can be planned as you will only need to replace some pieces of hardware. On the other hand, there are many reasons for unexpected downtime, such as problems with hardware and software, or any external condition that leads to the failure of the system.
Do not mix them

We still remember that one of the several purposes of OpenStack clustering is to make sure that services remain running in the case of a node failure. The HA functionality aims to make sure that the different nodes participating in a given cluster work in tandem to satisfy certain downtime. HA, in fact, is a golden goal for any organization where some useful concepts can be used to reach it with minimum downtime, such as the following:

- **Failover**: Migrate a service running on the failed node to a working one (switch between primary and secondary)
- **Fallback**: Once a primary is back after a failed event, the service can be migrated back from the secondary
- **Switchover**: Manually switch between nodes to run the required service

On the other side, we may find a different terminology, which you may have most likely already experienced, that is, **load balancing**. In a heavily loaded environment, load balancers are introduced to redistribute a bunch of requests to less loaded servers. This can be similar to the **high performance clustering** concept, but you should note that this cluster logic takes care of working on the same request, whereas a load balancer aims to relatively distribute the load based on its task handler in an optimal way.

HA levels in OpenStack

It might be important to understand the context of HA deployments in OpenStack. This makes it imperative to distinguish the different levels of HA in order to consider the following in the cloud environment:

- **L1**: This includes physical hosts, network and storage devices, and hypervisors
- **L2**: This includes OpenStack services, including compute, network, and storage controllers, as well as databases and message queuing systems
- **L3**: This includes the virtual machines running on hosts that are managed by OpenStack services
- **L4**: This includes applications running in the virtual machines themselves
OpenStack HA and Failover

The main focus of the supporting HA in OpenStack has been on L1 and L2, which are covered in this chapter. On the other hand, L3 HA has limited support in the OpenStack community. By virtue of its multistorage backend support, OpenStack is able to bring instances online in the case of host failure by means of live migration. Nova also supports the Nova evacuate implementation, which fires up API calls for VM evacuation to a different host due to a compute node failure. The Nova evacuate command is still limited as it does not provide an automatic way of instance failover. L2 and L3 HA are considered beyond the scope of this book. L4 HA is touched on, and enhanced by, the community in the Havana release. Basically, a few incubated projects in OpenStack, such as Heat, Savana, and Trove, have begun to cover HA and monitoring gaps in the application level. Heat will be introduced in Chapter 8, Extending OpenStack – Advanced Networking Features and Deploying Multi-tier Applications, while Savana and Trove are beyond the scope of this book.

Live migration is the ability to move running instances from one host to another with, ideally, no service downtime. By default, live migration in OpenStack requires a shared filesystem, such as a Network File System (NFS). It also supports block live migration when virtual disks can be copied over TCP without the need for a shared filesystem. Read more on VM migration support within the last OpenStack release at http://docs.openstack.org/admin-guide-cloud/content/section_configuring-compute-migrations.html.

A strict service-level agreement

Normally, if you plan to invest time and money in OpenStack clustering, you should refer to the HA architectural approaches in the first place. They guarantee business continuity and service reliability.

At this point, meeting these challenges will drive you to acquire skills you never thought you could master. Moreover, exposing an infrastructure that accepts failures might distinguish your environment as a blockbuster private cloud. Remember that this topic is very important in that all you have built within OpenStack components must be available to your end user.
Availability means that not only is a service running, but it *is also exposed and able to be consumed*. Let's see a small overview regarding the maximum downtime by looking at the availability percentage or *HA as X-nines*:

<table>
<thead>
<tr>
<th>Availability level</th>
<th>Availability percentage</th>
<th>Downtime/year</th>
<th>Downtime/day</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Nine</td>
<td>90</td>
<td>~36.5 days</td>
<td>~2.4 hours</td>
</tr>
<tr>
<td>2 Nines</td>
<td>99</td>
<td>~3.65 days</td>
<td>~14 minutes</td>
</tr>
<tr>
<td>3 Nines</td>
<td>99.9</td>
<td>~8.76 hours</td>
<td>~86 seconds</td>
</tr>
<tr>
<td>4 Nines</td>
<td>99.99</td>
<td>~52.6 minutes</td>
<td>~8.6 seconds</td>
</tr>
<tr>
<td>5 Nines</td>
<td>99.999</td>
<td>~5.25 minutes</td>
<td>~0.86 seconds</td>
</tr>
<tr>
<td>6 Nines</td>
<td>99.9999</td>
<td>~31.5 seconds</td>
<td>~0.0086 seconds</td>
</tr>
</tbody>
</table>

Basically, availability management is a part of IT best practices when it comes to making sure that IT services are running when needed, which reflects your *service-level agreement (SLA)*:

- Minimized downtime and data loss
- User satisfaction
- No repeat incidents
- Services must be consistently accessible

A paradox may appear between the lines when we consider that eliminating the SPOF in a given OpenStack environment will include the addition of more hardware to join the cluster. At this point, you might be exposed to creating more SPOF and, even worse, complicated infrastructure where maintenance turns into a difficult task.

**Measuring HA**

The following is a simple tip:

If you do not measure something, you cannot manage it. But what kind of metrics can be measured in a highly available OpenStack infrastructure?

Agreed, HA techniques come across as increasing the availability of resources, but still, there are always reasons you may face an interruption at some point! You may notice that the previous table did not mention any value equal to 100 percent uptime.
First, you may appreciate the nonvendor lock-in hallmark that OpenStack offers on this topic. Basically, you should mark the differences between HA functionalities that exist in a virtual infrastructure. Several HA solutions provide protection to virtual machines when there is a sudden failure in the host machine. Then, it will perform a restore situation for the instance on a different host. What about the virtual machine itself? Does it hang? So far, we have seen different levels of HA. In OpenStack, we have already seen cloud controllers run manageable services and compute hosts, which can be any hypervisor engine and third-rank the instance itself!

The last level might not be a cloud administrator task that maximizes its internal services' availability as it belongs to the end user. However, what should be taken into consideration, is what really affects the instance externally, such as:

- Storage attachment
- Bonded network devices

A good practice is to design the architecture with an approach that is as simple as possible by keeping efficient track of every HA level in our OpenStack cluster.

Eliminating any SPOF while designing the OpenStack infrastructure would help in reaching a scalable environment.

A good strategy to follow is to design an untrustworthy SPOF principle by ruling. This keyword can be found anywhere in any system. In Chapter 1, Designing OpenStack Cloud Architecture, within our first design, we highlighted a simple architecture that brings in many instances in order to maximize availability. Nowadays, large IT infrastructures are likely to suffer from database scalability across multiple nodes. Without exception, the database in the OpenStack environment will need to scale as well. We will cover how to implement a database HA solution in more detail later in this chapter.

High availability in OpenStack does not necessarily mean that it is designed to achieve maximum performance. On the other hand, you should consider the limitations of the overhead result on updating different nodes running the same service.
The HA dictionary

To ease the following sections of this chapter, it might be necessary to remember a few terminologies to justify high availability and failover decisions later:

- **Stateless service**: This is the service that does not require any record of the previous request. Basically, each interaction request will be handled based on the information that comes with it. In other words, there is no dependency between requests where data, for example, does not need any replication. If a request fails, it can be performed on a different server.

- **Stateful service**: This is the service where request dependencies come into play. Any request will depend on the results of the previous and the subsequent ones. Stateful services are difficult to manage, and they need to be synchronized in order to preserve consistency.

Let's apply our former definition to our OpenStack services:

<table>
<thead>
<tr>
<th>Stateful services</th>
<th>Stateless services</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL, RabbitMQ</td>
<td>nova-api, nova-conductor, glance-api, keystone-api, neutron-api, nova-scheduler, and web server (Apache/nginx)</td>
</tr>
</tbody>
</table>

Any HA architecture introduces an "active/active" or "active/passive" deployment, as covered in *Chapter 1, Designing OpenStack Cloud Architecture*. This is where your OpenStack environment will highlight its scalability level.

First, let's see the difference between both concepts in a nutshell in order to justify your decision:

- **Active/active**: Basically, all OpenStack nodes running the same stateful service will have an identical state. For example, deploying a MySQL cluster in the active/active mode will bring in a multimaster MySQL node design, which involves any update to one instance that may be propagated to all other nodes. Regarding the stateless services, redundancy will invoke instances to be load-balanced.

- **Active/passive**: In the case of stateful services, a failure event in one node will bring its associated redundant instance online. For example, within database clustering, only one master node comes into play, where the secondary node will act as a listener when failover occurs. It keeps load balancing handling requests within stateless services.
Hands on HA

Chapter 1, Designing OpenStack Cloud Architecture, provided a few hints on how to prepare for the first design steps: do not lock keys inside your car. At this point, we can go further due to the emerging different topologies, and it is up to you to decide what will fit best. The first question that may come into your mind: OpenStack does not include native HA components; how you can include them? There are widely used solutions for each component that we cited in the previous chapter in a nutshell.

Understanding HAProxy

HAProxy stands for High Availability Proxy. It is a free load balancing software tool that aims to proxy and direct requests to the most available nodes based on TCP/HTTP traffic. This includes a load balancer feature that can be a frontend server. At this point, we find two different servers within an HAProxy setup:

- A frontend server listens for requests coming on a specific IP and port, and determines where the connection or request should be forwarded
- A backend server defines a different set of servers in the cluster receiving the forwarded requests

Basically, HAProxy defines two different load balancing modes:

- **Load balancing layer 4**: Load balancing is performed in the transport layer in the OSI model. All the user traffic will be forwarded based on a specific IP address and port to the backend servers.
  
  For example, a load balancer might forward the internal OpenStack system's request to the Horizon web backend group of backend servers. To do this, whichever backend Horizon is selected should respond to the request under scope. This is true in the case of all the servers in the web backend serving identical content. The previous example illustrates the connection of the set servers to a single database. In our case, all services will reach the same database cluster.

- **Load balancing layer 7**: The application layer will be used for load balancing. This is a good way to load balance network traffic. Simply put, this mode allows you to forward requests to different backend servers based on the content of the request itself.
Many load balancing algorithms are introduced within the HAProxy setup. This is the job of the algorithm, which determines the server in the backend that should be selected to acquire the load. Some of them are as follows:

- **Round robin**: Here, each server is exploited in turn. As a simple HAProxy setup, round robin is a dynamic algorithm that defines the server's weight and adjusts it on the fly when the called instance hangs or starts slowly.

- **Leastconn**: The selection of the server is based on the lucky node that has the lowest number of connections.

  It is highly recommended that you use the leastconn algorithm in the case of long HTTP sessions.

- **Source**: This algorithm ensures that the request will be forwarded to the same server based on a hash of the source IP as long as the server is still up.

  Contrary to RR and leastconn, the source algorithm is considered a static algorithm, which presumes that any change to the server's weight on the fly does not have any effect on processing the load.

- **URI**: This ensures that the request will be forwarded to the same server based on its URI. It is ideal to increase the cache-hit rate in the case of proxy caches' implementations.

  Like the source, the URI algorithm is static in that updating the server's weight on the fly will not have any effect on processing the load.

You may wonder how the previous algorithms determine which servers in OpenStack should be selected. Eventually, the hallmark of HAProxy is a healthy check of the server's availability. HAProxy uses health check by automatically disabling any backend server that is not listening on a particular IP address and port.

But how does HAProxy handle connections? To answer this question, you should refer to the first logical design in *Chapter 1, Designing OpenStack Cloud Architecture*, which is created with virtual IP (VIP). Let's refresh our memory about the things that we can see there by treating a few use cases within a VIP.
**Services should not fail**

A VIP can be assigned to the active servers running all the OpenStack services that need to be configured to use the address of the server. For example, in the case of a failover of the nova-api service in controller node 1, the IP address will follow the nova-api in controller node 2, and all clients' requests, which are the internal system requests in our case, will continue to work:

---

**The load balancer should not fail**

The previous use case assumes that the load balancer never fails! But in reality, this is an SPOF that we have to arm by adding a VIP on top of the load balancer's set. Usually, we need a stateless load balancer in OpenStack services. Thus, we can undertake such challenges using software similar to Keepalived:
Keepalived is a free software tool that provides high availability and load balancing facilities based on its framework in order to check a Linux Virtual Server (LVS) pool state.

LVS is a highly available server built on a cluster of real servers by running a load balancer on the Linux operating system. It is mostly used to build scalable web, mail, and FTP services.

As shown in the previous illustration, nothing is magic! Keepalived uses the Virtual Router Redundancy Protocol (VRRP) protocol to eliminate SPOF by making IPs highly available. VRRP implements virtual routing between two or more servers in a static, default routed environment. Considering a master router failure event, the backup node takes the master state after a period of time.

In a standard VRRP setup, the backup node keeps listening for multicast packets from the master node with a given priority. If the backup node fails to receive any VRRP advertisement packets for a certain period, it will take over the master state by assigning the routed IP to itself. In a multibackup setup, the backup node with the same priority will be selected within its highest IP value to be the master one.

OpenStack HA under the hood
Deep down in the murky depths of HA, the setup of our magnificent OpenStack environment is much diversified! It may come across as a bit biased to favor a given HA setup, but remember that depending on which software clustering solution you feel more comfortable with, you can implement your HA OpenStack setup.

Let's shine the spotlight brightly on our first OpenStack design in Chapter 1, Designing OpenStack Cloud Architecture, and take a closer look at the pieces in the HA mode.

Next, we will move on to specific OpenStack core components and end up with exposing different possible topologies.
OpenStack HA and Failover

HA the database

There's no doubt that behind any cluster, lies a story! Creating your database in the HA mode in an OpenStack environment is not negotiable. We have set up MySQL in cloud controller nodes that can also be installed on separate ones. Most importantly, keep it safe not only from water, but also from fire. Many clustering techniques have been proposed to make MySQL highly available. Some of the MySQL architectures can be listed as follows:

- **Master/slave replication**: As exemplified in the following figure, a VIP that can be optionally moved has been used. A drawback of such a setup is the probability of data inconsistency due to delay in the VIP failing over (data loss).

![Master/slave replication diagram]

- **MMM replication**: By setting two servers, both of them become masters by keeping only one acceptable write query at a given time. This is still not a very reliable solution for OpenStack database HA as in the event of failure of the master, it might lose a certain number of transactions:
MySQL shared storage: Both servers will depend on a redundant shared storage. As shown in the following figure, a separation between servers processing the data and the storage devices is required. Note that an active node may exist at any point in time. If it fails, the other node will take over the VIP after checking the inactivity of the failed node and turn it off. The service will be resumed in a different node by mounting the shared storage within the taken VIP.
Such a solution is excellent in terms of the uptime, but it may require a powerful storage/hardware system which could be extremely expensive.

- **Block-level replication**: One of the most adopted HA implementations is the DRBD replication, which stands for **Distributed Replicated Block Device**. Simply put, it replicates data in the block device, which is the physical hard drive between OpenStack MySQL nodes.

What you need are just Linux boxes. The DRBD works on their kernel layer exactly at the bottom of the system I/O stack.

With shared storage devices, writing to multiple nodes simultaneously requires a cluster-aware filesystem, such as the Linux **Global File System (GFS)**.

DRBD can be a costless solution, but performance-wise, it cannot be a deal when you rely on hundreds of nodes. This can also affect the scalability of the replicated cluster.

- **MySQL Galera multimaster replication**: Based on multimaster replication, the Galera solution has a few performance challenges within an MMM architecture for the MySQL/innoDB database cluster. Essentially, it uses synchronous replication where data is replicated across the whole cluster. As was stated in our first logical design in *Chapter 1, Designing OpenStack Cloud Architecture*, a requirement of the Galera setup is the need for at least three nodes to run it properly. Let's dive into the Galera setup within our OpenStack environment and see what happens under the hood. In general, any MySQL replication setup can be simple to set up and make HA-capable, but data can be lost during the failing over. Galera is tightly designed to resolve such a conflict in the multimaster database environment. An issue you may face in a typical multimaster setup is that all the nodes try to update the same database with different data, especially when a synchronization problem occurs during the master failure. This is why Galera uses **Certification Based Replication (CBR)**.
Keep things simple; the main idea of CBR is to assume that the database can roll back uncommitted changes, and it is called **transactional** in addition to applying replicated events in the same order across all the instances. Replication is truly parallel; each one has an ID check. What Galera can bring as an added value to our OpenStack MySQL HA is the ease of scalability; there are a few more things to it, such as joining a node to Galera while it is automated in production. The end design brings an active-active multimaster topology with less latency and transaction loss.

A very interesting point in the last illustration is that every MySQL node in the OpenStack cluster should be patched within a **Write-Set Replication (wsrep)** API. If you already have a MySQL master-master actively working, you will need to install wsrep and configure your cluster.

Wsrep is a project that aims to develop a generic replication plugin interface for databases. Galera is one of the projects that use wsrep APIs by working on its wsrep replication library calls.

You can download and install Galera from [https://launchpad.net/galera/](https://launchpad.net/galera/). Every node will need a certain number of steps to configure a complete MySQL cluster setup.
OpenStack HA and Failover

HA in the queue
RabbitMQ is mainly responsible for communication between different OpenStack services. The question is fairly simple: no queue, no OpenStack service intercommunication. Now that you get the point, another critical service needs to be available and survive the failures. RabbitMQ is mature enough to support its own cluster setup without the need to go for Pacemaker or another clustering software solution.

The amazing part about using RabbitMQ is the different ways by which such a messaging system can reach scalability using an active/active design with:

- **RabbitMQ clustering**: Any data or state needed for the RabbitMQ broker to be operational is replicated across all nodes.
- **RabbitMQ mirrored queues**: As the message queue cannot survive in nodes in which it resides, RabbitMQ can act in active/active HA message queues. Simply put, queues will be mirrored on other nodes within the same RabbitMQ cluster. Thus, any node failure will automatically switch to using one of the queue mirrors.

Setting up queue mirroring does not enhance any load distribution across the cluster and only guarantees availability.

A good reference on the HA of queues within RabbitMQ can be found here: https://www.rabbitmq.com/ha.html.

Like any standard cluster setup, the original node handling the queue can be thought of as a master, while the mirrored queues in different nodes are purely slave copies. The failure of the master will result in the selection of the oldest slave to be the new master.

Keep calm and use HA
So far, we have introduced most of the possibilities that can make our OpenStack environment highly available. OpenStack cloud controller nodes, database clusters, and network nodes can be deployed in redundancy in the following ways:

- MySQL high availability through Galera active/active multimaster deployment and Keepalived
- RabbitMQ active-active high availability using mirrored queues and HAProxy for load balancing
The OpenStack API services' inclusion of nova-scheduler and glance-registry in cloud controllers nodes in the active-passive model using Pacemaker and Corosync

Neutrons agents using Pacemaker

Implementing HA on MySQL

In this implementation, we will need three separate MySQL nodes and two HAProxy servers, so we can guarantee that our load balancer will fail over if one of them fails. Keepalived will be installed in each HAProxy to control VIP. Different nodes in this setup will be assigned as the following:

- **VIP**: 192.168.47.47
- **HAProxy01**: 192.168.47.120
- **HAProxy02**: 192.168.47.121
- **MySQL01**: 192.168.47.125
- **MySQL02**: 192.168.47.126
- **MySQL03**: 192.168.47.127

In order to implement HA on MySQL, perform the following steps:

1. First, let’s start by installing and configuring our HAProxy servers:
   ```
   packtpub@haproxy1$ sudo yum update
   packtpub@haproxy1$ sudo yum install haproxy keepalived
   ```

2. Check whether the HAProxy is properly installed:
   ```
   packtpub@haproxy1$ haproxy -v
   HA-Proxy version 1.5.2 2014/07/12
   ```

3. Let’s configure our first HAProxy node. We start by backing up the default configuration file:
   ```
   packtpub@haproxy1$ sudo cp /etc/haproxy/haproxy.cfg /etc/haproxy/haproxy.cfg.bak
   packtpub@haproxy1$ sudo nano /etc/haproxy/haproxy.cfg
   ```

   ```
   global
   log 127.0.0.1 local2
   chroot /var/lib/haproxy
   pidfile /var/run/haproxy.pid
   ```
maxconn 1020  # See also: ulimit -n
user haproxy
group haproxy
daemon
stats socket /var/lib/haproxy/stats.sock mode 600 level admin
stats timeout 2m

defaults
mode tcp
log global
option dontlognull
option redispatch
retries 3
timeout queue 45s
timeout connect 5s
timeout client 1m
timeout server 1m
timeout check 10s
maxconn 1020

listen haproxy-monitoring *:80
mode tcp
stats enable
stats show-legends
stats refresh 5s
stats uri /
stats realm Haproxy\ Statistics
stats auth monitor:packadmin
stats admin if TRUE

frontend haproxy1  # change on 2nd HAProxy
bind *:3306
default_backend mysql-os-cluster
backend mysql-os-cluster
  balance roundrobin
  server mysql01 192.168.47.125:3306 maxconn 151 check
  server mysql02 192.168.47.126:3306 maxconn 151 check
  server mysql03 192.168.47.127:3306 maxconn 151 check

4. Start the haproxy service:
   packtpub@haproxy1$ sudo service haproxy start

5. Repeat steps 1 to 4, replacing haproxy1 with haproxy2 in the frontend section.

6. Now, we arm our HAProxy servers by adding the VRRP /etc/keepalived/keepalived.conf file. But first, we back up the original configuration file:
   packtpub@haproxy1$ sudo cp /etc/keepalived/keepalived.conf /etc/keepalived/keepalived.conf.bak
   packtpub@haproxy1$ sudo nano /etc/keepalived/keepalived.conf

   To bind a virtual address that does not exist physically on the server, you can add the following option to sysctl.conf in your CentOS box:
   net.ipv4.ip_nonlocal_bind=1

   Do not forget to activate the change using the following:
   packtpub@haproxy1$ sudo sysctl -p
   packtpub@haproxy1$ sudo nano /etc/keepalived/keepalived.conf

   vrrp_script chk_haproxy {
      script "killall -0 haproxy"
      interval 2
      weight 2
   }

   vrrp_instance MYSQL_VIP {
      interface eth0
      virtual_router_id 120
      priority 111  # Second HAProxy is 110
      advert_int 1
virtual_ipaddress {
    192.168.47.47/32 dev eth0
}

track_script {
    chk_haproxy
}

7. Repeat step 6 by replacing the priority to 110, for example, in the HAProxy2 node.

8. Check whether the VIP was assigned to eth0 in both the nodes:
   packtpub@haproxy1$ ip addr show eth0
   packtpub@haproxy2$ ip addr show eth0

9. Now you have HAProxy and Keepalived ready and configured; all we need to do is set up the Galera plugin through all the MySQL nodes in the cluster:
   packtpub@db01$ wget https://launchpad.net/codership-mysql/5.6/5.6.16-25.5/+download/MySQL-server-5.6.16_wsrep_25.5-1.rhel6.x86_64.rpm
   packtpub@db01$ wget https://launchpad.net/galera/0.8/0.8.0/+download/galera-0.8.0-x86_64.rpm

10. We need to install the previously downloaded rpm files using:
    packtpub@db01$ sudo rpm -Uvh galera-0.8.0-x86_64.rpm
    packtpub@db01$ sudo rpm -Uvh MySQL-server-5.6.16_wsrep_25.5-1.rhel6.x86_64.rpm

If you did not install MySQL within Galera from scratch, you should stop the mysql service first before proceeding with the Galera plugin installation. The example assumes that MySQL is installed and stopped. More information about the usage of Galera in OpenStack can be found here: http://docs.openstack.org/high-availability-guide/content/ha-aa-db-mysql-galera.html.
11. Once the Galera plugin is installed, log in to your MySQL nodes and create a new galera user with the galerapass password and, optionally, the haproxy username for HAProxy monitoring without a password for the sake of simplicity. Note that for MySQL clustering, a new sst user must exist. We will set up a new sstpassword password for node authentication:

```
mysql> GRANT USAGE ON *.* to sst@'%' IDENTIFIED BY 'sstpassword';
mysql> GRANT ALL PRIVILEGES on *.* to sst@'%';
mysql> GRANT USAGE on *.* to galera@'%' IDENTIFIED BY 'galerapass';
mysql> INSERT INTO mysql.user (host,user) values ('%','haproxy');
mysql> FLUSH PRIVILEGES;
mysql> quit
```

12. Configure the MySQL wresps Galera library in each MySQL node in /etc/mysql/conf.d/wsrep.cnf.

For db01.packtpub.com, add this code:
```
wsrep_provider=/usr/lib64/galera/libgalera_smm.so
wsrep_cluster_address="gcomm://"
wsrep_sst_method=rsync
wsrep_sst_auth=sst:sstpass
```

Restart the MySQL server:
```
packtpub@db01$ sudo /etc/init.d/mysql restart
```

For db02.packtpub.com, add this code:
```
wsrep_provider=/usr/lib64/galera/libgalera_smm.so
wsrep_cluster_address="gcomm://192.168.47.125"
wsrep_sst_method=rsync
wsrep_sst_auth=sst:sstpass
```

Restart the MySQL server:
```
packtpub@db02$ sudo /etc/init.d/mysql restart
```

For db03.packtpub.com, add this code:
```
wsrep_provider=/usr/lib64/galera/libgalera_smm.so
wsrep_cluster_address="gcomm://192.168.47.126"
wsrep_sst_method=rsync
wsrep_sst_auth=sst:sstpass
```
Restart the MySQL server:
```
packtpub@db01$ sudo /etc/init.d/mysql restart
```

Note that the `db01.packtpub.com gcomm://` address is left empty to create the new cluster. The last step will connect to the `db03.packtpub.com` node. To reconfigure it, we will need to modify our `/etc/mysql/conf.d/wsrep.cnf` file and point to `192.168.47.127`:

```
wrep_cluster_address = "gcomm://192.168.47.127"
```

From the MySQL command line, set your global MySQL settings as follows:
```
mysql> set global wsrep_cluster_address='gcomm://192.168.1.140:4567';
```

13. Check whether the Galera replication is running the way it should be running:
```
packtpub@db01$ mysql --e "show status like 'wsrep'" 
```

If your cluster is fine, you should see something like:
```
wsrep_ready = ON
```

Additional checks can be verified from the MySQL command line. In `db01.packtpub.com`, you can run:
```
Mysql> show status like 'wsrep%';
```

```
|wsrep_cluster_size | 3 |
|wsrep_cluster_status| Primary |
|wsrep_connected    | ON |
```

The `wsrep_cluster_size` node that shows value 3 means that our cluster is aware of three connected nodes while the current node is designated as a `wsrep_cluster_status primary` node.

Starting from step 9, you can add a new MySQL node and join the cluster.

Note that we have separated our MySQL cluster from the cloud controller, which means that OpenStack services running in the former node, including Keystone, Glance, Nova, and Cinder as well as Neutron nodes, need to point to the right MySQL server. Remember that we are using HAProxy while VIP is managed by Keepalived for MySQL high availability. Thus, you will need to reconfigure the Virtual IP in each service, as follows:
° **Nova**: /etc/nova/nova.conf  
  sql_connection=mysql://nova:openstack@192.168.47.47/nova

° **Keystone**: /etc/keystone/keystone.conf  
  sql_connection=mysql://keystone:openstack@192.168.47.47/keystone

° **Glance**: /etc/glance/glance-registry.conf  
  sql_connection=mysql://glance:openstack@192.168.47.47/glance

° **Neutron**: /etc/neutron/plugins/openvswitch/ovs_neutron_plugin.ini  
  sql_connection=mysql://neutron:openstack@192.168.47.47/neutron

° **Cinder**: /etc/cinder/cinder.conf  
  sql_connection=mysql://cinder:openstack@192.168.47.47/cinder

Remember that in order to edit your OpenStack configuration files, you will need to restart the corresponding services. Ensure that after each restart, the service is up and running and does not show any error in the log files.

If you are familiar with `sed` and `awk` command lines, it might be easier to reconfigure files using them. You can take a look at another useful shell tool to manipulate ini and conf files; `crudini` can be found at [http://www.pixelbeat.org/programs/crudini/](http://www.pixelbeat.org/programs/crudini/). To update an existing configuration file, the command line is fairly simple:

```
# crudini --set <Config_File_Path> <Section_Name> <Parameter> <Value>
```

To update, for example, the `/etc/nova/nova.conf` file showed previously, you can enter the following command line:

```
# crudini --set /etc/nova/nova.conf database connection mysql://nova:openstack@192.168.47.47/nova
```
Implementing HA on RabbitMQ

In this setup, we will use a node to introduce minor changes to our RabbitMQ instances running in cloud controller nodes. We will enable the mirrored option in our RabbitMQ brokers. In this example, we assume that the RabbitMQ service is running on three OpenStack cloud controller nodes, as follows:

- **VIP**: 192.168.47.47
- **HAProxy01**: 192.168.47.120
- **HAProxy02**: 192.168.47.121
- **Cloud controller 01**: 192.168.47.100
- **Cloud controller 02**: 192.168.47.101
- **Cloud controller 03**: 192.168.47.102

In order to implement HA on RabbitMQ, perform the following steps:

1. Stop RabbitMQ services on the second and third cloud controller. Copy the erlang cookie from the first cloud controller and add the additional nodes:
   ```bash
   packtpub@cc01$ scp /var/lib/rabbitmq/.erlang.cookie root@cc02:/var/lib/rabbitmq/.erlang.cookie
   packtpub@cc01$ scp /var/lib/rabbitmq/.erlang.cookie root@cc03:/var/lib/rabbitmq/.erlang.cookie
   ```

2. Set the rabbitmq group and user with 400 file permissions in both the additional nodes:
   ```bash
   packtpub@cc02$ sudo chown rabbitmq:rabbitmq /var/lib/rabbitmq/.erlang.cookie
   packtpub@cc02$ sudo chmod 400 /var/lib/rabbitmq/.erlang.cookie
   packtpub@cc03$ sudo chown rabbitmq:rabbitmq /var/lib/rabbitmq/.erlang.cookie
   packtpub@cc03$ sudo chmod 400 /var/lib/rabbitmq/.erlang.cookie
   ```

3. Start the RabbitMQ service in cc02 and cc03:
   ```bash
   packtpub@cc02$ service rabbitmq-server start
   packtpub@cc02$ chkconfig rabbitmq-server on
   packtpub@cc03$ service rabbitmq-server start
   packtpub@cc03$ chkconfig rabbitmq-server on
   ```
4. Now, it's time to form the cluster and enable the mirrored queue option. Currently, all the three RabbitMQ brokers are independent and they are not aware of each other. Let's instruct them to join one cluster unit. First, stop the rabbimqctl daemon.

On the cc02 node, run these commands:

```bash
# rabbitmqctl stop_app
Stopping node 'rabbit@cc02' ...
... done.
# rabbitmqctl join-cluster rabbit@cc01
Clustering node 'rabbit@cc02' with 'rabbit@cc01' ...
... done.
# rabbitmqctl start_app
Starting node 'rabbit@cc02' ...
... done
```

On the cc03 node, run the following commands:

```bash
# rabbitmqctl stop_app
Stopping node 'rabbit@cc03' ...
... done.
# rabbitmqctl join-cluster rabbit@cc01
Clustering node 'rabbit@cc03' with 'rabbit@cc01' ...
... done.
# rabbitmqctl start_app
Starting node 'rabbit@cc03' ...
... done
```

5. Check the nodes in the cluster by running them from any RabbitMQ node:

```bash
# rabbitmqctl cluster_status
Cluster status of node 'rabbit@cc03' ...

[{nodes,[[disc,[rabbit@cc01,'rabbit@cc02',
     'rabbit@cc03']]]},
 {running_nodes,['rabbit@cc01','rabbit@cc02',
    'rabbit@cc03']},
{partitions,[]}]  
... done.
```
6. The last step will instruct RabbitMQ to use mirrored queues. By doing this, mirrored queues will enable both producers and consumers in each queue to connect to any RabbitMQ broker so that they can access the same message queues. The following command will sync all the queues across all cloud controller nodes by setting an HA policy:

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

Note that the previous command line settles a policy where all queues are mirrored to all nodes in the cluster.

7. Edit its configuration file in each RabbitMQ cluster node to join the cluster on restarting `/etc/rabbitmq/rabbitmq.config`:

```
[{rabbit,
  [{cluster_nodes, [{rabbit@cc01', 'rabbit@cc02', 'rabbit@cc03'}], ram}]}]
```

8. We can proceed to set up a load balancer for RabbitMQ. We need to only add a new section in both the haproxy1 and haproxy2 nodes and reload the configurations:

```
listen rabbitmqcluster 192.168.47.47:5670
  mode tcp
  balance roundrobin
  server cc01 192.168.47.100:5672 check inter 5s rise 2 fall 3
  server cc02 192.168.47.101:5672 check inter 5s rise 2 fall 3
  server cc03 192.168.47.102:5672 check inter 5s rise 2 fall 3
```

Note that we are listening on the VIP 192.168.47.47. Reload the configuration on both HAProxy nodes:

```
# service haproxy reload
```

Using VIP to manage both HAProxy nodes as a proxy for RabbitMQ might require you to configure each OpenStack service to use the 192.168.47.47 address and the 5670 port. Thus, you will need to reconfigure the RabbitMQ settings in each service in the VIP, as the following:

- **Nova**: `/etc/nova/nova.conf`:

  ```
  # crudini --set /etc/nova/nova.conf DEFAULT rabbit_host 192.168.47.47
  # crudini --set /etc/nova/nova.conf DEFAULT rabbit_port 5470
  ```
Chapter 6

- **Glance**: /etc/glance/glance-api.conf:
  
  ```
  # crudini --set /etc/glance/glance-api.conf DEFAULT
  rabbit_host 192.168.47.47
  # crudini --set /etc/glance/glance-api.conf DEFAULT
  rabbit_port 5470
  ```

- **Neutron**: /etc/neutron/neutron.conf:
  
  ```
  # crudini --set /etc/neutron/neutron.conf DEFAULT
  rabbit_host 192.168.47.47
  # crudini --set /etc/neutron/neutron.conf DEFAULT
  rabbit_port 5470
  ```

- **Cinder**: /etc/cinder/cinder.conf:
  
  ```
  # crudini --set /etc/cinder/cinder.conf DEFAULT
  rabbit_host 192.168.47.47
  # crudini --set /etc/cinder/cinder.conf DEFAULT
  rabbit_port 5470
  ```

**Implementing HA on OpenStack cloud controllers**

Moving on to the setting up of highly available OpenStack cloud controllers requires a way of managing the services running in the former nodes. Another alternative for the high-availability game is using Pacemaker and Corosync. As a native high-availability and load-balancing stack solution for the Linux platform, Pacemaker depends on Corosync to maintain cluster communication based on the messaging layer. Corosync supports multicast as the default network configuration communication method. For some environments that do not support multicast, Corosync can be configured for unicast. In multicast networks, all the cluster nodes are connected to the same physical network device, it will be necessary to make sure that at least one multicast address is configured in the configuration file. Corosync can be considered as a message bus system that allows OpenStack services running across different cloud controller nodes to manage quorum and cluster membership to Pacemaker. But how does Pacemaker interact with these services? Simply put, Pacemaker uses **Resource Agents (RAs)** to expose the interface for resource clustering. Natively, Pacemaker supports over 70 RAs found in [http://www.linux-ha.org/wiki/Resource_Agents](http://www.linux-ha.org/wiki/Resource_Agents).
OpenStack HA and Failover

In our case, we will use native OpenStack RAs, including:

- The OpenStack compute service
- The OpenStack identity service
- The OpenStack image service

There is a native Pacemaker RA to manage MySQL databases and VIP, which you can use as an alternative for the MySQL Galera replication solution.

In order to implement HA on OpenStack cloud controllers, perform the following steps:

1. Install and configure Pacemaker and Corosync on cloud controller nodes:
   
   ```
   # yum update
   # yum install pacemaker corosync
   
   Corosync allows any server to join a cluster using active-active or active-passive fault-tolerant configurations. You will need to choose an unused multicast address and a port. Create a backup for the original Corosync configuration file and edit /etc/corosync/corosync.conf as follows:
   ```
   ```
   # cp /etc/corosync/corosync.conf /etc/corosync/corosync.conf.bak
   # nano /etc/corosync/corosync.conf
   ```
   ```
   Interface {
      ringnumber: 0
      bindnetaddr: 192.168.47.0
      mcastaddr: 239.225.47.10
      mcastport: 4000
      ....
   }
   ```

In the case of a unicast network, you might be needed to specify the addresses of all nodes that are allowed as members of the OpenStack cluster, in the Corosync configuration file. There is no need for a multicast cluster. A sample example template can be found at http://docs.openstack.org/high-availability-guide/content/_set_up_corosync_unicast.html.
Generate an authorization key on the cc01 node to enable communication between cloud controller nodes:

```bash
# sudo corosync-keygen
```

Copy the generated `/etc/corosync/authkey` and `/etc/corosync/corosync.conf` files to other nodes in the cluster:

```bash
# scp /etc/corosync/authkey /etc/corosync/corosync.conf
    packpub@192.168.47.101:/etc/corosync/
# scp /etc/corosync/authkey /etc/corosync/corosync.conf
    packpub@192.168.47.102:/etc/corosync/
```

Start the Pacemaker and Corosync services:

```bash
# service pacemaker start
# service corosync start
```

A good way to check the setup is to run the following command:

```bash
#crm_mon -1
```

Online: [cc01 cc02 cc03]

First node (cc01)

By default, Corosync uses **Shoot The Other Node In The Head** (STONITH) option. It is used to avoid a split-brain situation where each service node believes that the other(s) is (are) broken and it is the elected one. Thus, in the case of a STONITH death match, the second node, for example, shoots the first one to ensure that there is only one primary node running. In a simple two nodes Corosynced environment, it might be convenient to disable it by running:

```bash
# crm configure property stonith-enabled= "false"
```

On cc01, we can set up a VIP that will be shared between the three servers. We can use 192.168.47.48 as the VIP with a 3-second monitoring interval:

```bash
# crm configure primitive VIP ocf:heartbeat:IPaddr2 params
   ip=192.168.47.48 cidr_netmask=32 op monitor interval=3s
```

We can see that the VIP has been assigned to the cc01 node. Note that the use of the VIP will be assigned to the next cloud controller if cc01 does not show any response during 3 seconds:

```bash
# crm_mon -l
```

Online: [ cc01 cc02]

VIP (ocf::heartbeat:IPaddr2): Started cc01
Optional, you can create a new directory to save all downloaded resource agent scripts under /usr/lib/ocf/resource.d/openstack.

Creating a new VIP will require you to point OpenStack services to the new virtual address. You can overcome such repetitive reconfiguration by keeping both IP addresses of the cloud controller and the VIP. In each cloud controller, ensure that you have exported the needed environment variables as follows:

```
# export OS_AUTH_URL=http://192.168.47.48:5000/v2.0/
```

2. Set up RAs and configure Pacemaker for Nova.

First, download the resource agent in all the three cloud controller nodes:

```
# cd /usr/lib/ocf/resource.d/openstack
# wget https://raw.github.com/leseb/OpenStack-ra/master/nova-api
# wget https://raw.github.com/leseb/OpenStack-ra/master/nova-cert
# wget https://raw.github.com/leseb/OpenStack-ra/master/nova-consoleauth
# wget https://raw.github.com/leseb/OpenStack-ra/master/nova-scheduler
# wget https://raw.github.com/leseb/OpenStack-ra/master/nova-vnc
# chmod a+rx *
```

You can check whether the Pacemaker is aware of new RAs or not by running this:

```
#crm ra info ocf:openstack:nova-api
```

Now, we can proceed to configure Pacemaker to use these agents to control our Nova service. The next configuration creates p_nova_api, a resource to manage the OpenStack nova-api:

```
#crm configure primitive p_nova-api ocf:openstack:nova-api \ 
   params config="/etc/nova/nova.conf" op monitor interval="5s"\ 
   timeout="5s"
```

Create p_cert, a resource to manage the OpenStack nova-cert:

```
#crm configure primitive p_cert ocf:openstack:nova-cert \ 
   params config="/etc/nova/nova.conf" op monitor interval="5s"\ 
   timeout="5s"
```
Create `p_consoleauth`, a resource to manage the OpenStack nova-consoleauth:

```bash
# crm configure primitive p_consoleauth ocf:openstack:nova-consoleauth
    params config="/etc/nova/nova.conf" op monitor interval="5s" timeout="5s"
```

Create `p_scheduler`, a resource to manage the OpenStack nova-scheduler:

```bash
# crm configure primitive p_scheduler ocf:openstack:nova-scheduler
    params config="/etc/nova/nova.conf" op monitor interval="5s" timeout="5s"
```

Create `p_novnc`, a resource to manage the OpenStack nova-vnc:

```bash
# crm configure primitive p_novnc ocf:openstack:nova-vnc
    params config="/etc/nova/nova.conf" op monitor interval="5s" timeout="5s"
```

3. Set up RA and configure Pacemaker for Keystone:

Download the resource agent in all three cloud controller nodes:

```bash
# cd /usr/lib/ocf/resource.d/openstack
# wget https://raw.github.com/madkiss/openstack-resource-agents/master/ocf/keystone
```

Proceed to configure Pacemaker to use the downloaded resource agent to control the Keystone service. The next configuration creates `p_keysone`, a resource to manage the OpenStack identity service:

```bash
# crm configure primitive p_keysone ocf:openstack:keystone
    params config="/etc/keystone/keystone.conf" op monitor interval="5s" timeout="5s"
```

4. Set up RA and configure Pacemaker for Glance.

Download the resource agent in all three cloud controller nodes:

```bash
# cd /usr/lib/ocf/resource.d/openstack
```
Proceed to configure Pacemaker to use the downloaded resource agent to control the Glance API service. The next configuration creates `p_glance-api`, a resource to manage the OpenStack Image API service:

```
# crm configure primitive p_glance-api ocf:openstack:glance-api \
params config="/etc/glance/glance-api.conf" op monitor \
interval="5s" timeout="5s"
```

Create `p_glance-registry`, a resource to manage the OpenStack glance-registry:

```
# crm configure primitive p_glance-registry \
ocf:openstack:glance-registry params config="/etc/glance/ \
glance-registry.conf " op monitor interval="5s" timeout="5s"
```

5. Set up RA and configure Pacemaker for the Neutron server:

Download the resource agent in all three cloud controller nodes:

```
# cd /usr/lib/ocf/resource.d/openstack
# wget https://raw.github.com/madkiss/\nopenstack-resource-agents/master/ocf/neutron-server
```

Now, we can proceed to configure Pacemaker to use these agents to control our Neutron server service. The next configuration creates `p_neutron-server`, a resource to manage the OpenStack networking server:

```
# crm configure primitive p_neutron-server ocf:openstack: \
neutron-server params config="/etc/neutron/neutron.conf" \
op monitor interval="5s" timeout="5s"
```

Check whether our Pacemaker is handling our OpenStack services correctly:

```
# crm_mon -1
Online: [ cc01 cc02 cc03 ]
VIP (ocf::heartbeat:IPaddr2): Started cc01
p_nova-api (ocf::openstack:nova-api):
 Started cc01
p_cert (ocf::openstack:nova-cert):
 Started cc01
p_consoleauth (ocf::openstack:nova-consoleauth):
 Started cc01
```
To use private and public IP addresses, you might need to create two different VIPs. For example, you will have to define your endpoint as follows:

```
keystone endpoint-create --region $KEYSTONE_REGION \
   --service-id $service-id --publicurl 'http://PUBLIC_VIP:9292' \
   --adminurl 'http://192.168.47.48:9292' \
   --internalurl 'http://192.168.47.48:9292'
```

Implementing HA on network nodes

Extending our OpenStack deployment will necessitate the network controller be brought to its own cluster stack. As we have concluded previously, Neutron is very extensible in terms of the plugin and network configuration. Whichever network setup you imply, a network controller will have to sit on three different networks:

- Management network
- Data network
- External network or Internet (Internet access for instances)
OpenStack HA and Failover

To ensure a fault-tolerant network controller cluster, we will use Pacemaker to avoid any SPOF in the overall OpenStack environment:

1. Set up RA and configure Pacemaker for the Neutron L3 agent.
   Download the resource agent in all three cloud controller nodes:
   
   ```bash
   # cd /usr/lib/ocf/resource.d/openstack
   ```

   The Neutron L3 agent provides layer 3 and **Network Address Translation (NAT)** forwarding to allow instances, access to the tenant networks.

   Proceed to configure Pacemaker to use the downloaded resource agent to control Neutron agent L3. The next configuration creates `p_neutron-l3-agent`, a resource to manage the OpenStack Image API service:
   
   ```bash
   # crm configure 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="5s" timeout="5s"
   ```

2. Set up RA and configure Pacemaker for the Neutron DHCP agent.
   Download the resource agent in all three cloud controller nodes:
   
   ```bash
   # cd /usr/lib/ocf/resource.d/openstack
   ```

   By default, the Neutron DHCP agent uses dnsmasq to assign IP addresses to instances.

   Proceed to configure Pacemaker to use the downloaded resource agent to control the Neutron DHCP agent. The next configuration creates `p_neutron-dhcp-agent`, a resource to manage the OpenStack DHCP agent:
   
   ```bash
   # crm configure 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="5s" timeout="5s"
   ```
Chapter 6

3. Set up RA and configure Pacemaker for the Neutron metadata agent.

Download the resource agent in all three cloud controller nodes:

```
# cd /usr/lib/ocf/resource.d/openstack
```

The Neutron metadata agent enables instances on tenant networks to reach the Compute API metadata.

Proceed to configure Pacemaker to use the downloaded resource agent to control the Neutron metadata agent. The next configuration creates `p_neutron-metadata-agent`, a resource to manage the OpenStack metadata agent:

```
# crm configure 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="5s" timeout="5s"
```

Summary

In this chapter, you learned some of the most important concepts about high availability and failover. You also learned the different options available to build a redundant OpenStack architecture with a robust resiliency. You will know how to diagnose your OpenStack design by eliminating any SPOF across all services. We highlighted different open source solutions out of the box to arm our OpenStack infrastructure and make it as fault-tolerant as possible. Different technologies were introduced, such as HAProxy, database replication such as Galera, Keepalived, Pacemaker, and Corosync. This completes the first part of the book that aimed to cover different architecture levels and several solutions to end up with an optimal OpenStack solution for a medium and large infrastructure deployment.

Now that we have crystallized the high availability aspect in our private cloud, we will focus on building a multinode OpenStack environment in the next chapter and dive deeper into orchestrating it. You can call it my first production day.
Where to buy this book

You can buy Mastering OpenStack from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.