OpenStack

源代码分析

2014 年 7 月

作者: 王智民

贡献者:

创建时间: **2014-7-19**

稳定程度: 初稿

修改历史

版本	日期	修订人	说明
1.0	2014-7-19	王智民	初稿

目录

1 引言

1.1 编写目的

本文档结合手册和软件包分析了 OpenStack 的框架和一些关键问题的解决方案。

本文档希望不仅仅从框架上去理解 OpenStack，还要从代码的层次去理解和熟悉 OpenStack 的设计理念，以便于在后续的云计算产品开发过程中参考和借鉴，做到与时俱进，知己知彼。

本文档会根据 OpenStack 的发展做动态持续更新。

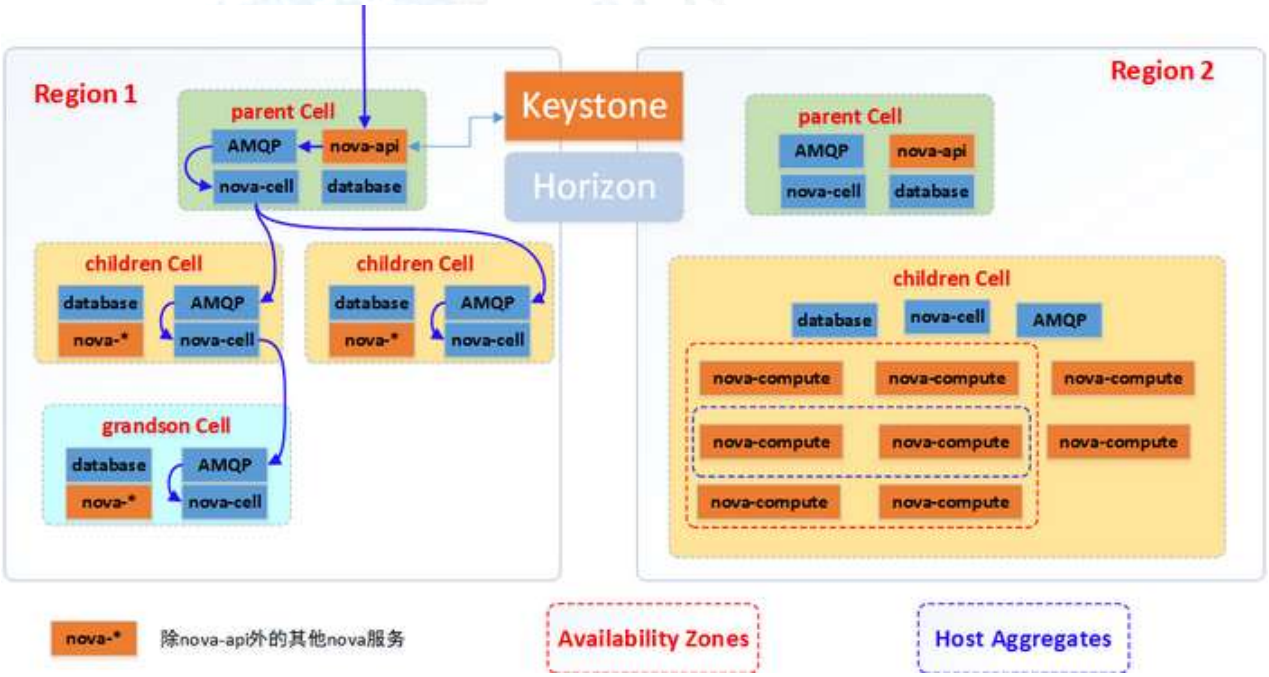
1.2 背景

在云时代，不懂 OpenStack 就好比做嵌入式开发不懂 Linux 一样，闭塞。

在云时代，不用 vMware 产品就好比现在办公不用 windows 一样，别扭。

2 基本概念

2.1 Regions+Cells+Availability Zones+Host Aggregates



1. Regions

更像是一个地理上的概念，每个 region 有自己独立的 endpoint，regions 之间完全隔离，但是多个 regions 之间共享同一个 keystone 和 dashboard。（注：目前 openstack 的 dashboard 还不支持多 region），所以除了提供隔离的功能，region 的设计更多侧重地理位置的概念，用户可以选择离自己更近的 region 来部

署自己的服务。

2. Cells

cell 是 openstack 一个非常重要的概念，主要用来解决 openstack 的扩展性和规模瓶颈。众所周知，openstack 是由很多的组件通过松耦合构成，那么当达到一定的规模后，某些模块必然成为整个系统的瓶颈。比较典型的组件就是 database 和 AMQP 了，所以，每个 cell 有自己独立的 DB 和 AMQP。另外，由于 cell 被实现为树形结构，自然而然引入了分级调度的概念。

通过在每级 cell 引入 nova-cell 服务，实现了以下功能：

（1）Messages 的路由，即父 cell 通过 nova-cell 将 Messages 路由到子 cell 的 AMQP 模块

（2）分级调度功能，即调度某个 instances 的时候先要进行 cell 的选择，目前只支持随机调度，后续会增加基于 filter 和 weighing 策略的调度。

（3）资源统计，子 cell 定时的将自己的资源信息上报给父 cell，用来给分级调度策略提供决策数据和基于 cell 的资源监控

（4）cell 之间的通信（通过 rpc 完成）

最后，所有的子 cell 公用底层 cell 的 nova-api，子 cell 包含除了 nova-api 之外的其他 nova 服务，当然所有的 cell 都共用 keystone 服务。

（注：nova-*是指除了 nova-api 之外的其他 nova 服务，子 cell + 父 cell 才构成了完整的 nova 服务）

3. Availability Zones

AZ 可以简单理解为一组节点的集合，这组节点具有独立的电力供应设备，比如一个个独立供电的机房，一个个独立供电的机架都可以被划分成 AZ。所以，AZ 主要是通过冗余来解决可用性问题。在亚马逊的声明中，instance 不可用指的是用户所有 AZ 中的同一 instances 都不可达才表明不可用。

AZ 之间共享所有的 nova 服务和 keystone 服务。另外，AZ 是用户可见的一个概念，用户在创建 instance 的时候可以选择创建到哪些 AZ 中，以保证 instance 的可用性。

4.Host Aggregates

Host Aggregates 也是一组节点的组合，但强调这组节点具有共同的属性，比如：cpu 是制定类型的一组节点，disks 是 ssd 的一组节点，os 是 linux 或 windows 的一组节点等等。

需要注意的是，Host Aggregates 是用户不可见的概念，主要用来给 nova-scheduler 通过某一属性来进行 instance 的调度，比如讲数据库服务的 instances 都调度到具有 ssd 属性的 Host Aggregate 中，又或者让某个 flavor 或某个 image 的 instance 调度到同一个 Host Aggregates 中。

Host Aggregates 之间共享 keystone 和所有的 nova 服务。

2.2 Domain+Project+Group+User+Rule

Domain 是 Keystone v3 版本提出的概念，project 对应 v2 版本的 tenant。

3 框架分析

3.1 0 层分解

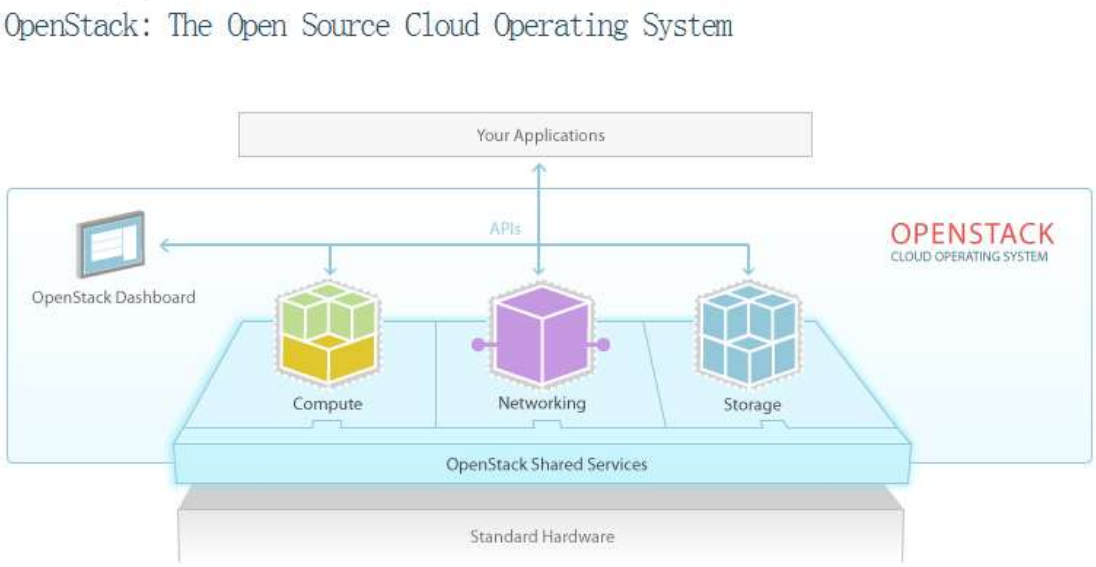
3.1.1 OpenStack 概况

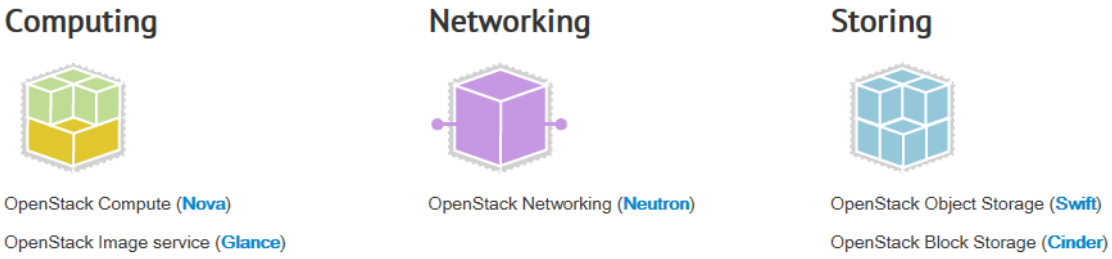
OpenStack is a cloud operating system that controls large pools of compute, storage, and networking resources throughout a datacenter, all managed through a dashboard that gives administrators control while empowering their users to provision resources through a web interface.

The OpenStack Open Source Cloud Mission: to produce the ubiquitous Open Source Cloud Computing platform that will meet the needs of public and private clouds regardless of size, by being simple to implement and massively scalable.

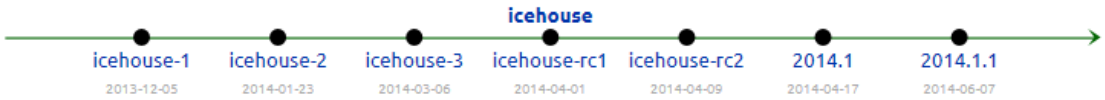
OpenStack 官方网站：http://www.openstack.org/

OpenStack 讨论区，包括获取代码或共享代码：https://wiki.openstack.org/wiki/Main_Page





截止到 2014-7 月 OpenStack 发行版本到了 icehouse:



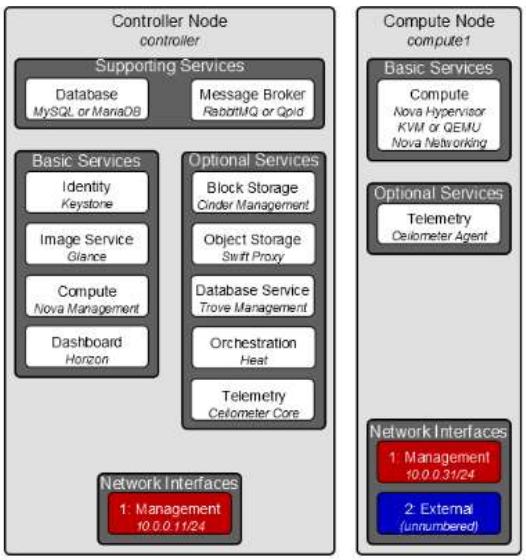
3.1.2 OpenStack 安装

OpenStack 的各个组件都体现为某个或某几个 service，这些 services 可以单独或组合运行在某个或某些物理节点上，也就是说 service 与物理节点是一种松耦合的方式。

当然 service 从角色上来讲基本可以分为两个角色：controller 和 worker，两个角色都支持分布式的运行模式。

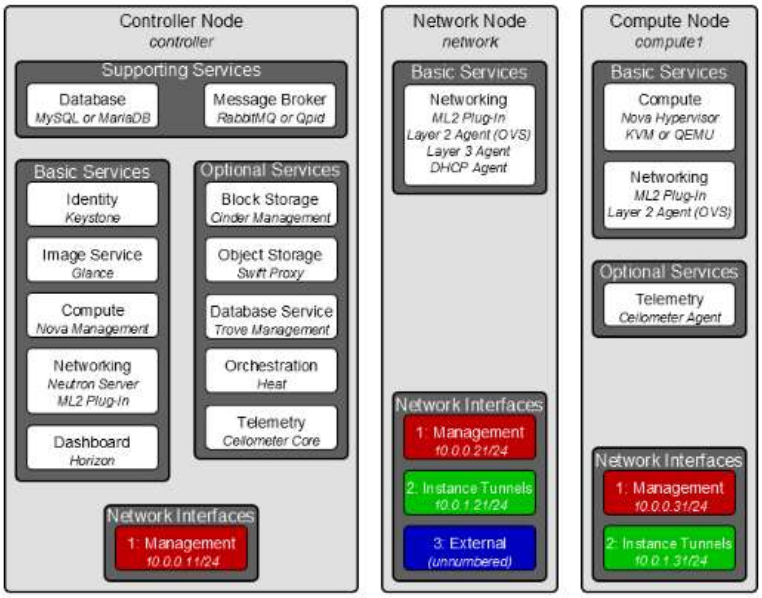
如果不使用 neutron 网络组件，而是用 nova-network 组件，那么最小安装可以是两个节点：

Figure 1.3. Two-node architecture with legacy networking (nova-network)



如果要使用 Nutron 网络组件，简易最小安装 3 个物理节点：

Figure 1.2. Three-node architecture with OpenStack Networking (neutron)



3.2 1 层分解

3.2.1 子项目与服务

1 层分解描述 OpenStack 的子项目以及对应负责的服务。

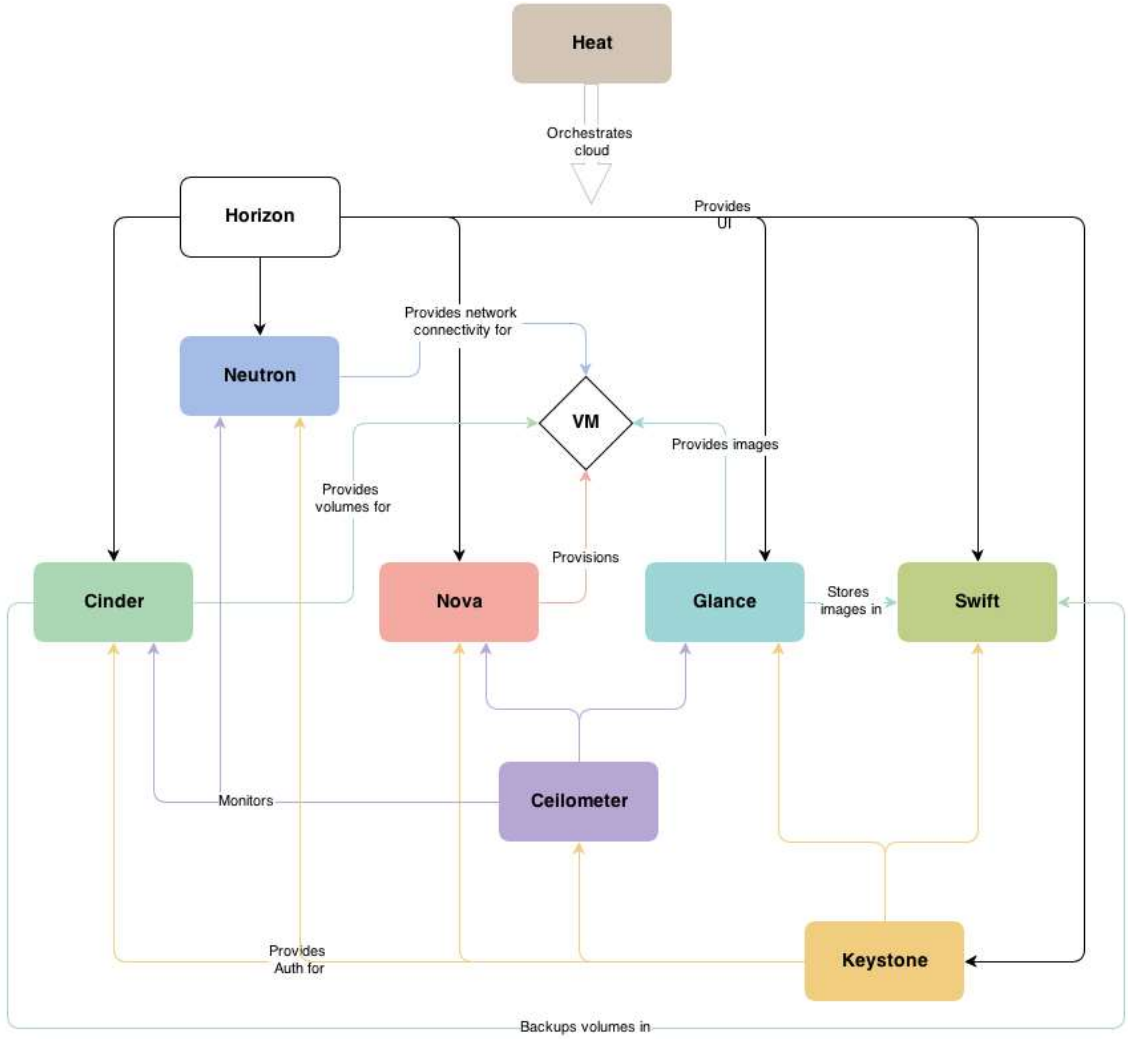
Table 1.1. OpenStack services		
Service	**Project name**	**Description**
Dashboard	Horizon	Provides a web-based self-service portal to interact with underlying OpenStack services, such as launching an instance, assigning IP addresses and configuring access controls.
Compute	Nova	Manages the lifecycle of compute instances in an OpenStack environment. Responsibilities include spawning, scheduling and decommissioning of virtual machines on demand.
Networking	Neutron	Enables network connectivity as a service for other OpenStack services, such as OpenStack Compute. Provides an API for users to define networks and the attachments into them. Has a pluggable architecture that supports many popular networking vendors and technologies.
Storage		
Object	Swift	Stores and retrieves arbitrary unstructured

Storage		data objects via a RESTful, HTTP based API. It is highly fault tolerant with its data replication and scale out architecture. Its implementation is not like a file server with mountable directories.
Block Storage	Cinder	Provides persistent block storage to running instances. Its pluggable driver architecture facilitates the creation and management of block storage devices.
Shared services		
Identity service	Keystone	Provides an authentication and authorization service for other OpenStack services. Provides a catalog of endpoints for all OpenStack services.
Image Service	Glance	Stores and retrieves virtual machine disk images. OpenStack Compute makes use of this during instance provisioning.
Telemetry	Ceilometer	Monitors and meters the OpenStack cloud for billing, benchmarking, scalability, and statistical purposes.
Higher-level services		
Orchestration	Heat	Orchestrates multiple composite cloud applications by using either the native HOT template format or the AWS CloudFormation template format, through both an OpenStack-native REST API and a CloudFormation-compatible Query API.
Database Service	Trove	Provides scalable and reliable Cloud Database-as-a-Service functionality for both relational and non-relational database engines.

3.2.2 Conceptual architecture

Launching a virtual machine or instance involves many interactions among several services. The following diagram provides the conceptual architecture of a typical OpenStack environment.

Figure 1.1. Conceptual architecture



3.2.3 服务组件之间通信

OpenStack uses a message broker to coordinate operations and status information among services. The message broker service typically runs on the controller node. OpenStack supports several message brokers including RabbitMQ, Qpid, and ZeroMQ. However, most distributions that package OpenStack support a particular message broker.

RabbitMQ

Qpid

ZeroMQ

3.3 2 层分解

3.3.1 计算组件 Nova

3.3.1.1 Nova 是如何支持 KVM、XEN 等各种 Hypervisor 的

nova\virt\hyperv\driver.py:

```
…
from nova.virt import driver
...
class HyperVDriver(driver.ComputeDriver):
```

nova\virt\libvirt\driver.py:

```
…
driver_opts = [
    cfg.StrOpt('compute_driver',
               help='Driver to use for controlling virtualization. Options
,
                    'include: libvirt.LibvirtDriver, xenapi.XenAPIDriver, '
                    'fake.FakeDriver, baremetal.BareMetalDriver, '
                    'vmwareapi.VMwareVCDriver, hyperv.HyperVDriver'),
```

libvirtDriver 指 KVM;

XenAPIDriver 指 XEN;

VMwareVCDriver 指 vmware 的 vShpere（ESXi？）

HyperVDriver 指微软的 Hyper-V。

3.3.1.2 KVM 是如何支持各种存储接口的

KVM 需要借助 qemu-kvm。

qemu-kvm 会提供命令选项：

-drive [file=file]

Qemu-kvm 无论是什么存储接口（块存储、本地文件系统、网络文件系统），都是用 file 指向对应的文件。

比如是块存储接口，则 file 指向块设备文件所在的路径和文件；

如果是 NFS 文件接口，则 file 指向 NFS 在本地的挂接点和对应的空文件；

如果是 CIFS 文件接口，则 file 指向 CIFS 在本地的挂接点和对应的空文件；

如果是 RBD 接口，则 file 指向 RBD 在 linux 下体现出来的块设备文件；

如果是 Ceph FS 接口，则 file 指向 Ceph client 给出的文件；（说明：Ceph client 主要作用是将 Ceph FS 做到网络文件系统，类似于 NFS、CIFS，而 Fuse 是一个工具将非网络文件系

统可以网络远程挂载，比如 HDFS、GlusterFS 等都需要借助 Fuse 才能够远程的挂载）

3.3.1.3 Kvm、qemu、kvm-qemu、libvirt、virsh、virt-manager

qemu 是一套虚拟机管理系统，kqemu 是 qemu 的加速器，可以认为是 qemu 的一个插件；qemu 可以虚拟出不同架构的虚拟机，如在 x86 平台上可以虚拟出 power 机器；

kvm 是另外的一套虚拟机管理系统，包括内核虚拟构架和处理器相关模块，其借用了 quem 其它一些组件，kvm 的非内核部分是由 qemu 实现的；加载了模块后，才能进一步通过其他工具创建虚拟机。但仅有 KVM 模块是远远不够的，因为用户无法直接控制内核模块去做事情，还必须有一个用户空间的工具

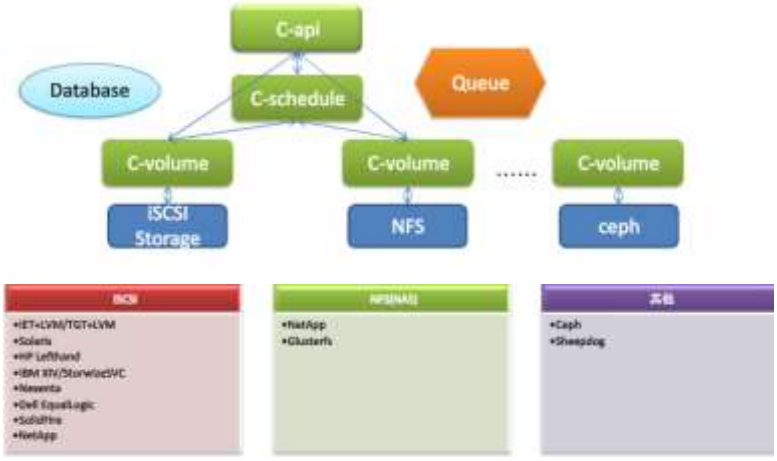
qemu-kvm：kvm 是 linux 的一个模块，管理和创建完整的虚拟机需要相应的一些管理工具，由于 kvm 是在 qemu 的基础上开发的，KVM 使用了 QEMU 的基于 x86 的部分，并稍加改造，形成可控制 KVM 内核模块的用户空间工具 QEMU-KVM。

libvirt，virt-manager,virsh:由于 qemu-kvm 的效率及通用性问题，有组织开发了 libvirt 用于虚拟机的管理，带有一套基于文本的虚拟机的管理工具--virsh，以及一套用户渴望的图形界面管理工具--virt- manager。libvirt 是用 python 语言写的通用的 API，不仅可以管理 KVM，也可用于管理 XEN；

kvm 负责 cpu 虚拟化+内存虚拟化，实现了 cpu 和内存的虚拟化，但 kvm 不能模拟其他设备；qemu 是模拟 IO 设备（网卡，磁盘），kvm 加上 qemu 之后就能实现真正意义上服务器虚拟化，一般称之为 qemu-kvm。

3.3.2 块存储组件 Cinder

Cinder 的体系结构如下：



Cinder provides an infrastructure for managing volumes in OpenStack. It was originally a Nova component called nova-volume, but has become an independent project since the Folsom release.

3.3.2.1 存储基础知识

目前 openstack 的源代码随着模块和功能的增加和完善，代码量快速增长，功能的实现复杂度也随之增加，引进 taskflow 库能够实现方便的代码管理，并且增加功能实现的安全性。简单来说，当实现一个功能时，应用 taskflow 模式能够实现对 flow 执行的管理，能够开始、中止、重新开始以及逆转回滚等操作，比如当执行某个 flow 操作出现异常时，可以视具体情况尝试进行 flow 的逆转回滚操作，实现回溯到 flow 执行之前的状态。

3.3.2.2 taskflow 库

目前 openstack 的源代码随着模块和功能的增加和完善，代码量快速增长，功能的实现复杂度也随之增加，引进 taskflow 库能够实现方便的代码管理，并且增加功能实现的安全性。简单来说，当实现一个功能时，应用 taskflow 模式能够实现对 flow 执行的管理，能够开始、中止、重新开始以及逆转回滚等操作，比如当执行某个 flow 操作出现异常时，可以视具体情况尝试进行 flow 的逆转回滚操作，实现回溯到 flow 执行之前的状态。

3.3.2.3 API 模块

代码目录：cinder\api

cinder 是如何实现卷的建立：

cinder\api\v1\volumes.py:

cinder.api.v1.volumes.VolumeController.create，在这部分内容中，应用了taskflow 库（或者可以理解成一种设计模式）来实现卷的建立过程:

第一步：通过 req 和 body 获取建立卷所需的相关参数，因为建立卷的方法有几种，后面会分析到，具体采用哪种卷的建立方法，是根据 req 和 body 中的相关参数决定的，而且卷的建立也许要确定一些参数，例如卷的大小等等；

第二步：调用方法 create(在 cinder\volume\api.py 中的 class API 中实现的 def create)实现卷的建立；

第三步：获取建立卷后的反馈信息，实现格式转换，并获取其中重要的属性信息，以便上层调用生成卷的建立的响应信息

```
class API(base.Base):
    """API for interacting with the volume manager."""
    def create(self, context, size, name, description, snapshot=None,
                           image_id=None, volume_type=None, metadata=None,
                           availability_zone=None, source_volume=None,
                           scheduler_hints=None, backup_source_volume=None):
        1.构建字典create_what，实现整合建立卷的具体参数；
        create_what = {
                    'context': context,
                    'raw_size': size,
                    'name': name,
                    'description': description,
                    'snapshot': snapshot,
                    'image_id': image_id,
                    'raw_volume_type': volume_type,
                    'metadata': metadata,
                    'raw_availability_zone': availability_zone,
                    'source_volume': source_volume,
                    'scheduler_hints': scheduler_hints,
                    'key_manager': self.key_manager,
                    'backup_source_volume': backup_source_volume,
                }
        2.构建并返回用于建立卷的flow
        try:
                    flow_engine = create_volume.get_flow(self.scheduler_rpcapi,
                                                         self.volume_rpcapi,
                                                         self.db,
                                                         self.image_service,
                                                         check_volume_az_zone,
                                                         create_what)
        3.执行构建的用于建立卷的flow
        flow_engine.run()
        4.从flow中获取建立卷的反馈信息
        volume = flow_engine.storage.fetch('volume')
```

如何采用 taskflow 思路建立卷的 flow：

cinder\volume\flows\api\create_volume.py

```
def get_flow(scheduler_rpcapi, volume_rpcapi, db,
                    image_service,
                    az_check_functor,
                    create_what):
    """Constructs and returns the api entrypoint flow.
    # 获取flow任务的名称；
    flow_name = ACTION.replace(":", "_") + "_api"
    # 获取类Flow的初始化对象；
    # Flow：线性工作流管理类；
    api_flow = linear_flow.Flow(flow_name)
    # 添加一个给定的task到flow；
    # ExtractVolumeRequestTask：实现提取并验证输入的请求信息，并返回经过验证的参数信息，以便用于其他task当中；
    # image_service：获取默认的镜像服务类
    # az_check_functor：验证availability_zone是否是可用的（即是否包含在可用zone的列表中）；
    api_flow.add(ExtractVolumeRequestTask(
        image_service,
        az_check_functor,
        rebind={'size': 'raw_size',
                'availability_zone': 'raw_availability_zone',
                'volume_type': 'raw_volume_type'}))
    api_flow.add(QuotaReserveTask(),
                 EntryCreateTask(db),
                 QuotaCommitTask())
```

3.3.2.4 Volume 模块

代码目录：cinder\volume

卷的创建过程：cinder\volume\api.py:

```
def create(self, req, body):
        """Creates a new volume."""
    ...
    size = volume.get('size', None)
    ...
    new_volume = self.volume_api.create(context,
                                        size,
                                        volume.get('display_name'),
                                        volume.get('display_description'),
                                        **kwargs)
    ...
  return {'volume': retval}
```

----子模块：drivers 和 flow

---------子模块 drivers：cinder\volume\drivers

cinder\volume\drivers\block_device.py:块设备的驱动

class BlockDeviceDriver(driver.ISCSIDriver):

def create_volume(self, volume):
def create_export(self, context, volume):

def copy_image_to_volume(self, context, volume, image_service, image_id):
def copy_volume_to_image(self, context, volume, image_service, image_meta):

def create_cloned_volume(self, volume, src_vref):
def find_appropriate_size_device(self, size):

cinder\volume\drivers\huawei\huawei_hvs.py:某个厂商的驱动

create_volume()-->self.common.create_volume(volume),在下面定义

self.common = HVSCommon(configuration=self.configuration)

cinder\volume\drivers\huawei\rest_common.py:create_volume()

cinder\volume\drivers\rbd.py:

RADOS block device 驱动，RADOS 即 Reliable, Autonomic Distributed Object Store，是 Ceph 的核心之一，作为 Ceph 分布式文件系统的一个子项目，特别为 Ceph 的需求设计，能够在动态变化和异质结构的存储设备机群之上提供一种稳定、可扩展、高性能的单一逻辑对象(Object)存储接口和能够实现节点的自适应和自管理的存储系统。事实上，RADOS 也可以单独作为一种分布式数据存储系统，给适合相应需求的分布式文件系统提供数据存储服务。

class RADOSClient(driver, pool=None)

class RBDDriver(*args, **kwargs)

class RBDImageIOWrapper(rbd_meta)

class RBDImageMetadata(image, pool, user, conf)

class RBDVolumeProxy(driver, name, pool=None, snapshot=None, read_only=False)

cinder\volume\drivers\lvm.py: Driver for Linux servers running LVM.

class LVMISCSIDriver(*args, **kwargs)

class LVMISERDriver(*args, **kwargs)

class LVMVolumeDriver(vg_obj=None, *args, **kwargs)

3.3.3 网络组件 Neutron

3.3.3.1 Neutron 框架

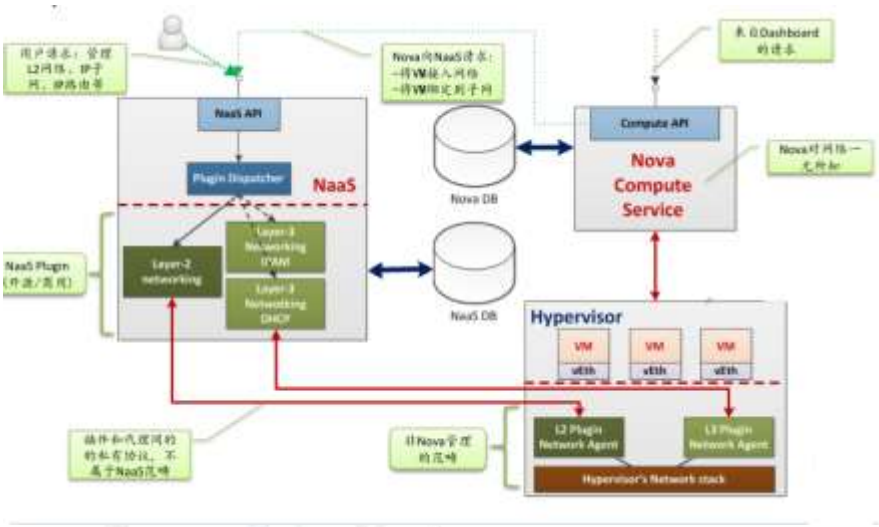
Neutron 的体系结构如下：

网络节点上运行的 Neutron APIs

+网络节点上运行的 Modular Plugin 和非 Modular Plugin

+网络节点上运行的 L3~7 网络 service 服务

+计算节点上运行的对应 plugin 的 agent



【APIs】

目录代码：Neutron\wsgi.py

wsgi 是用 python 语言 web 开发标准，在 wsgi.py 文件中定义了各种类，比如请求 dispatch：

class WorkerService(object): """Wraps a worker to be handled by ProcessLauncher"""

class Server(object): """Server class to manage multiple WSGI sockets and applications."""

class Middleware(object): """Base WSGI middleware wrapper.

class Request(webob.Request):

class ActionDispatcher(object): """Maps method name to local methods through action name."""

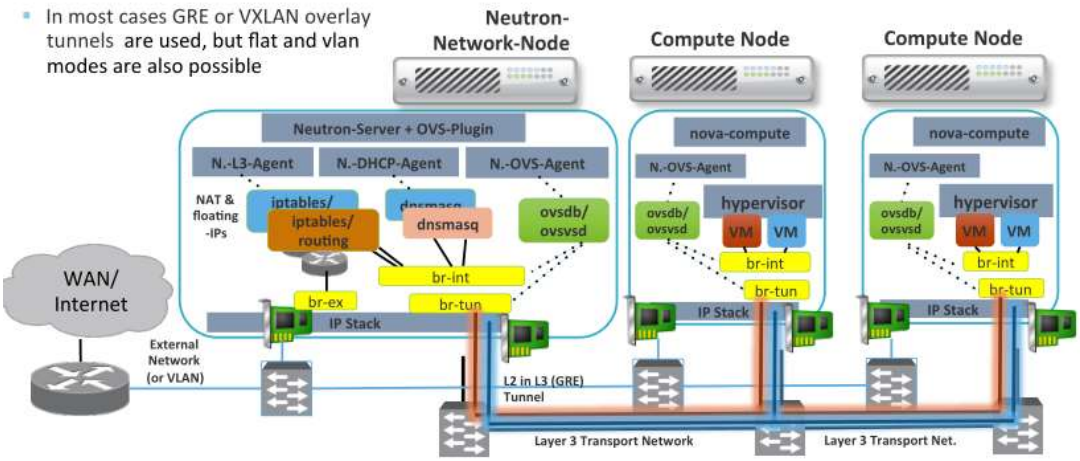
class Application(object):"""Base WSGI application wrapper"""

class Router(object): """WSGI middleware that maps incoming requests to WSGI apps."""

class Controller(object): """WSGI app that dispatched to methods.

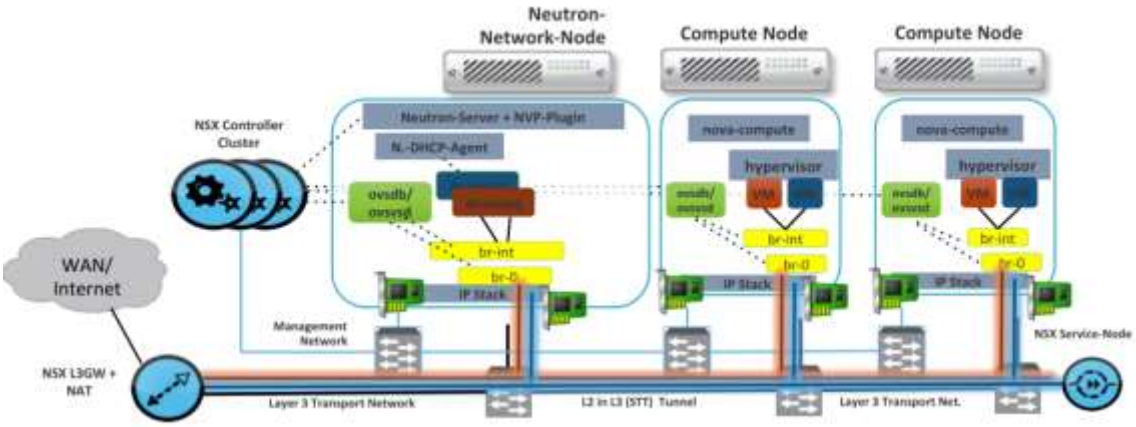
【Plugins】

下图是 OVS 融入到 Neutron 框架所涉及的方面：OVS-Plugin、OVS-agent 和 OVS。OVS-Plugin 运行在网络节点上，OVS-agent 运行在计算节点上，OVS 运行在计算节点上。



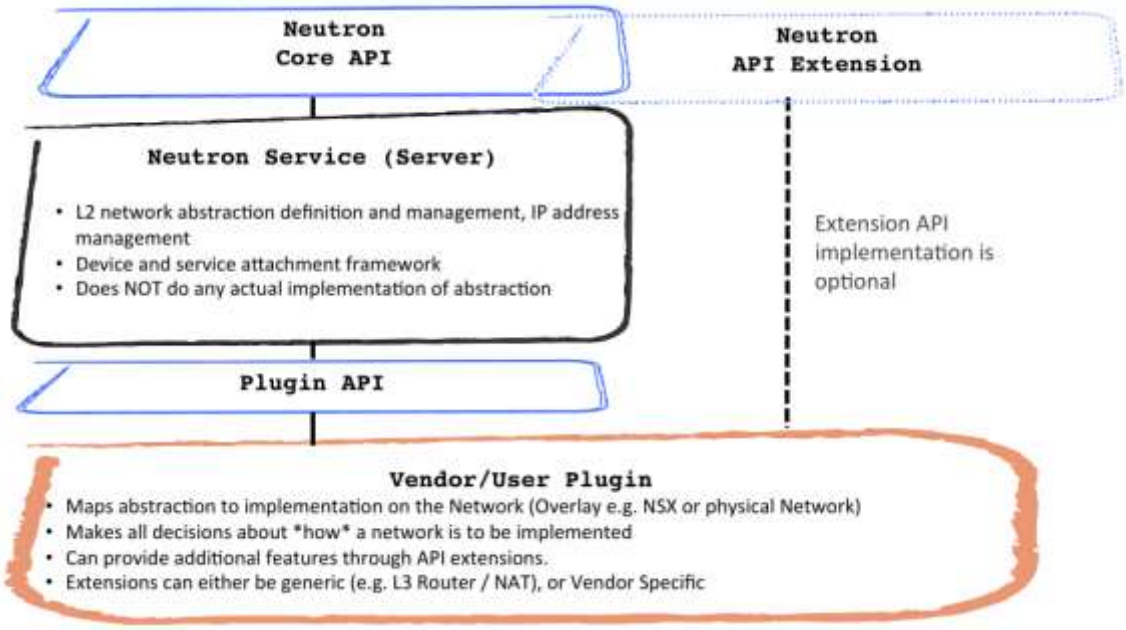
- Neutron-OVS-Agent: Receives tunnel & flow setup information from OVS-Plugin and programs OVS to build tunnels and to steers traffic into those tunnels
- Neutron-DHCP-Agent: Sets up dnsmasq in a namespace per configured network/subnet, and enters mac/ip combination in dnsmasq dhcp lease file
- Neutron-L3-Agent: Sets up iptables/routing/NAT Tables (routers) as directed by OVS Plugin or ML2 OVS mech_driver

下图是 vmware 的 NSX controller 融入到 Neutron 框架中涉及的方面：NVP-Plugin、OVS agent 和 OVS。NVP-Plugin 运行在网络节点上，OVS agent 和 OVS 本身运行在计算节点上。

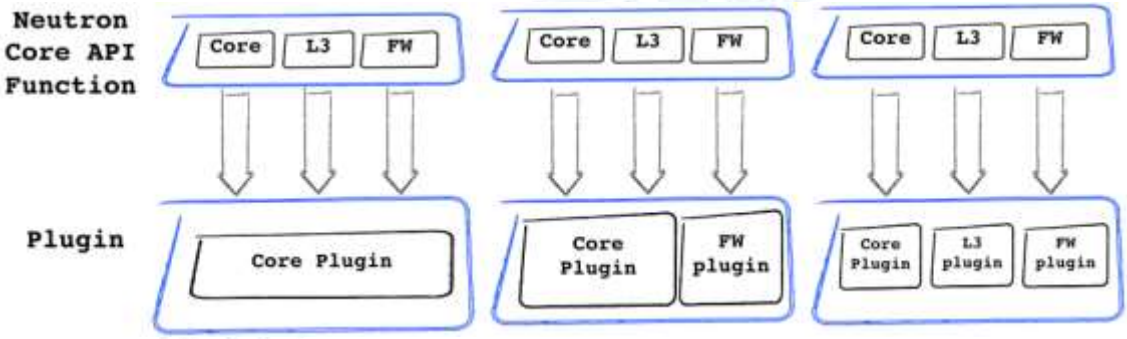


- Centralized scale-out controller cluster controls all Open vSwitches in all Compute- and Network Nodes. It configures the tunnel interfaces and programs the flow tables of OVS
- NSX L3 Gateway Service (scale-out) is taking over the L3 routing and NAT functions
- NSX Service-Node relieves the Compute Nodes from the task of replicating broadcast, unknown unicast and multicast traffic sourced by VMs
- Security-Groups are implemented natively in OVS, instead of iptables/ebtables

Neutron 的 Plugin 的设计框架：



Neutron 将 Plugin 分为两类，一类叫 Core Plugin，一类叫 Service Plugin。Core Plugin 主要是负责 l2 网络、IPADM 等，Service Plugin 主要是 L3~7 的网络服务，比如 NAT、LB 等等。



注意，在 Neutron 中有很多所谓的 drivers，有 agent 的 drivers，有 plugin 的 drivers，那么这些 drivers 是起什么作用的呢？Drivers 主要作用有：

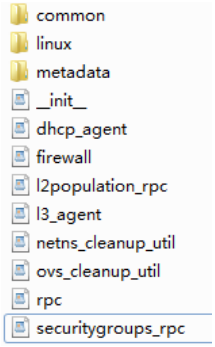
1）无论是 plugin 还是 agent，都需要运行在某个操作系统，一般是 linux，所以需要 plugin 和 agent 能够进入 linux 的初始化过程，以加入到操作系统的运行过程中

2）提供给 plugin 或 agent 一些驱动相关的 API

3）Agent 的驱动还承担与具体的业务模块的交换，比如 OVS agent 需要与 OVS 本身进行交互

3.3.3.2 Agent

Agent 在 Neutron 的代码位置：neutron\agent



可以在 neutron 中的 agent 有 L2、L3、FW、LB 等 agent，一般运行在计算节点上，分别对应有 L2 Plugin(ML2 和 Monolithic）、L3-router Plugin、FW Plugin、LB Plugin，这些 Plugin 一般都运行在网络节点。

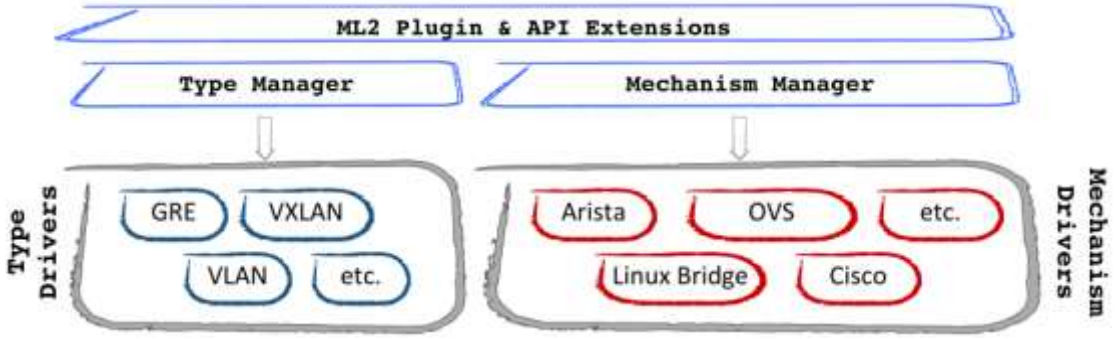
需要注意的是，二层逻辑交换、三层逻辑路由、安全逻辑处理、负载均衡等本身是需要单独的系统来实现的，这个系统可能运行在计算节点上，也可能运行在网络节点上，甚至是存储节点上。比如 OVS 有三个部分：OVS plugin、OVS agent 和 OVS 本身。

3.3.3.3 Core Plugin(ML2+Monolithic)

Core Plugin 里面在 Neutron 版本中又分为两类，一类是 Modular Plugin（ML2），一类是之前版本中就支持的一些 Plugin（monolithic）。

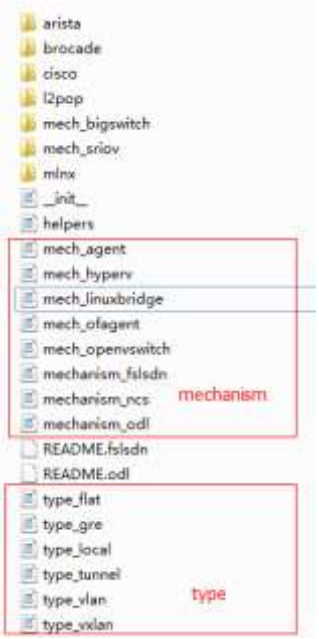
ML2 Plugin separates core functions like IPAM, virtual network id management, etc. from vendor/implementa5on specific func5ons, and therefore makes it easier for vendors not to reinvent to wheel with regards to ID Management, DB access …

ML2 calls the management of network types “type drivers”, and the implement ation on specific part “mechanism drivers”.



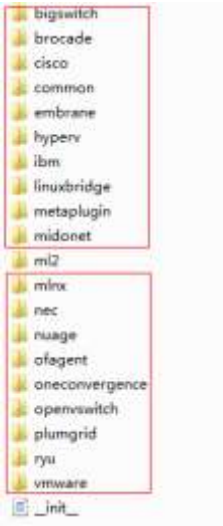
ML2

neutron\plugins\ml2\drivers:



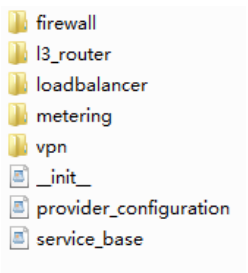
Monolithic plugins

neutron\plugins:

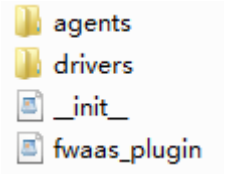


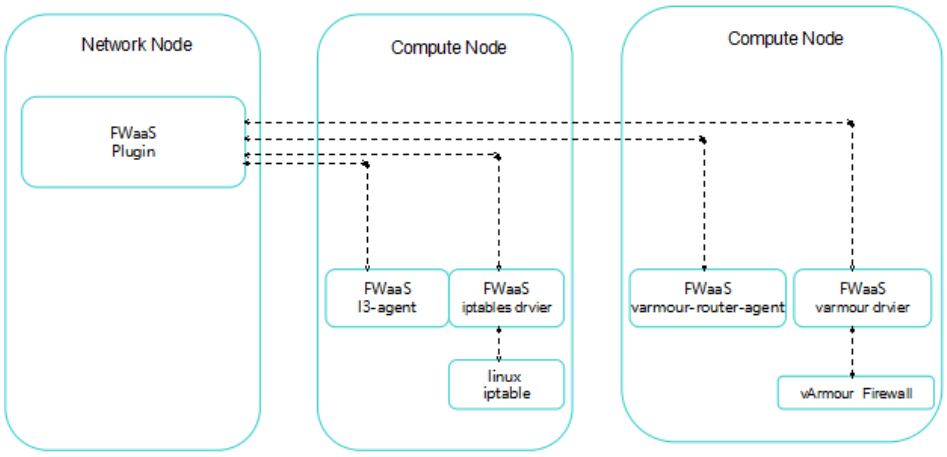
3.3.3.4 Service Plugin(Firewall)

neutron\services:



neutron\services\firewall：防火墙的 plugin、agent 及 drivers





FWaaS Plugin：neutron\services\firewall\fwaas_plugin.py

FWaaS l3-agent：neutron\services\firewall\agents\l3reference\firewall_l3_agent.py

FWaaS iptables driver：neutron\services\firewall\drivers\linux\iptables_fwaas.py

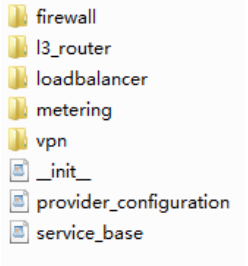
FWaaS varmour-router-agent：neutron\services\firewall\agents\varmour\varmour_router.py

FWaaS varmour driver：neutron\services\firewall\drivers\varmour\varmour_fwaas.py

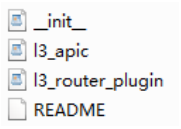
NOTE：Privately held and based in Santa Clara, CA, vArmour Networks is the pioneer of software defined security (SDSec). vArmour's SDSec solution brings a scalable, flexible, programmable network security enforcement layer to highly virtualized computing environments in a way that fits; the security now mirrors the environment, rather than contorting the environment to fit the security. While fully compatible with SDN technologies like OpenFlow, vArmour's SDSec solution can be deployed today in both conventional and virtual networking environments. The result: Intelligent security that works like the rest of your virtual data center.

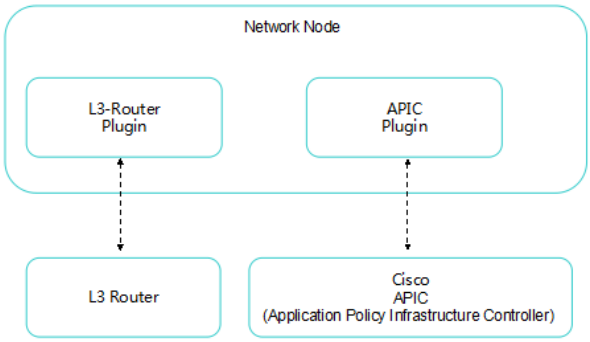
3.3.3.5 Service Plugin(Router)

neutron\services:



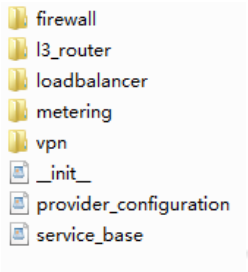
neutron\services\l3_router：



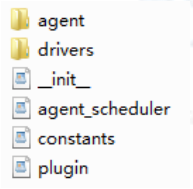


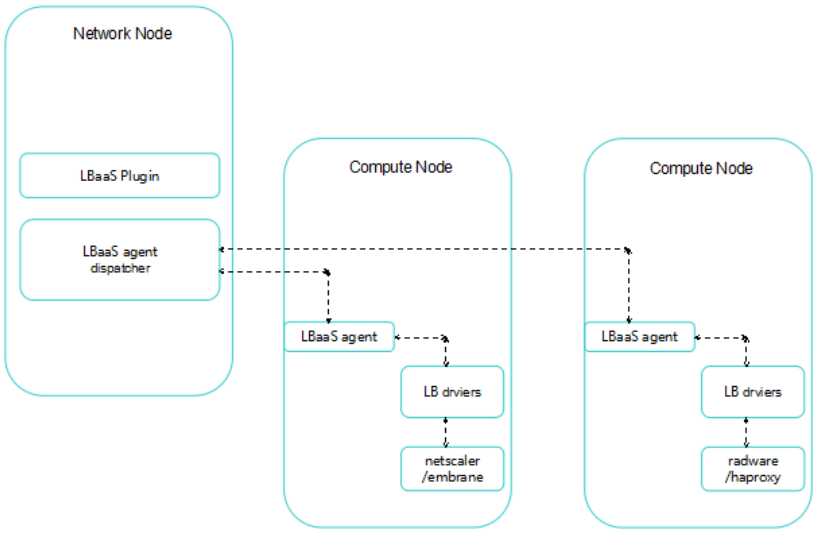
3.3.3.6 Service Plugin(LB)

neutron\services:



neutron\services\loadbalancer:

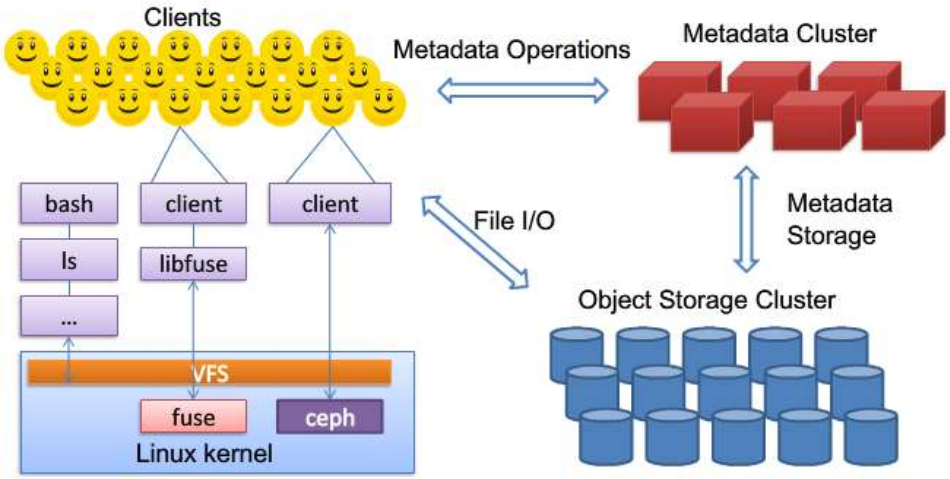




4 Ceph

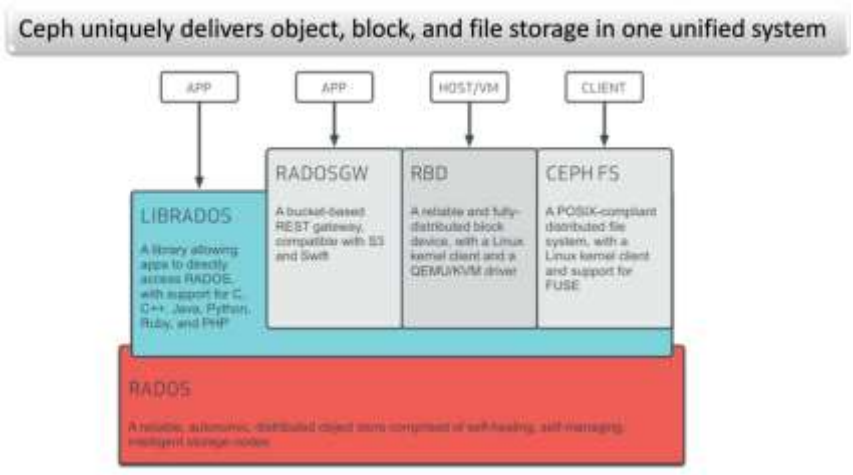
Ceph 是加州大学 Santa Cruz 分校的 Sage Weil（DreamHost 的联合创始人）专为博士论文设计的新一代自由软件分布式文件系统。自 2007 年毕业之后，Sage 开始全职投入到 Ceph 开 发之中，使其能适用于生产环境。Ceph 的主要目标是设计成基于 POSIX 的没有单点故障的分布式文件系统，使数据能容错和无缝的复制。2010 年 3 月，Linus Torvalds 将 Ceph client 合并到内 核 2.6.34 中。

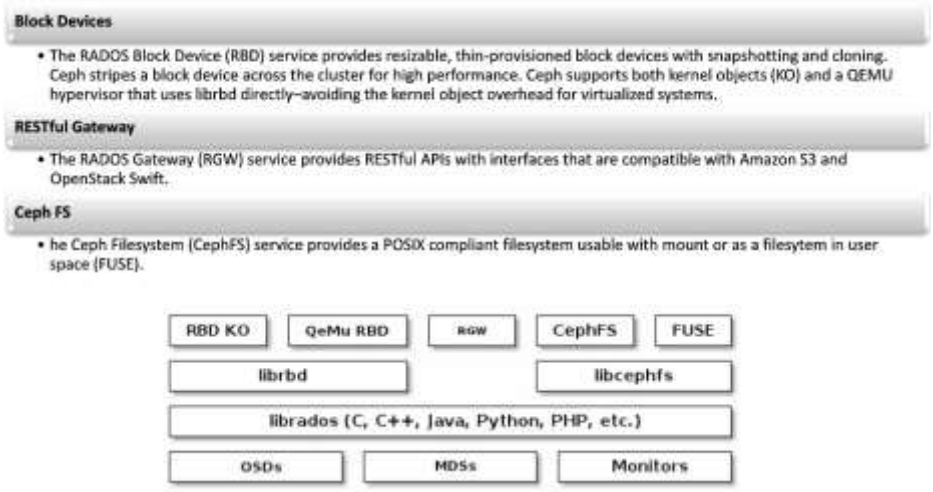
Ceph 的使用场景如下：



4.1 1 级分解

Ceph 1 级分解为 RADOS、ceph client、RADOSGW、RBD 和 CEPH FS 几个模块：





4.2 2 级分解

4.2.1 RADOS

RADOS (Reliable, Autonomic Distributed Object Store) 是 Ceph 的核心之一，作为 Ceph 分布式文件系统的一个子项目，特别为 Ceph 的需求设计，能够在动态变化和异质结构的存储设备机群之上提供一种稳定、可扩展、高性能的单一逻辑对象(Object)存储接口和能够实现节点的自适应和自管理的存储系统。事实上，RADOS 也可以单独作为一种分布式数据存储系统，给适合相应需求的分布式文件系统提供数据存储服务。

RADOS 系统由 OSD(Object Storage Device）、MDS（MetaData Server）和 Monitor 组成，这三个角色都可以是 Cluster 方式运行。

MDS 负责管理 Cluster Map，其中 Cluster Map 是整个 RADOS 系统的关键数据结构，管理机群中的所有成员、关系、属性等信息以及数据的分发。

4.2.1.1 Cluster Map

存储机群的管理，唯一的途径是 Cluster Map 通过对 MDS 操作完成。Cluster Map 是整个 RADOS 系统的核心数据结构，其中指定了机群中的 OSDs 信息和所有数据的分布情况。所有涉及到 RADOS 系统的 Storage 节点和 Clients 都有最新 epoch 的 Cluster Map 副本。因为 ClusterMap 的特殊性，Client 向上提供了非常简单的接口实现将整个存储机群抽象为单一的逻辑对象存储结构。

Cluster Map 的更新由 OSD 的状态变化或者其他事件造成数据层的变化驱动，每一次 Cluster Map 更新都需要将 map epoch 增加，map epoch 使 Cluster Map 在所有节点上的副本都保持同步，同时，map epoch 可以使一些过期的 Cluster Map 能够通过通信对等节点及时更新。

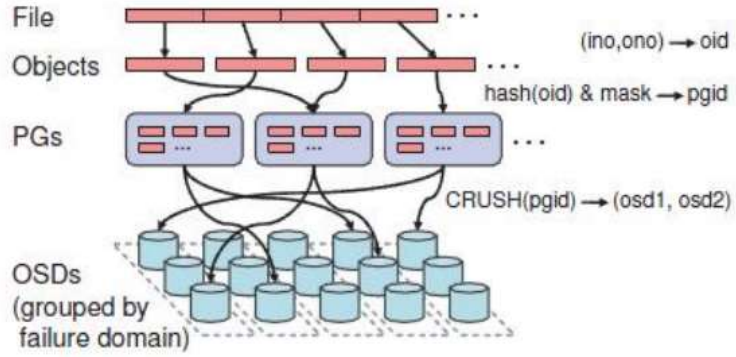
在大规模的分布式系统中，OSDs 的 failures/recoveries 是常见的，所以，Cluster Map 的更新就比较频繁，如果将整个 Cluster Map 进行分发或广播显然会造成资源的浪费，RADOS 采用

分发 incremental map 的策略避免资源浪费，其中 incremental map 仅包含了两个连续 epoch 之间 Cluster Map 的增量信息。

4.2.1.2 Data Placement

数据迁移：当有新的储存设备加入时，机群上的数据会随机的选出一部分迁移到新的设备上，维持现有存储结构的平衡。

数据分发：通过两个阶段的计算得到合适的 Object 的存储位置。



1. Object 到 PG 的映射

PG (Placement Group)是 Objects 的逻辑集合。相同 PG 里的 Object 会被系统分发到相同的 OSDs 集合中。由 Object 的名称通过 Hash 算法得到的结果结合其他一些修正参数可以得到 Object 所对应的 PG。

2. RADOS 系统根据根据 Cluster Map 将 PGs 分配到相应的 OSDs

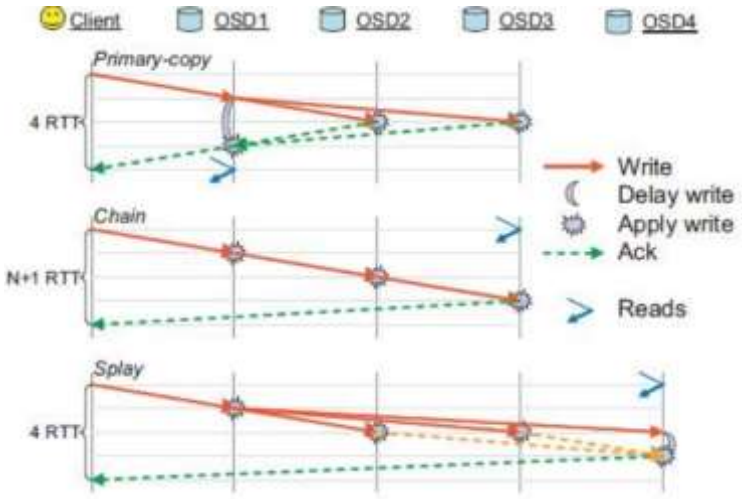
这组 OSDs 正是 PG 中的 Objects 数据的存储位置。RADOS 采用 **CRUSH** 算法实现了一种稳定、伪随机的 hash 算法。CRUSH 实现了平衡的和与容量相关的数据分配策略。CRUSH 得到的一组 OSDs 还不是最终的数据存储目标，需要经过初步的 filter，因为对于大规模的分布式机群，宕机等原因使得部分节点可能失效，filter 就是为过滤这些节点，如果过滤后存储目标不能满足使用则阻塞当前操作。

3. Map propagate（同步）

Cluster Map 在 OSD 之间的更新是通过一种抢占式的方法进行。Cluster Map epoch 的差异只有在两个通信实体之间有意义，两个通信实体在进行信息交换之前都需要交换 epoch，保证 Cluster Map 的同步。这一属性使得 Cluster Map 在通信实体内部之间的更新分担了全局的 Cluster Map 分发压力。

每一个 OSD 都会缓存最近 Cluster Map 和到当前时刻的所有 incremental map 信息，OSD 的所有 message 都会嵌入 incremental map，同时侦听与其通信的 peer 的 Cluster Map epoch。当从 peer 收到的 message 中发现其 epoch 是过期的，OSD share 相对 peer 来说的 incremental map，使通信的 peers 都保持同步；同样的，当从 peer 收到 message 中发现本地 epoch 过期，从其嵌入到 message 中的 incremental map 中分析得到相对本地的 incremental map 然后更新，保持同步。

数据复制（Replication）：RADOS 实现了三种不同的 Replication 方案，见下图：



Primary-copy：读写操作均在 primary OSD 上进行，并行更新 replicas；

Chain：链式读写，读写分离；

Spaly：Primary-copy 和 Chain 的折中方案：并行更新 replicas 和读写分离。

数据一致性（Consistency）：一致性问题主要有两个方面，分别是 Update 和 Read：

Update：在 RADOS 系统中所有 Message 都嵌入了发送端的 map epoch 以协调机群的一致性。

Read：当 read operation 的发起方还不知道待读的 OSD 已经失效，会导致数据不能从该 OSD 读取从而需转向新的 OSD，为了解决此问题，同一个 PG 所在的 OSDs 需要实时交换 Heartbeat。

失效检测（Failure Detection）：RADOS 采取异步、有序的点对点 Heartbeat。

数据迁移与数据恢复（Data Migration & Failure Recovery）：一旦 Cluster Map 发生变化，相应的 OSDs 上的数据也需要做相应的调整。数据的移植和数据恢复都是由 Primary OSD 负责统一完成。

4.2.1.3 元数据管理

RADOS 的元数据 Cluster Map 管理由 MDS 服务器来负责，其工作原理如下：

1. 首先在多个 MDS 中选举 Leader，之后 Leader 向所有 MDS 请求 Map Epoch，MDS 周期性向 Leader 汇报结果并告知其活跃(Active MDS)，Leader 统计 Quorum。

这阶段的意义是保证所有的 MDS 的 Map Epoch 都是最新的，通过 Incremental updates 对已失效的 Cluster Map 进行更新。

2. Leader 周期向每一个 Active MDS 授权许可,分发 Cluster Map 副本给 OSDs 和 Clients 的服务。当授权失效但 Leader 仍没有重新分发,则认为 Leader died，此时重回第一阶段进行 Leader 重选；当 Active MDS 没有周期向 Leader 反馈 ACK，则认为有 MDS died，重回第一阶段进行 Leader 选举并更新 Quorum。

Leader 周期分发 Lease 和 Active MDS 周期反馈 ACK 的另外一个作用是同步 MDS 的 Cluster Map。

Active MDS 收到 Update 请求时，首先验证当前的 Epoch 是否为最新，如果不是，更新后向上汇报到 Leader，Leader 分发给所有的 MDS，同时回收授权，重新开始新一轮的 Leader 选举。

4.2.2 BRD

Cinder 的体系结构如

4.2.3 Ceph client

Cinder 的体系结构如

5 FlashCache

FlashCache is a general purpose writeback block cache for Linux。

FlashCache 是 facebook 技术团队开发的新开源项目，主要目的是用 SSD 硬盘来缓存数据以加速 MySQL 的一个内核模块。可以看到，它最初是用来做数据库加速，但同时，它也被作为通用的缓存模块而设计，能够用于任何搭建在块设备上的应用程序。

5.1 FlashCache 技术特点

一、可靠性

（1）由于 flashcache 本身的电子特性引起的位反转和串扰，会影响可靠性；

（2）flashcache 本身的异常处理机制是否完善；

（3）SSD 的容量一般比较大，在服务器宕机时，如何及时的转移数据并恢复业务运行；

（4）随便负载增大，对 flashcache 的性能会有影响，如何保证其性能相对平滑；

二、cache 策略

（1）目前支持 wb(write back)，wa(write around,应用在读负载很大，随机读，但同时伴随有大量的顺序写的情况，如 HBASE)，wt(write through)三种模式，但 wt 模式性能较差；

三、cache 算法

（1）基于时间，如 LRU

（2）基于效率

（3）兼顾时间和效率，能使 flashcache 的性能在任何情况下，均有较好性能

（4）基于特定业务模型，如 IBM，提出了针对顺序和随机 IO 均有较高性能的 SARC 算法

四、元数据

（1）元数据分布

元数据占有的存储空间，及运行时占用的内存；

元数据占用空间太大，会加剧 SSD 的消耗，并且会损坏和减小 SSD 的寿命；

flashcache 占用存储空间不算太大，但会占用较大内存，如 300G 的 cache，占用 1G 内存；

（2）元数据更新

元数据以元数据块大小（即元数据扇区）为单位进行批量更新；

但没有考虑到定期更新，即如果某 set 内长时间没有 IO，则其中的脏块会长时间得不到清理，危害数据安全；可考虑增加定期更新；

元数据以元数据块大小对齐。

五、数据的组织和管理

（1）512 路组相联哈希；

（2）SSD 被分为多个具有相同大小（默认，512 个块）的 set，set 为数据查找、更新，以及数据清理的范围；

flashcache 的不足

一、可靠性

（1）异常处理机制比较简单，支持 checksum，但软件实现的 checksum 容易制约性能；

二、cache 策略

三、cache 算法

（1）目前只支持 LRU 和 FIFO，实现比较简单，但增加算法十分容易；

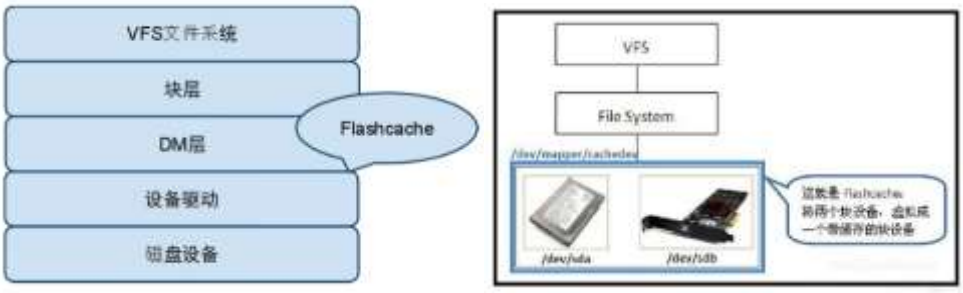
（2）在 LRU 基础上选择合适的二次算法，避免暂时热点的出现；

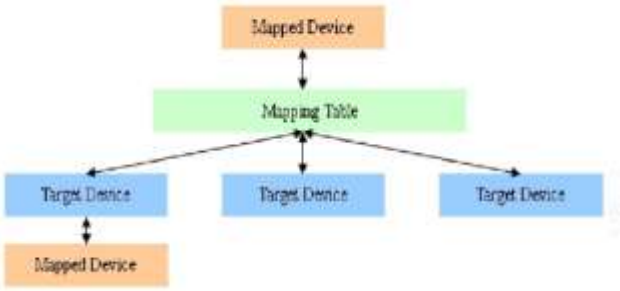
5.2 FlashCache 实现分析

5.2.1 工作原理

基于 Device Mapper，它将快速的 SSD 硬盘和普通的硬盘映射成一个带缓存的逻辑块设备，作为用户操作的接口。用户直接对这个逻辑设备执行读写操作，而不直接对底层的 SSD 或者普通硬盘操作。如果对底层的这些块设备操作，那么会失去作为一个整体提供的缓存功能。

5.2.2 Linux 内核层次





Flashcache 通过在文件系统和块设备驱动层中间增加一缓存层次实现的。

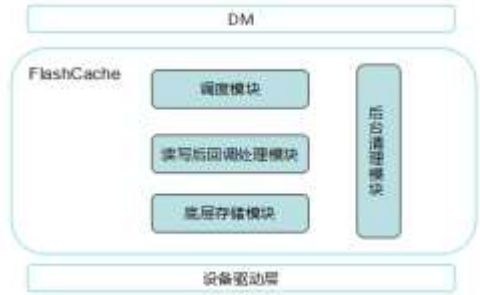
由于 DM 是作为虚拟的块设备驱动在内核中被注册的，它不是一个真实的设备驱动，不能完成 bio 的处理，因此，它主要是基于映射表对 bio 进行分解、克隆和重映射，然后，bio 到达底层真实的设备驱动，启动数据传输。

在 Device mapper 中，引入了 target_driver，每个 target_driver 由 target_type 类型描述，代表了一类映射，它们分别用来具体实现块设备的映射过程。通过调用某一 target_driver 的 map 方法，来映射从上层分发下来的 bio，也即是，找到正确的目标设备，并将 bio 转发到目标设备的请求队列，完成操作。flashcache_target 就是这样一个新的 target_driver（作为一个新的映射类型，target_type 是必须的），以模块化的方式加入到了 DM 层。

```
Flashcache_conf.c:
static struct target_type flashcache_target = {
        .name      = "flashcache",
        .version= {1, 0, 4},
        .module = THIS_MODULE,
        .ctr        = flashcache_ctr,
        .dtr        = flashcache_dtr,
        .map        = flashcache_map,
        .status = flashcache_status,
        .ioctl          = flashcache_ioctl,
};
Flashcache_conf.c:
int __init flashcache_init(void)
{
      ...
      r = dm_register_target(&flashcache_target);
      ...
}
```

5.2.3 逻辑结构

FlashCache 的内部模块逻辑结构如下图所示：



调度模块，在代码中对应 flashcache_map 映射函数，它是 flashcache 缓存层次数据入口，所以到达逻辑设备的读写请求，最终都会经过 DM 层的处理，通过 flashcache_map 进入调度模块。称之为“调度”，主要是指，接收到数据后，它会根据 bio 请求的读写类型、是否命中缓存等因素，选择不同的处理分支，如 flashcache_read/write 或者 flashcache_uncached_io，在 read 和 write 中会选择是 flashcache_read_hit/miss 还是 flashcache_write_hit/miss。经过不同分支的读写，会调用底层存储模块来完成磁盘或 cache 的数据读写。

```
Flashcache_main.c:flashcache_map(struct dm_target *ti, struct bio *bio)
```

读写后回调处理模块，在代码中对应 flashcache_io_callback，它是采用状态机实现的，根据调度模块中的读写类型进行后续的处理，如读未命中情况下，磁盘读完成后，回调到逻辑处理模块，由它负责将从磁盘读取的数据写回到 SSD，或者写未命中情况下，写 SSD 完成后，回调到逻辑处理模块执行元数据的更新，再有就是对调度模块中读写操作的错误进行处理。

```
flashcache_do_io()-->
flashcache_read_hit()--->
flashcache_read_miss()--->
flashcache_write_miss()--->
flashcache_write_hit()-->
Flashcache_main.c:flashcache_io_callback(unsigned long error, void *context)
{
    ...
    switch (job->action)
    {
        case READDISK:
        ...
        schedule_work(&_kcached_wq);
        ...
        case READCACHE:
        ...
        case READFILL:
        ...
        case WRITECACHE:
        ...
    }
    ...
```

```
Flashcache_conf.c:struct work_struct _kcached_wq;
int __init
flashcache_init(void)
{
    ...
    INIT_WORK(&_kcached_wq, do_work, NULL);
```

此模块有一个工作队列_kcached_wq，由 flashcache_io_callback()函数根据情况进行此工作队列的调度。此队列的工作在 do_work()函数中进行，主要是执行元数据的更新与完成处理函数、读磁盘后的 SSD 写入、以及对等待队列的处理。

底层存储模块，主要提供了两种方式来完成真实的数据读写，一是由 DM 提供的 dm_io 函数，它最终还是通过 submit_bio 的方式，将由调度模块处理过的 bio 提交到通用块层，进行转发到真实的设备驱动，完成数据读写；另外，一种方式，kcopyd，是由内核提供的一种底层拷贝函数，主要负责脏块的写回（从 SSD 到磁盘），会引起元数据的更新。

```
Flashcache_conf.c:      .map        = flashcache_map,
int flashcache_map(struct dm_target *ti, struct bio *bio)
{
        ...
        if (unlikely(dmc->sysctl_pid_do_expiry &&
                        (dmc->whitelist_head || dmc->blacklist_head)))
                flashcache_pid_expiry_all_locked(dmc);
        //判断是否可以cache
        uncacheable = (unlikely(dmc->bypass_cache) ||
                        (to_sector(bio->bi_size) != dmc->block_size) ||
                        /*
                       * If the op is a READ, we serve it out of cache whenever possible,
                       * regardless of cacheablity
                       */
                        (bio_data_dir(bio) == WRITE &&
                       ((dmc->cache_mode == FLASHCACHE_WRITE_AROUND) ||
                         flashcache_uncacheable(dmc, bio))));
        if (uncacheable) { //不能cache的情况处理
                flashcache_setlocks_multiget(dmc, bio);
                queued = flashcache_inval_blocks(dmc, bio);
                flashcache_setlocks_multidrop(dmc, bio);
                if (queued) {
                        if (unlikely(queued < 0))
                                flashcache_bio_endio(bio, -EIO, dmc, NULL);
                } else {
                        /* Start uncached IO */
                        flashcache_start_uncached_io(dmc, bio);
                }
         } else { //可以cache的情况
                if (bio_data_dir(bio) == READ)
                        flashcache_read(dmc, bio);
                else
                        flashcache_write(dmc, bio);
         }

}
```

```
static void flashcache_read(struct cache_c *dmc, struct bio *bio)
{
        ...
        res = flashcache_lookup(dmc, bio, &index); //在cache中查找的核心函数
        /* Cache Read Hit case */
         if (res > 0) {    //res > 0表示命中，但是数据可能是valid和invalid
                cacheblk = &dmc->cache[index];
                if ((cacheblk->cache_state & VALID) &&
                      (cacheblk->dbn == bio->bi_sector)) {
                        flashcache_read_hit(dmc, bio, index);
                        return;
                }
         }
        //出错的情况
        if (res == -1 || flashcache_uncacheable(dmc, bio)) {
                ...
                if (res == -1)
                        flashcache_clean_set(dmc, hash_block(dmc, bio->bi_sector), 0);
                /* Start uncached IO */
                flashcache_start_uncached_io(dmc, bio);
                return;
         }
        //cache miss情况
        flashcache_read_miss(dmc, bio, index);
}
```

```
Flashcache_subr.c:      error = flashcache_dm_io_sync_vm(dmc, &where, READ, block);
Flashcache_subr.c:flashcache_dm_io_async_vm(struct cache_c *dmc, unsigned int num_regions,
Flashcache_main.c:flashcache_dirty_writeback(struct cache_c *dmc, int index)
{
    ...
    #if LINUX_VERSION_CODE < KERNEL_VERSION(2,6,26)
        kcopyd_copy(flashcache_kcp_client, &job->job_io_regions.cache, 1, &job->job_io_regions.disk, 0,
    #if LINUX_VERSION_CODE < KERNEL_VERSION(2,6,25)
                    flashcache_kcopyd_callback,
    #else
                    (kcopyd_notify_fn) flashcache_kcopyd_callback,
    #endif
                    job);
    #else
        dm_kcopyd_copy(flashcache_kcp_client,  &job->job_io_regions.cache,  1,
&job->job_io_regions.disk, 0,
                    (dm_kcopyd_notify_fn) flashcache_kcopyd_callback,
                    (void *)job);
    #endif
    ...
}
```

后台清理模块，是针对每个 set 进行数据清理，它会基于两种策略对脏块做回收：（1）set 内脏块超过了阈值；（2）脏块超过了设定的空闲时间，即 fallow_delay，一般是 15 分钟，在 15 分钟没有被操作则会被优先回收。

后台清理有一个队列 delayed_clean。主要负责对整个 cache 设备的脏块清理，由 flashcache_clean_set 在特定条件下调用，通过 flashcache_clean_all_sets()执行对所有 sets 的扫描与清理。

需要注意的是，并没有单独的线程在后台做定期空闲块回收，必须由 IO 操作触发，也就是必须由 IO 操作触发调用 flashcache_clean_set(),进而执行脏块的回收策略：

（1）set 内脏块超过了阈值；

（2）脏块超过了设定的空闲时间，即 fallow_delay，一般是 15 分钟，在 15 分钟没有被操作则会被优先回收

```
Flashcache.h:      struct work_struct delayed_clean;
int flashcache_ctr(struct dm_target *ti, unsigned int argc, char **argv)
{
      ...
       INIT_WORK(&dmc->delayed_clean, flashcache_clean_all_sets, dmc);
      ...
}
void flashcache_clean_set(struct cache_c *dmc, int set, int force_clean_blocks)
{
      ...
      for (i = start_index ;
              (dmc->sysctl_fallow_delay > 0 && //脏块超时
                cache_set->dirty_fallow > 0 &&    //有脏块
                time_after(jiffies, cache_set->fallow_next_cleaning) &&
                i < end_index) ; i++) {
              cacheblk = &dmc->cache[i];
              if (!(cacheblk->cache_state & DIRTY_FALLOW_2))
                      continue;
              if (!flashcache_can_clean(dmc, cache_set, nr_writes)) {
                      /*
                       * There are fallow blocks that need cleaning, but we
                       * can't clean them this pass, schedule delayed cleaning
                       * later.
                       */
                      do_delayed_clean = 1;
                      goto out;
              }
              ...
              flashcache_clear_fallow(dmc, i);
              ...
       }
      if (do_delayed_clean)
            schedule_delayed_work(&dmc->delayed_clean, 1*HZ);
      ...
}
```

IO 操作触发调用 flashcache_clean_set()的地方有：

flashcache_write()

flashcache_write_hit()

flashcache_write_miss()

flashcache_read()

flashcache_read_miss()

flashcache_kcopyd_callback()

flashcache_md_write_done()

flashcache_clean_all_sets()

5.2.4 缓存算法

FlashCache 采用 LRU 缓存算法：

```
int flashcache_ctr(struct dm_target *ti, unsigned int argc, char **argv)
{
        ...
        flashcache_reclaim_init_lru_lists(dmc); //为每个set设置两个lru队列LRU_HOT和LRU_WARM
        flashcache_hash_init(dmc); //为每个set做hash初始化
        ...
}
void flashcache_lru_accessed(struct cache_c *dmc, int index)
{
        ...
        /*
         * Block is accessed.
         *
         * Algorithm :
            if (block is in the warm list) { //
                    block_lru_refcnt++;
                    if (block_lru_refcnt >= THRESHOLD) {
                        clear refcnt
                        Swap this block for the block at LRU end of hot list //Least Recently Used
                    } else
                        move it to MRU end of the warm list    //MRU: More Recently Used
            }
            if (block is in the hot list)
                    move it to MRU end of the hot list
         */
}
```

6 网络虚拟化

网络虚拟化是实现 SDN 的必然技术手段。那么网络虚拟化的技术路线当前有哪些呢？在 OpenStack 生态圈中又有哪些相关的发展呢？

网络虚拟化当前主要有两大阵营：

1）以 vmware 和 microsoft 为代表的云计算软件厂商，典型的技术代名词是 VXLAN 和 NVGRE

2）以 Cisco、BigSwitch 为代表的网络厂商，典型的技术代名词是 OpFlex 和 OpenFlow

6.1 云对网络的需求

下图是私有云和混合云的典型结构，可以看出云对数据中心的网络提出了下面的一些需求：

1）多租户：能够灵活的、随时的划分网络子网，以便于针对租户、客户、内部某部门提供个性化的网络服务（比如带宽、路由转发路径、网络延时等）

2）多租户隔离：网络子网之间可以随时实施隔离、控制等相关的安全策略

3）虚机跨子网无缝迁移：虚机 vm 能够跨网络子网、跨物理地域的热迁移，而不需要改变虚机 vm 的属性（比如 ip 地址）、交换机的 VLAN 属性、路由器的路由表、安全设备的安全策略、负载均衡器的均衡策略等

4）非云向云无缝迁移：云化的数据中心能够与非云化（传统）数据中心安全连接，并且非云化数据中心内部的服务器能够在不改变本身属性（比如 ip 地址）和周边设备的属性的情况下平滑迁移到云化数据中心的某个虚机 vm 上

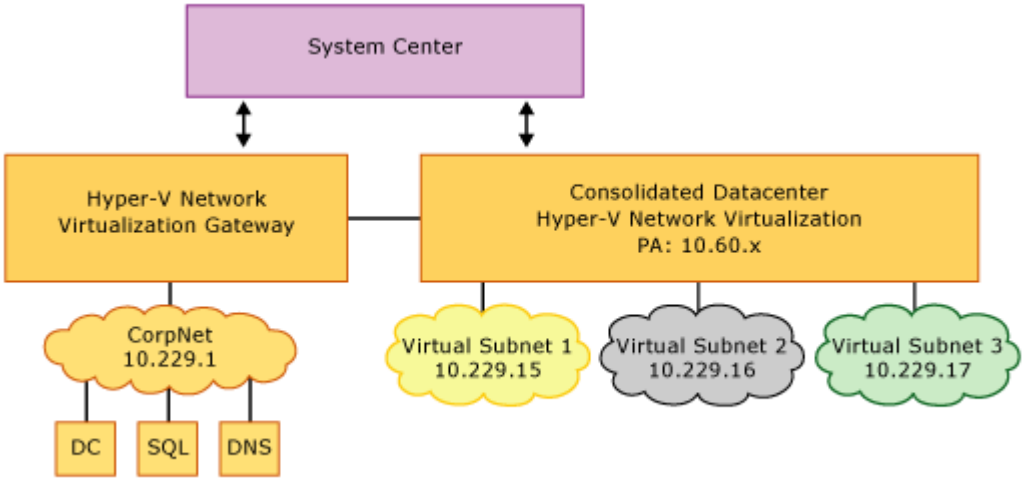


Figure 1: Private cloud deployment

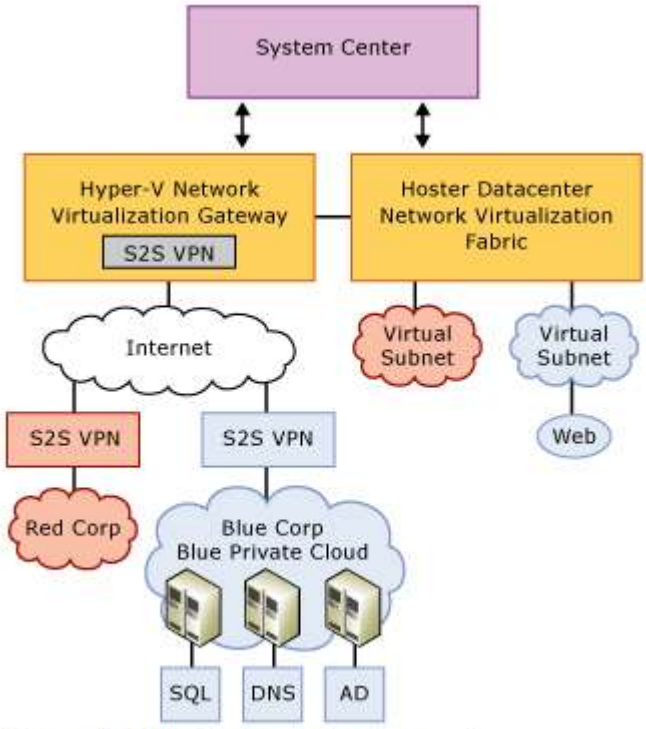


Figure 2: Hybrid cloud deployment

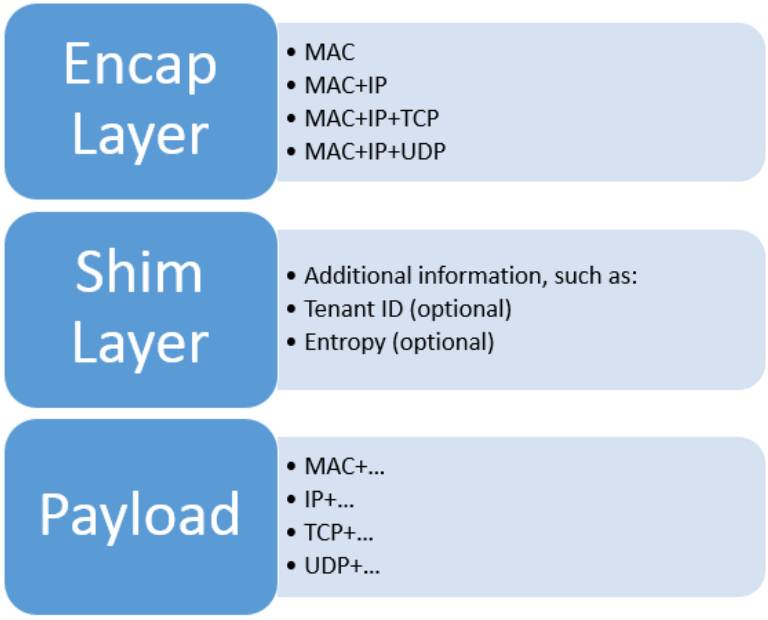
6.2 网络虚拟化常见技术

6.2.1 SDN

不在这里阐述，请参见《SDN 市场与技术预研报告.doc》

6.2.2 隧道技术

隧道封装一般的层次如下：



- 最外层的是封装层（encapsulation layer）

由于需要在网络中传输，必须是合法的二层数据包，因此最外层必须是 MAC。当我们说封装层是 N 层时，意味着加上去了 2 … N 层封装包头。

- 中间的是可选的夹层（shim layer）

包含了一些附加信息和标志位，例如用于标识不同客户的虚拟网络的 Tenant ID，以及用于提高 hash 均匀性的熵（Entropy）。

- 内层的是客户实际发送的数据包

这一层决定了虚拟网络在客户看来是长什么样的。例如 IP-in-IP tunnel 的内层是 IPv4 包，则在客户看来虚拟网络就是个 IPv4 网络，里面跑 TCP、UDP、ICMP 或者任何其他四层协议都行。

虚拟网络并不是层次越低（越靠近物理层）越好，因为越底层的协议越难优化。

Encap Layer	Shim Layer	Payload	Protocol
MAC	…	PPP+…	PPPoE
MAC	Tenant ID…	MAC+…	MAC-in-MAC (PBB)
MAC+IP	Tenant ID, Entropy…	MAC+…	NVGRE
MAC+IP	…	PPP+…	PPTP
MAC+IP+UDP	Tenant ID…	MAC+…	VXLAN
MAC+IP+UDP	Tunnel ID, Session ID…	MAC+…	L2TP
MAC+IP+TCP	Tenant ID (context)…	MAC+…	STT
MAC+IP+UDP/TCP	TLS…	MAC+…	OpenVPN TAP
MAC	Tenant ID… (stackable)	IP+…	MPLS
MAC+IP	(none)	IP+…	IP-in-IP
MAC+IP	Tenant ID, Sequence No…	IP+…	GRE
MAC+IP	(none)	IPv6+…	6to4
MAC+IP+UDP/TCP	TLS…	IP/IPv6+…	OpenVPN TUN
MAC+IP+UDP	(none)	IPv6+…	Teredo
MAC+IP	(none)	TCP/UDP + …	NAPT / NAT
MAC+IP+TCP	TLS…	TCP/UDP + …	SOCKS5

6.2.2.1 NAT

一对一的地址转换：

一对多的地址转换 NAPT（Network Address and Port Translation）：

但是 NAPT 的缺陷是，10.0.0.2 和 10.0.0.3 上面不能提供相同的服务（端口号相同）。

6.2.2.2 IP in IP

NAPT 要求共享公网 IP 的两台机器不能提供相同的服务，这个限制很多时候是不可接受的。比如我们经常需要 SSH 或远程桌面到每一台机器上。隧道技术应运而生。最简单的三层隧道技术是 IP-in-IP。

DMAC	SMAC	Public SIP	Public DIP	Private SIP	Private DIP

头部中有 IP-in-IP 标志（IP protocol number = 0×04，图中未显示）。

但是 IP in IP Tunneling 技术存在下面的一些本身无法解决的问题：

1）在 IP in IP 隧道中只能传输 IPv4 报文，必能传输 ICMP 等报文，则造成隧道两端的无法 ping 通（除非在两端都配置静态 ARP）

2）如果在隧道一端有两个客户的虚拟子网，碰巧这两个子网的私网 IP 地址都重叠了，这时候从另外一端来的报文就无法知道该送给谁了

3）如果负载均衡器无法识别 IP in IP 报文，则只能根据 Public SIP 和 Public DIP 做 hash，Public DIP 一般是相同的，那么只有源 IP 一个变量，hash 的均匀性很难保证

6.2.2.3 GRE vs. IP in IP

GRE（Generic Routing Encapsulation）协议比 IP-in-IP 协议增加了一个中间层（shim layer），包括 32 位的 GRE Key（Tenant ID 或 Entropy）和序列号等信息。GRE Key 解决了前面 IP-in-IP tunnel 的第二个问题，使得不同的客户可以共享同一个物理网络和一组物理机器，这在数据中心中是重要的。

Bits 0–3			4–12	13–15	16–31
C		K	S	Reserved0	Version	Protocol Type
Checksum *(optional)*					Reserved1 *(optional)*
Key *(optional)*					
Sequence Number *(optional)*					

6.2.2.4 GRE vs. NVGRE

GRE 虚拟出来的网络是 IP 网络，这就意味着 IPv6 和 ARP 包不能在 GRE 隧道中传输。IPv6 的问题比较容易解决，只要修改 GRE header 中的 Protocol Type 就行。但 ARP 的问题就不

那么简单了。ARP 请求包是广播包：“Who has 192.168.0.1? Tell 00:00:00:00:00:01”，这反映了二层与三层网络的一个本质区别：二层网络是支持广播域（Broadcast Domain）的。所谓广播域，就是一个广播包应该被哪些主机收到。VLAN 是实现广播域的常用方式。

当然，IP 也支持广播，不过发往三层广播地址（如 192.168.0.255）的包仍然是发往二层的广播地址（ff:ff:ff:ff:ff:ff），通过二层的广播机制实现的。如果我们非要让 ARP 协议在 GRE 隧道中工作，也不是不行，只是大家一般不这么做。

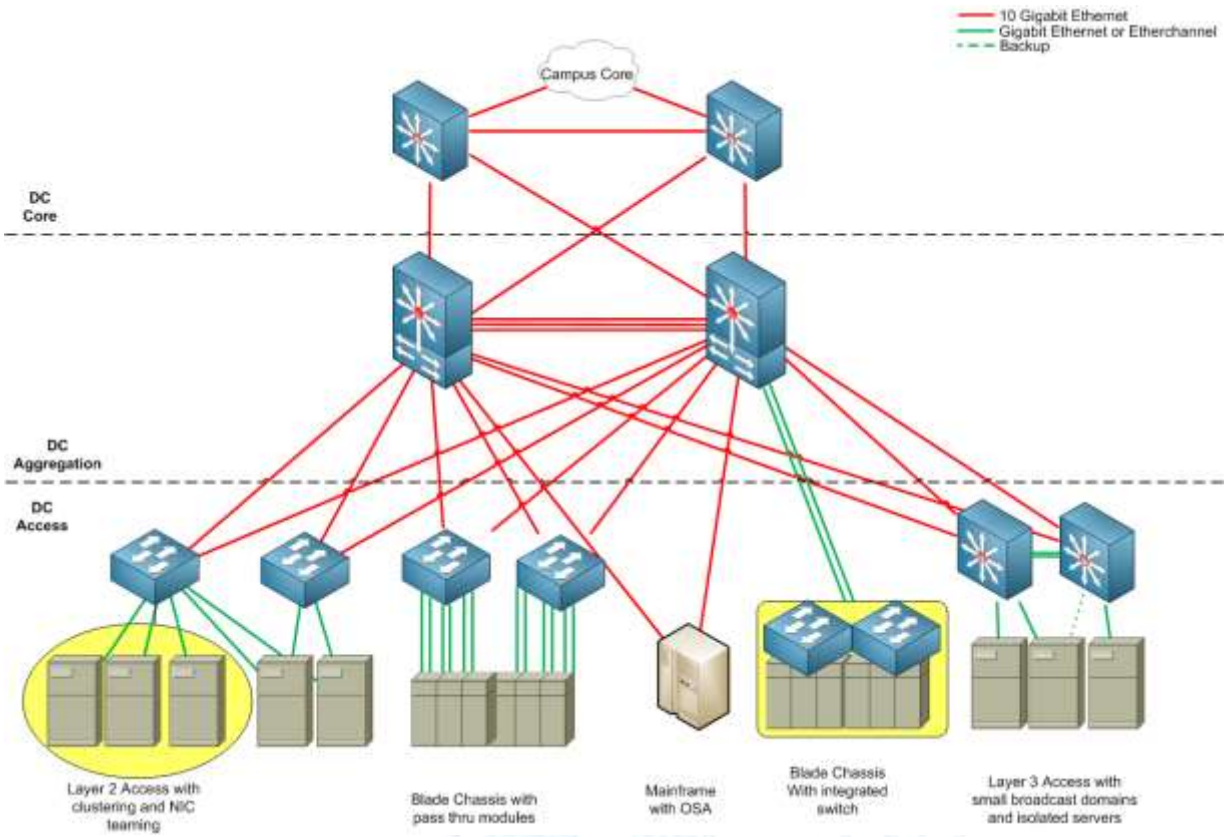
为了支持所有现有的和未来可能有的三层协议，并且支持广播域，就需要客户的虚拟网络是二层网络。NVGRE 和 VXLAN 是两种最著名的二层网络虚拟化协议。

NVGRE（Network Virtualization GRE）相比 GRE 的本质改动只有两处：

- 内层 Payload 是二层 Ethernet 帧而非三层 IP 数据包。注意，内层 Ethernet 帧末尾的 FCS（Frame Check Sequence）被去掉了，因为封装层已经有校验和了，而计算校验和会加重系统负载（如果让 CPU 计算的话）
- 中间层的 GRE key 拆成两部分，前 24 位作为 Tenant ID，后 8 位作为 Entropy

有了 NVGRE，为什么还要用 GRE 呢？抛开历史和政治原因，虚拟网络的层次越低，越不容易优化：

- 如果虚拟网络是二层的，由于 MAC 地址一般是非常零散的，只能给每台主机插入一条转发规则，网络规模大了就是问题。如果虚拟网络是三层的，就可以根据网络拓扑分配 IP 地址，使得网络上临近的主机 IP 地址也在同一子网中（Internet 正是这样做的），这样路由器上只需根据子网的网络地址和子网掩码前缀匹配，能减少大量的转发规则。
- 如果虚拟网络是二层的，ARP 广播等包就会广播到整个虚拟网络，因此二层网络（我们常说的局域网）一般不能太大。如果虚拟网络是三层的，由于 IP 地址是逐级分配的，就不存在这个问题。
- 如果虚拟网络是二层的，交换机要依赖生成树（spanning tree）协议以避免环路。如果虚拟网络是三层的，就可以充分利用路由器间的多条路径增加带宽和冗余性。数据中心的网络拓扑一般如下图所示

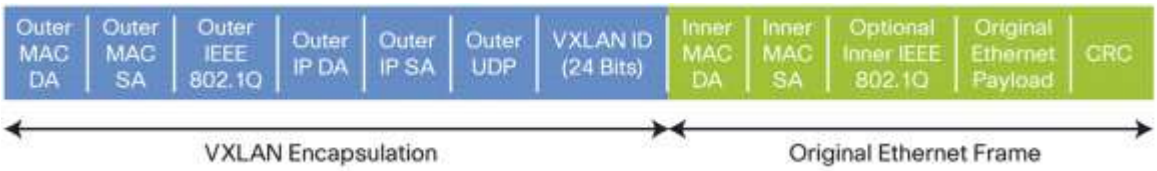


虚拟网络的层次如果更高，payload 里不包括网络层，一般就不能称之为“虚拟网络”了，但仍然属于隧道技术的范畴。SOCKS5 就是这样一种 payload 是 TCP 或 UDP 的协议。它的配置灵活性比基于 IP 的隧道技术更高，例如可以指定 80 端口（HTTP 协议）走一个隧道，443 端口（HTTPS 协议）走另一个隧道。ssh 的 -L（本地转发）、-D（动态转发）参数用的就是 SOCKS5 协议。SOCKS5 的缺点则是不支持任意的三层协议，如 ICMP 协议（SOCKS4 甚至不支持 UDP，因此 DNS 处理起来比较麻烦）。

6.2.2.5 NVGRE vs. VXLAN

NVGRE 虽然有 8 位的 Entropy 域，但做负载均衡的网络设备如果不认识 NVGRE 协议，仍然根据 “源 IP、目的 IP、四层协议、源端口、目的端口” 的五元组来做 hash，这个 entropy 仍然派不上用场。

VXLAN（Virtual Extensible LAN）的解决方案是：封装层除了 MAC 和 IP 层，再增加一个 UDP 层，使用 UDP 源端口号作为 entropy，UDP 目的端口号作为 VXLAN 协议标识。这样负载均衡设备不需要认识 VXLAN 协议，只要把这个包按照正常的 UDP 五元组做 hash 就行。



VXLAN 的夹层比 GRE 的夹层稍简单，仍然是用 24 位作为 Tenant ID，没有 Entropy 位。添加包头的网络设备或者操作系统虚拟化层一般是把内层 payload 的源端口（source port）复

制一份，作为封装层的 UDP 源端口。由于发起连接的操作系统在选择源端口号时，一般是顺序递增或者随机，而网络设备内部的 hash 算法一般就是 XOR，这样得到的 hash 均匀性一般比较好。

6.2.2.6 STT vs. VXLAN

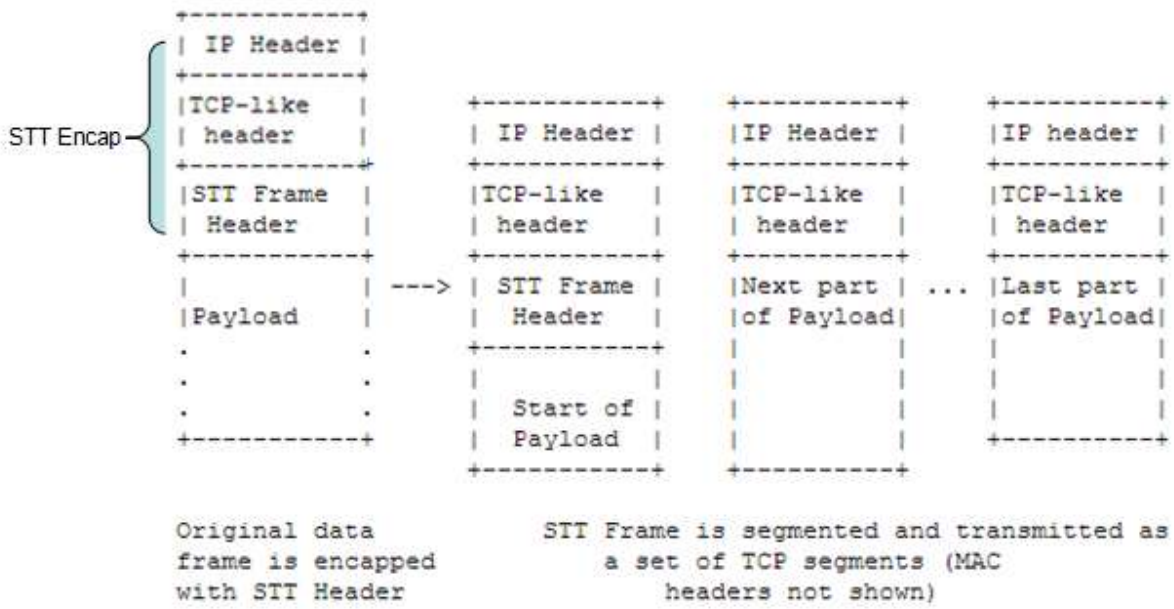
STT（Stateless Transport Tunneling）是 2012 年新提出的、仍然处于草案状态的网络虚拟化协议。STT 与 VXLAN 相比，初看起来只是把 UDP 换成了 TCP，事实上 STT 与 VXLAN 的报文格式大不相同。

STT 为什么要用 TCP 呢？STT 只是借了个 TCP 的壳，根本没有用 TCP 的状态机，更不用说确认、重传和拥塞控制机制了。

STT 想借用的是现代网卡的 LSO（Large Send Offload）和 LRO（Large Receive Offload）机制。

LSO 使得发送端可以生成长达 64KB(甚至更长)的 TCP 包，由网卡硬件把大包的 TCP payload 部分拆开，复制 MAC、IP、TCP 包头，组成可以在二层发送出去的小包（如以太网的 1518 字节，或者开启了 Jumbo Frame 的 9K 字节）。LRO 使得接收端可以把若干个同一 TCP 连接的小包合并成一个大包，再生成一个网卡中断发送给操作系统。

如下图所示，Payload 在被发送出去之前，要加上 STT Frame Header（中间层）和 MAC header、IP header、TCP-like header（封装层）。网卡的 LSO 机制会把这个 TCP 包拆成小块，复制封装层添加到每个小块前面，再发送出去。



我们知道用户-内核态切换和网卡中断是很消耗 CPU 时间的，网络程序（如防火墙、入侵检测系统）在传送大量数据时，如果数据包可以更大，系统的负载就能减轻。

STT 最大的问题是不容易在网络设备上实施针对某个客户（tenant ID）的策略。如上图所示，

对于一个大包，只有第一个小包头部有 STT Frame Header，而后续的小包里没有可以标识客户的信息。如果我要限制某个客户从香港数据中心到芝加哥数据中心的流量不超过 1Gbps，就是不可实现的。如果使用其他网络虚拟化协议，由于每个包里都有标识客户的信息，这种策略只要配置在边界路由器上即可（当然边界路由器需要认识这种协议）。

6.3 虚拟化网络并不完全等同于云网络

从上面云对网络的需求可以看出，并不是网络虚拟化后能够完全解决的。网络虚拟化只是做到将网络资源池化并有效进行分配和管理，这是 SDN 最重要的目标，而云网络则不仅需要做到这一点，还需要网络能够支持虚机的动态迁移，所以针对此，当前的主要解决方案是大二层网络结构，而实现大二层网络结构的技术当前主要有：

1）TRILL（Cisco、华为）

基本思路：重新构建新网络结构，重点需要解决环路问题

2）VXLAN（vmware、OpenStack）

3）NVGRE（Microsoft、OpenStack）

4）STT

基本思路：利用现有的数据中心传统三层网络结构，在不改变当前的网络结构下，利用隧道封装技术实现二层 Over 三层，重点需要解决效率问题

基本组件：

A.提供面向租户的门户用于创建虚拟网络

B.虚拟机管理器（VMM）提供虚机 vm 网络的管理

C.虚拟分布式交换机（vDS）提供虚拟化网络流量转换与封装

D.网络虚拟化网关提供虚拟与物理网络之间的连接

6.3.1 Microsoft 的 NVGRE

微软提出的云网络解决方案，主体思想是希望借助 NVGRE 来满足云数据中心对网络的需求（多租户、多租户隔离、虚机跨子网无缝迁移、非云向云无缝迁移）。

6.3.1.1 微软的云网络解决方案的四大基本组件

A.提供面向租户的门户用于创建虚拟网络：Windows Azure Pack for Windows Server

B.虚拟机管理器（VMM）提供虚机 vm 网络的管理：System Center 2012 R2 虚拟机管理器（VMM）

C.虚拟分布式交换机（vDS）提供虚拟化网络流量转换与封装：Hyper-V 网络虚拟化

D.网络虚拟化网关提供虚拟与物理网络之间的连接：Hyper-V 网络虚拟化网关

6.3.1.2 微软的云网络解决方案的基本概念

A.VM 网络

不同 VM 网络不允许使用相同的 IP 地址前缀，所以 VM 网络实际就是一个三层网络，VM 网络之间必须经过路由转发；

每个 VM 网络，数据中心管理软件会分配一个唯一的路由域 ID（RDID），不过似乎看不出来这个 ID 的意义有多大

B. 虚拟子网

虚拟子网是 VM 网络下的子网，是一个广播域（类似于 VLAN），二层网络，每个虚拟子网必须使用相同的 IP 前缀；

每个虚拟子网会分配一个唯一的虚拟子网 ID（VSID），对应一个租户，这个 ID 在整个数据中心是唯一的

每个 VM 网络可以有多个虚拟子网，每个虚拟子网的 VSID 不同

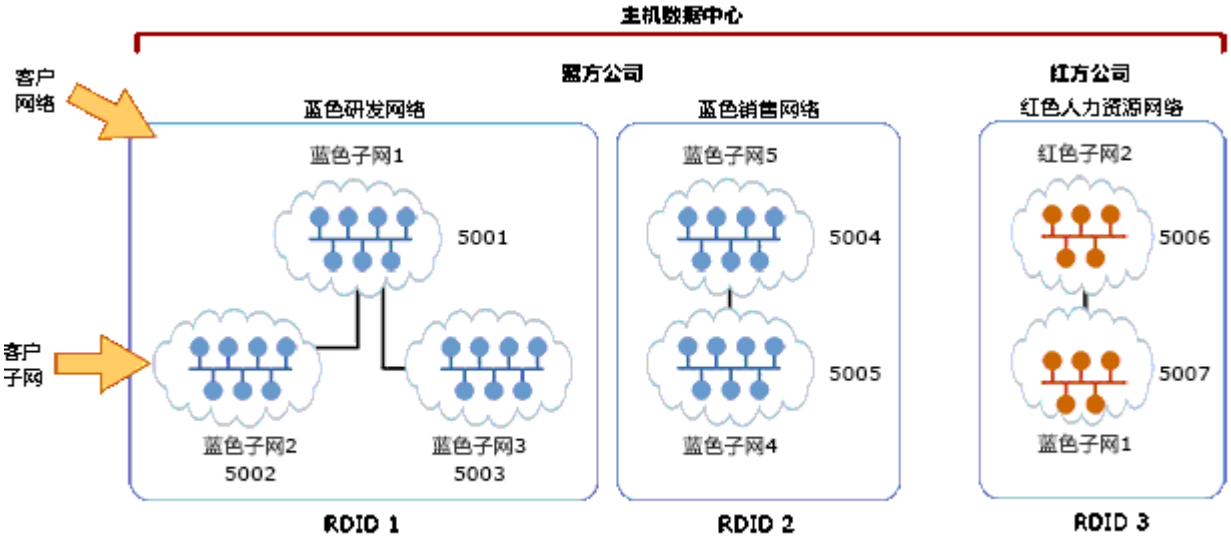


图 2：客户网络和虚拟子网

C. 客户地址（CA）

就是虚机的虚拟网卡上配置的 IP 地址，可由客户访问。

D. 程序地址（PA）

就是物理节点比如物理服务器上的物理网卡上配置的 IP 地址，PA 在物理网络上是可见的，但在虚拟机上不可见。

PA 地址用于“VM 网络”的通信，CA 地址用于“虚拟子网”的通信。

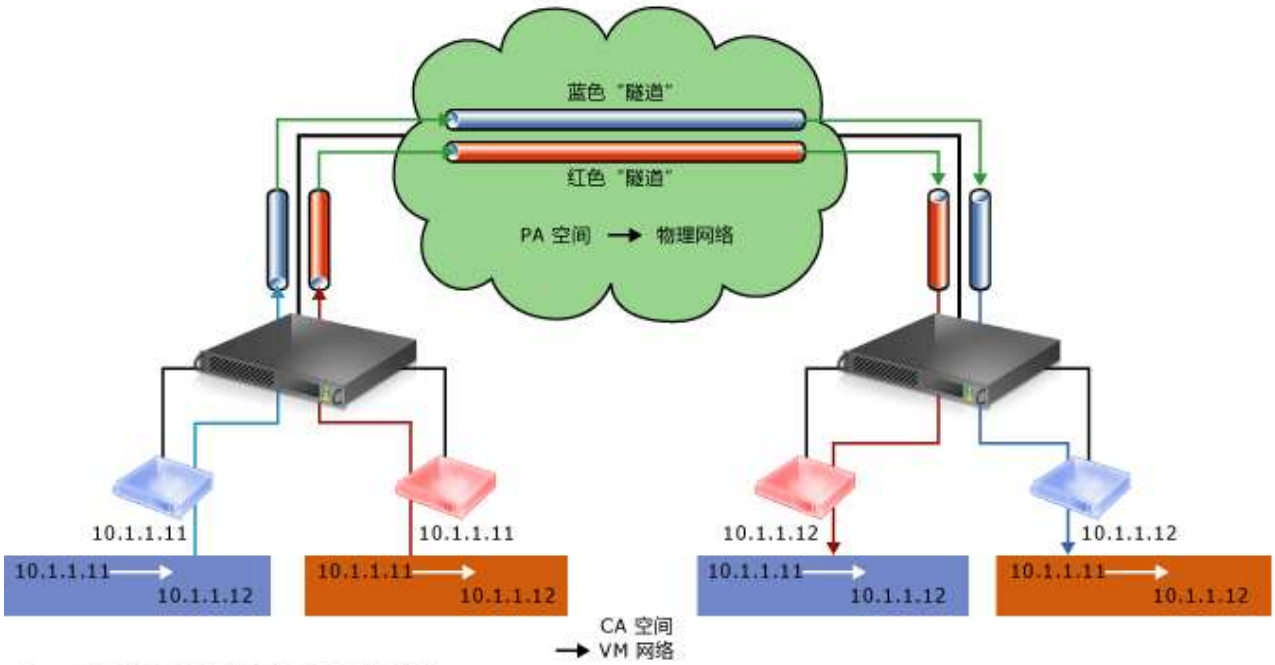


图 6：物理基础结构的网络虚拟化概念图

6.3.1.3 NVGRE 机制实现多租户和大二层网络

1）利用 GRE 头的秘钥字段存储 VSID，通过 VSID 来实现多租户的隔离，VSID 占 24 位，所以租户个数也突破了 VLAN 的 4096 局限；

2）利用 GRE 报文，外层用 PA 地址，内层是 CA 地址，从而使得由 CA 地址组成的虚拟子网能够跨越三层（包括 IP 网段和地域）实现二层的通信，这样虚机可以在隧道内自有迁移，不需要改变 CA 地址，也不需要改变 PA 地址。

3）同一个 VM 网络的不同虚拟子网地址空间可以是重叠的，Hyper-V 的 vDS 会根据 GRE 头中的 VSID 来进行区分。

4）CA 与 PA 的地址映射、GRE 的封装与解封装都在 Hyper-V 的 vDS 上进行的。

5）CA 与 PA 的地址映射、以及 VSID 的分配都是在微软的虚机管理系统（VMM）中设置的

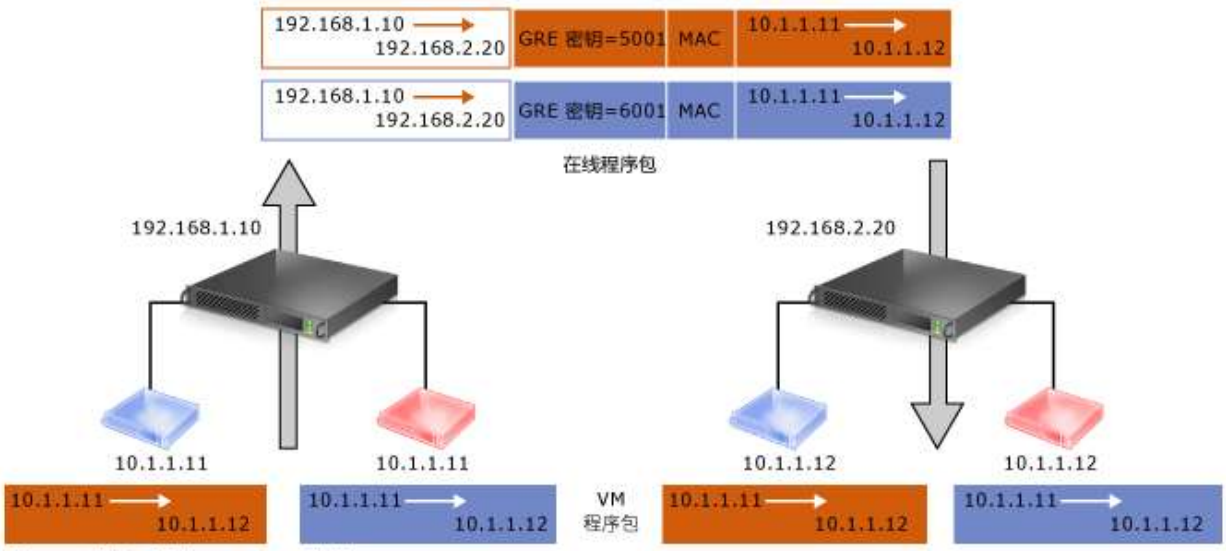


图 7：网络虚拟化 - NVGRE 封装

问题 1：同一个租户（VSID）的虚拟子网内部虚机如果在同一台服务服务器内部，它们之间通信会出物理服务器吗？

不会出物理服务器，会在 Hyper-V 的 vDS 内部进行转发（还会做 GRE 封装吗？不确定）

6.3.1.4 NVGRE 机制实现多租户及大二层的通信过程分析

下图展示了一个两名客户在云数据中心交互的部署案例，其中 CA-PA 关系由 HNV 策略所定义。

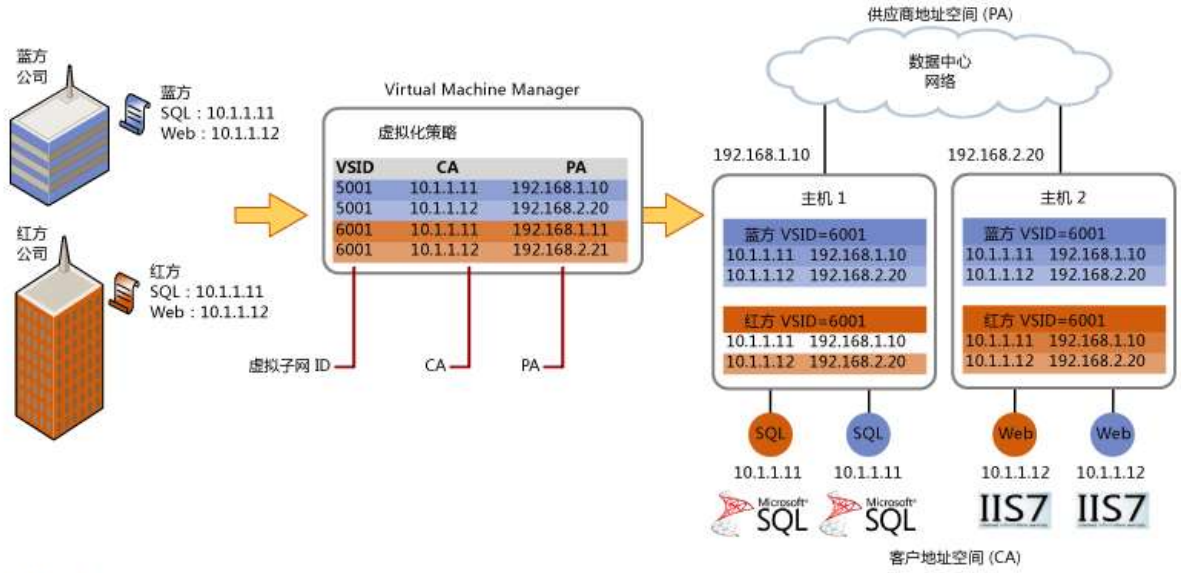


图8：多租户部署案例

需求说明：

在转到托管提供商的共享 IaaS 服务之前：

Contoso 公司运行的是一台 SQL Server（名为 SQL），IP 地址为 10.1.1.11，以及一台 Web 服务器（名为 Web）IP 地址为 10.1.1.12，并用其 SQL Server 处理数据库事务。

Fabrikam 公司运行的是一台 SQL Server（也名为 SQL），IP 地址为 10.1.1.11，以及一台 Web 服务器（也名为 Web），IP 地址为 10.1.1.12，并用其 SQL Server 处理数据库事务。

Contoso 公司和 Fabrikam 公司将其各自的 SQL Server 和 Web 服务器移至同一托管提供商的共享 IaaS 服务，而巧合的是，这些服务器在 Hyper-V 主机 1 中运行 SQL 虚拟机，在 Hyper-V 主机 2 中运行 Web (IIS7) 虚拟机。所有虚拟机都保持其原有的内部网 IP 地址（其 CA）。

策略配置并应用到 Hyper-V 主机 1 和 Hyper-V 主机 2：

Contoso 公司虚拟机的 PA：VSID 为 5001，SQL 为 192.168.1.10，WEB 为 192.168.2.20

Fabrikam 公司虚拟机的 PA：VSID 为 6001，SQL 为 192.168.1.10，WEB 为 192.168.2.20

过程分析：

当 Hyper-V 主机 2 上的 Contoso 公司 Web 虚拟机查询位于 10.1.1.11 的 SQL Server 时

Hyper-V 主机 2，基于其策略设置，将转化以下位置数据包中的地址：

源：10.1.1.12（Contoso 公司 Web 的 CA）

目标：10.1.1.11（Contoso 公司 SQL 的 CA）

封装的数据包含有：

GRE 报头中的 VSID 为：5001

外部源：192.168.2.20（Contoso 公司 Web 的 PA）

外部目标：192.168.1.10（Contoso 公司 SQL 的 PA）

当在 Hyper-V 主机 1 上接收数据包时，基于其策略设置，它会解封以下 NVGRE 数据包：

外部源：192.168.2.20（Contoso 公司 Web 的 PA）

外部目标：192.168.1.10（Contoso 公司 SQL 的 PA）

GRE 报头中的 VSID 为：5001

解封的数据包（原始数据包来自于 Contoso 公司 Web 虚拟机）被传送到 Contoso 公司 SQL 虚拟机：

源：10.1.1.12（Contoso 公司 Web 的 CA）

目标：10.1.1.11（Contoso 公司 SQL 的 CA）

当 Hyper-V 主机 1 上的 Contoso 公司 SQL 虚拟机响应查询时，会发生以下情况：

Hyper-V 主机 1，基于其策略设置，转化以下位置数据包中的地址：

源：10.1.1.11（Contoso 公司 SQL 的 CA）

目标：10.1.1.12（Contoso 公司 Web 的 CA）

数据包封装地址:

GRE 报头中的 VSID 为：5001

外部源：192.168.1.10（Contoso 公司 SQL 的 PA）

外部目标：192.168.2.20（Contoso 公司 Web 的 PA）

当在 Hyper-V 主机 2 处收到数据包时，它将基于策略设置，解封以下数据包:

源：192.168.1.10（Contoso 公司 SQL 的 PA）

目标：192.168.2.20（Contoso 公司 Web 的 PA）

GRE 报头中的 VSID 为：5001

将解封的数据包发送到以下 Contoso 公司 Web 虚拟机：

源：10.1.1.11（Contoso 公司 SQL 的 CA）

目标：10.1.1.12（Contoso 公司 Web 的 CA）

6.3.1.5 网络虚拟化网关

这里所说的网络虚拟化网关是指一个功能组件，可以集成到 TOR 交换机、负载均衡器，也可以是一台单独的设备。

需要虚拟化网关的场景一般有三种：

1）私有云中充当路由器

私有云中，有部分是采用 Hyper-V 实现了虚拟网络，另外一部分是非虚拟网络，在两个网络之间则需要有虚拟化网关来实现路由功能，我理解其实这种场景不需要特殊的网关设备，普通的三层交换机或路由器就可以，只不过微软希望用 windows server 来实现这个网关功能而已。

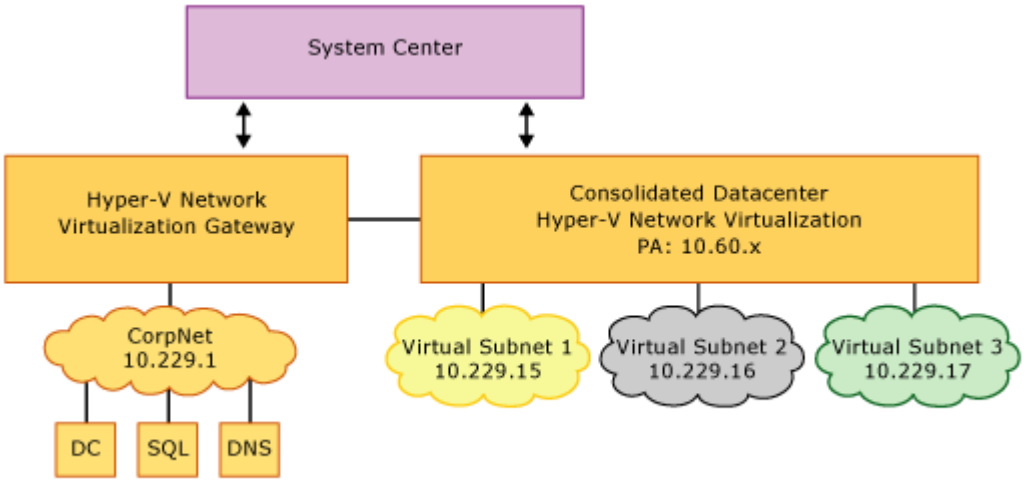


Figure 1: Private cloud deployment

2）混合云中充当 VPN 网关

在这种场景下，如果仅仅是一个租户，其实与普通的 GRE 或 ipsec over GRE 网关没有啥区别，但是如果需要支持多个租户的情况，一种办法是用普通的 VPN 网关，通过多个公网 IP 地址建立多个 VPN 隧道来实现，另外一种办法是，VPN 网关只有一个公网 IP，需要虚拟化网关根据 GRE 头中的 VSID 来区分多租户

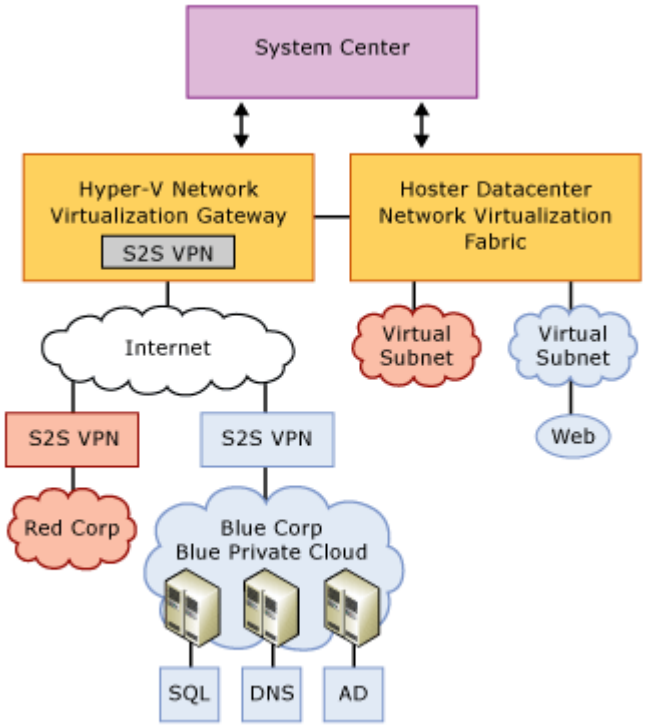


Figure 2: Hybrid cloud deployment

3）网络负载均衡器实现多租户

基本有两种场景：

一种是不同租户的 VIP 不一样，内部的 CA 空间地址可能重叠，这时候需要在网络虚拟化网关上做 NVGRE 封装，将 VIP 与 VSID 对应起来，当报文到达 Hyper-V 的 vDS 后，再解 NVGRE 封装，送到对应租户的 CA 空间。

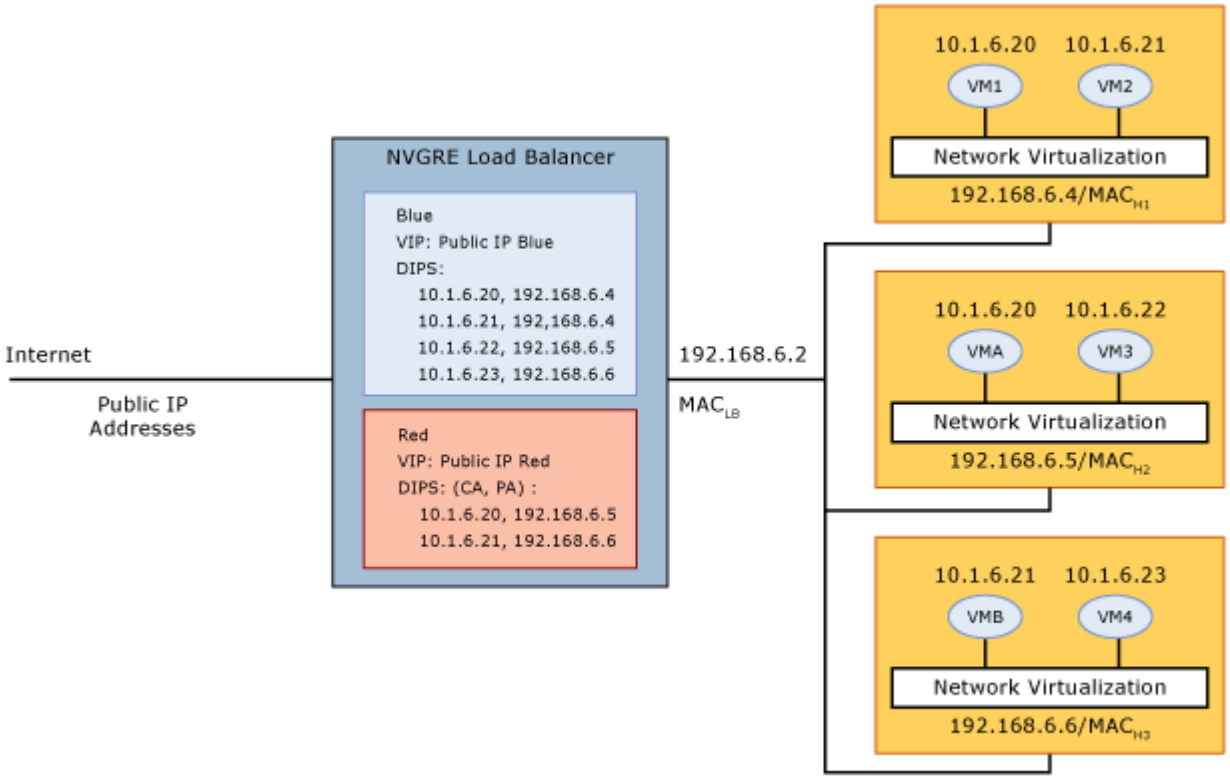


Figure 4: Multi-tenant Load Balancer deployment

另外一种场景是，办公网络与服务器网络都采用 Hyper-V 虚拟化，在服务器前面有负载均衡器，所有租户看到的 VIP 都是一个，租户的子网网络地址空间可能重叠，所以租户的唯一区分就是靠 VSID 来进行，于是需要在负载均衡器里面嵌入网络虚拟化网关，来看 NVGRE 头里面的 VSID 来进行 VIP 与 DIP 的映射关系。

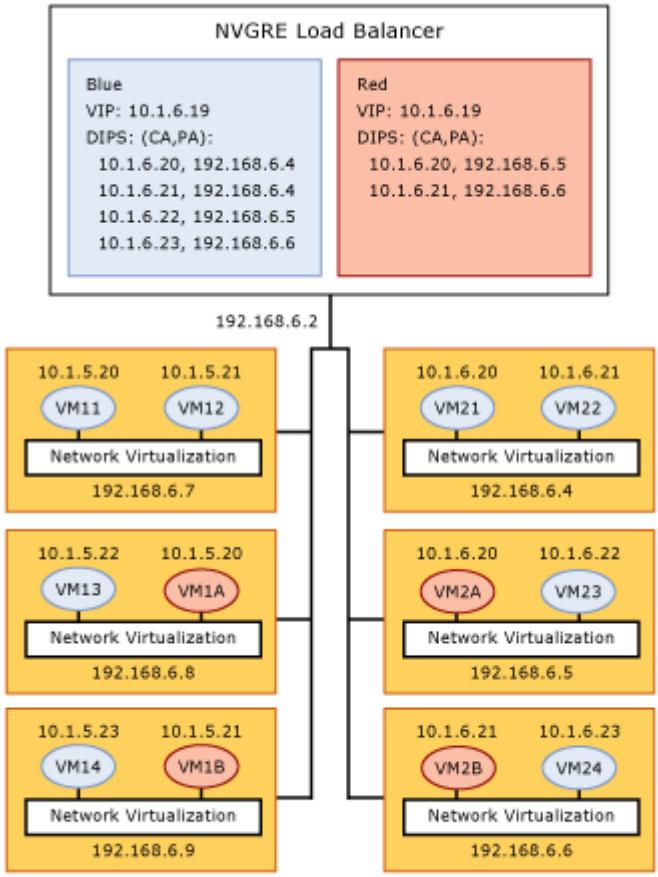


Figure 6: Load Balancer placed internally in the datacenter

The Blue CA VIP and the Red CA VIP on the load balancer both have the PA of 192.168.6.2. The load balancer needs to look at the VSID in the NVGRE packet and then determine the appropriate VIP to DIP mapping. For example Blue virtual machine 11 will send a packet to Blue CA VIP. The NVGRE packet header will have Source IP: 192.168.6.7, Destination IP: 192.168.6.2, GRE Key Field Blue VSID1, Inner Source IP: 10.1.5.20, Inner Destination IP: Blue CA VIP, and the rest of the original packet. The load balancer's policy must match on both the VSID and the VIP because with overlapping CA IP addresses the VIP by itself may not be unique.

6.3.2 vMware 的 VXLAN

vmware 提出的云网络解决方案，主体思想是希望借助 VXLAN 来满足云数据中心对网络的需求（多租户、多租户隔离、虚机跨子网无缝迁移、非云向云无缝迁移）。

6.3.2.1 vmware 的云网络解决方案的四大基本组件

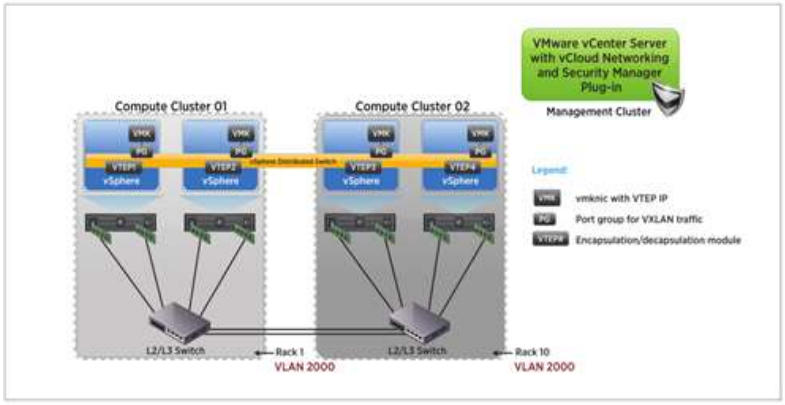
A.提供面向租户的门户用于创建虚拟网络：VMware vCloud Networking and Security Manager

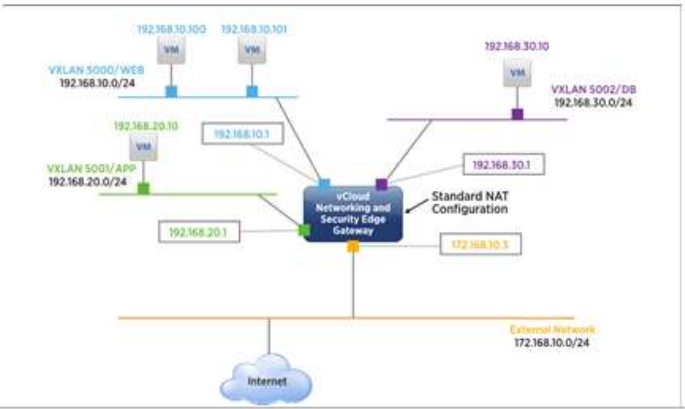
B.虚拟机管理器（VMM）提供虚机 vm 网络的管理：VMware vCloud Networking and Security

Manager

C.虚拟分布式交换机(vDS)提供虚拟化网络流量转换与封装：Virtual Tunnel End Point(VTEP) included in VMware vSphere Distributed Switch

D.网络虚拟化网关提供虚拟与物理网络之间的连接：VMware vCloud Networking and Security Edge Gateway





6.3.2.2 VETP

A virtual tunnel endpoint (VTEP) is configured on every host as part of the VXLAN configuration process. The VTEP consists of the following three modules:

1) vmkernel module - VTEP functionality is part of the VDS and is installed as a VMware Installation Bundle(VIB). This module is responsible for VXLAN data path processing, which includes maintenance of forwarding tables and encapsulation and deencapsulation of packets.

2) vmknic virtual adapter - This adapter is used to carry VXLAN control traffic, which includes response to multicast join, DHCP, and ARP requests.

3) VXLAN port group - This is configured during the initial VXLAN configuration process; includes physical NICs, VLAN information, teaming policy, and so on. These port group parameters dictate how VXLAN traffic is carried in and out of the host VTEP through the physical NICs.

Each vSphere host VTEP is assigned with a unique IP address, which is configured on the vmknic virtual adapter and is used to establish host-to-host communication tunnels and carry VXLAN traffic.

6.3.2.3 网络虚拟化网关

The VMware vCloud Networking and Security Edge™ gateway is a virtual appliance with advanced network services support such as perimeter firewall, DHCP, NAT, VPN, load balancer, and VXLAN gateway function.

The VXLAN gateway function of the vCloud Networking and Security Edge gateway is one of the key components of the VXLAN network design. The vCloud Networking and Security Edge gateway acts as a transparent bridge between the VXLAN and non-VXLAN infrastructure. It is used in the following scenarios:

1) When a virtual machine connected to a logical L2 network must communicate with a physical server or virtual machine running on a host that does not support VXLAN, the traffic is directed through the vCloud Networking and Security Edge gateway.

2) When a virtual machine on one logical L2 network must communicate with a virtual machine on another logical L2 network, the vCloud Networking and Security Edge gateway can provide that connectivity.

6.3.3 OpenStack 不支持 NVGRE，支持 GRE

OpenStack 不支持 NVGRE，在 OVS 项目中支持 GRE。

由于 NVGRE 封装的是二层报文，GRE 封装的是三层报文，所以 OpenStack 如果是用 GRE，则意味着无法实现大二层的网络，只能实现多租户（一个租户对应一个 tunnel）、同一个租户的子网是一个三层子网，无法实现虚机自由迁移。

OVS 实现 GRE 涉及的三个文件：

openvswitch-2.1.2\datapath\vport-gre.c

```
static int gre_rcv(struct sk_buff *skb,
static int gre_send(struct vport *vport, struct sk_buff *skb)
```

openvswitch-2.1.2\datapath\linux\compat\gre.c

openvswitch-2.1.2\datapath\linux\compat\ip_tunnels_core.c

在 OVS 中体现 GRE 封装头格式的函数：

Bits 0–3			4–12	13–15	16–31
C		K	S	Reserved0	Version	Protocol Type
Checksum *(optional)*					Reserved1 *(optional)*
Key *(optional)*					
Sequence Number *(optional)*					

```
openvswitch-2.1.2\datapath\linux\compat\gre.c
struct gre_base_hdr {
        __be16 flags;      //16 位标志位
        __be16 protocol; //Protocol Type 16 位
};
void gre_build_header(struct sk_buff *skb, const struct tnl_ptk_info *tpi,
                      int hdr_len)
{
        struct gre_base_hdr *greh;

        __skb_push(skb, hdr_len);

        greh = (struct gre_base_hdr *)skb->data;
        greh->flags = tnl_flags_to_gre_flags(tpi->flags);
        greh->protocol = tpi->proto;

        if (tpi->flags & (TUNNEL_KEY | TUNNEL_CSUM | TUNNEL_SEQ)) {
                __be32 *ptr = (__be32 *)(((u8 *)greh) + hdr_len - 4);

                if (tpi->flags & TUNNEL_SEQ) {
                        *ptr = tpi->seq;
                        ptr--;
                }
                if (tpi->flags & TUNNEL_KEY) {
                        *ptr = tpi->key;
                        ptr--;
                }
                if (tpi->flags & TUNNEL_CSUM && !is_gre_gso(skb)) {
                        *ptr = 0;
                        *(__sum16 *)ptr = csum_fold(skb_checksum(skb, 0,
                                                    skb->len, 0));
                }
        }
}
```

6.3.4 OpenStack 支持 VXLAN

OpenStack 在 OVS 项目中支持 VXLAN。

OVS 实现 VXLAN 涉及的三个文件：

openvswitch-2.1.2\datapath\vport-vxlan.c

```
static void vxlan_rcv(struct vxlan_sock *vs, struct sk_buff *skb, __be32 vx_vni)
static int vxlan_tnl_send(struct vport *vport, struct sk_buff *skb)
```

openvswitch-2.1.2\datapath\linux\compat\vxlan.c

openvswitch-2.1.2\datapath\linux\compat\ip_tunnels_core.c

根据 VXLAN 协议，会将 GRE 头的 key 字段的前 24 位作为 tenant ID，后 8 位未使用（而 NVGRE 则是将前 24 位作为 tenant ID，后 8 位当做 Entropy 位），从 vxlan 接收报文过程即可看到 VXLAN 的 tenant ID（叫 vni）的操作：

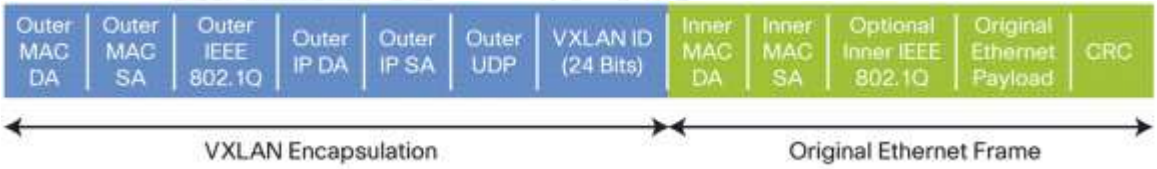
```
static void vxlan_rcv(struct vxlan_sock *vs, struct sk_buff *skb, __be32 vx_vni)
{
        ...

        /* Save outer tunnel values */
        iph = ip_hdr(skb);
        key = cpu_to_be64(ntohl(vx_vni) >> 8);
        ovs_flow_tun_key_init(&tun_key, iph, key, TUNNEL_KEY);

        ovs_vport_receive(vport, skb, &tun_key);
}
```

```
static int vxlan_tnl_send(struct vport *vport, struct sk_buff *skb)
{
      ...
        err = vxlan_xmit_skb(vxlan_port->vs, rt, skb,
                             saddr, OVS_CB(skb)->tun_key->ipv4_dst,
                             OVS_CB(skb)->tun_key->ipv4_tos,
                             OVS_CB(skb)->tun_key->ipv4_ttl, df,
                             src_port, dst_port,
                             htonl(be64_to_cpu(OVS_CB(skb)->tun_key->tun_id) << 8));
}
```

VXLAN 的封装格式如下图：



体现在代码中：

```
#define VXLAN_HLEN (sizeof(struct udphdr) + sizeof(struct vxlanhdr))
/* VXLAN protocol header */
struct vxlanhdr {
        __be32 vx_flags;
        __be32 vx_vni; //这里虽然定义占用32位，但实际上在接受或发送的适合会做左移或右移8位
};
```

7 FWaaS

The Firewall-as-a-Service (FWaaS) plug-in adds perimeter firewall management to Networking. FWaaS uses iptables to apply firewall policy to all Networking routers within a project. FWaaS supports one firewall policy and logical firewall instance per project.

7.1 FWaaS 应用场景

FWaaS 即 FW 作为服务，涉及到

1）租户的识别

2）租户私有网络的构建（VPC）

3）租户虚机南北向流量防护

4）租户虚机东向西流量防护

5）租户网络安全防护

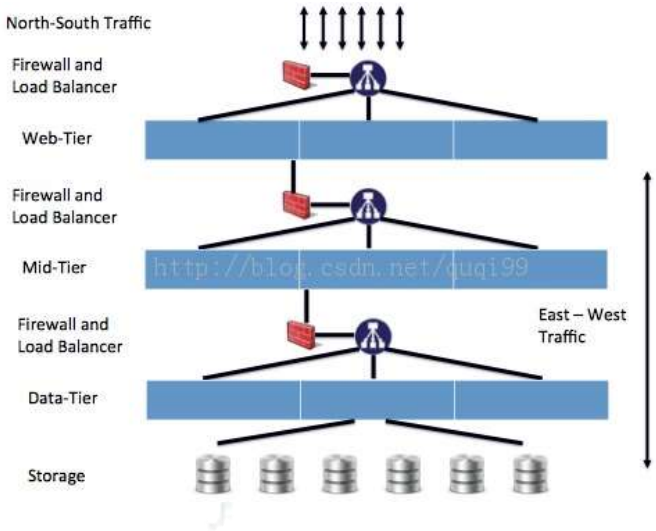
6）租户应用安全防护

7）租户数据安全防护

8）租户病毒防护

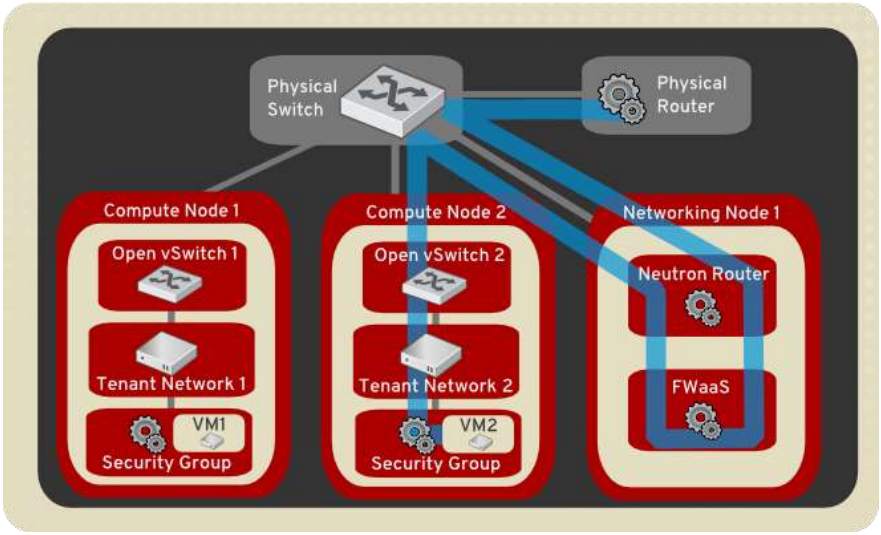
9）租户远程接入防护

如下图所示。

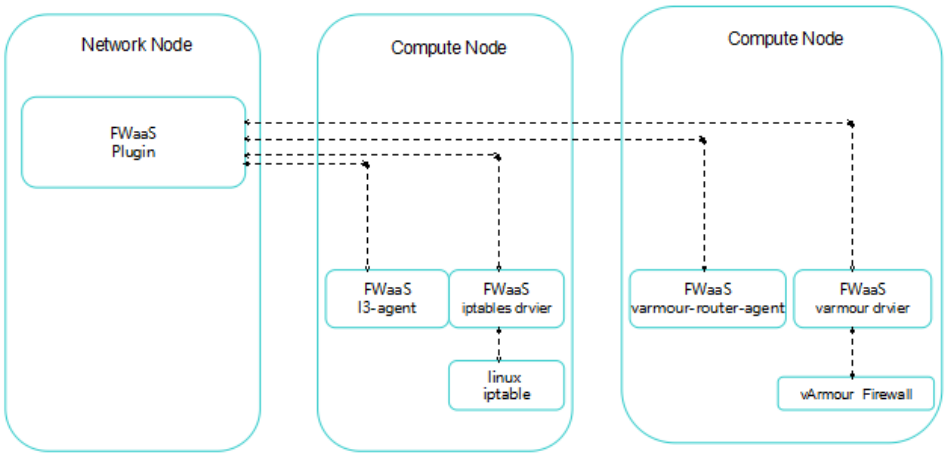


7.2 FWaaS in Openstack(Neutron)

FWaaS 在 Openstack 基础上实现的框架如下， illustrates the flow of ingress and egress traffic for the VM2 instance：



7.2.1 1 级分解



FWaaS Plugin：neutron\services\firewall\fwaas_plugin.py

FWaaS l3-agent：neutron\services\firewall\agents\l3reference\firewall_l3_agent.py

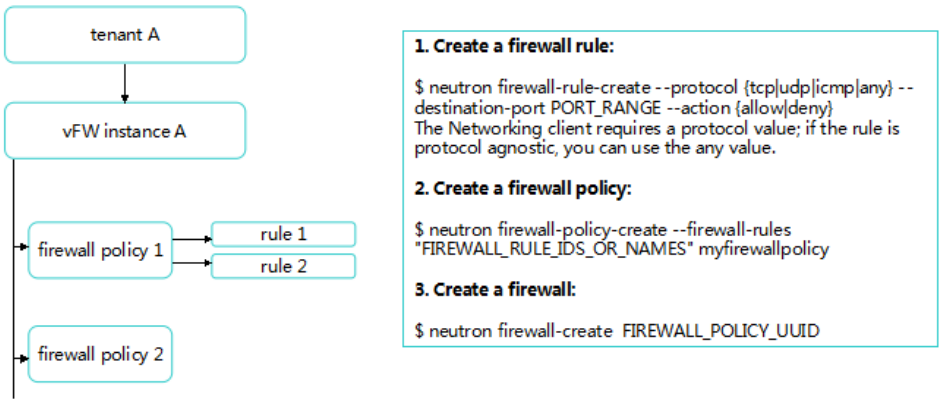
FWaaS iptables driver：neutron\services\firewall\drivers\linux\iptables_fwaas.py

FWaaS varmour-router-agent：neutron\services\firewall\agents\varmour\varmour_router.py

先不说 vArmour 是怎么实现 FWaaS 的，看 Linux 是如何实现的。

FWaaS L3-agent 实际是利用 linux 的 namespace 机制实现的逻辑路由器；FWaaS iptables driver 是利用 linux 的 iptables 实现安全策略与规则。

FWaaS 租户、虚机、安全策略、安全规则的关系：



7.2.2 2 级分解--Linux 如何实现 FWaaS

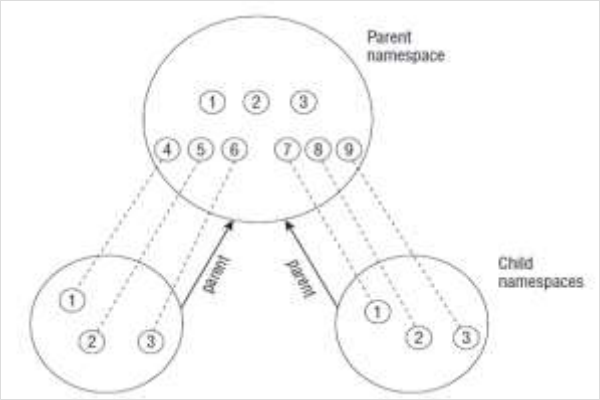
7.2.2.1 Linux 如何实现多租户

FWaaS L3-agent 实际是利用 linux 的 namespace 机制实现的逻辑路由器。运行在网关节点上。Linux 内核中的 Namespace 提供了一个轻量级的基于系统调用层面的虚拟化解决方案。相比传统的使用 VMWare, QEMU, Xen, KVM, Hurd 的虚拟，基于 namespace 的轻量级虚拟具有易使用，易管理，无需硬件虚拟化支持，低成本等优点。该机制类似于 Solaris 中的 zone 或 FreeBSD 中的 jail。

LXC（Linux containers）就是利用这一特性实现了资源的隔离。LXC 正是通过在 clone flag 为进程创建一个有独立 PID，IPC，FS，Network，UTS 空间的 container。一个 container 就是一个虚拟的运行环境，对 container 里的进程是透明的，它会以为自己是直接在一个系统上运行的。一个 container 就像传统虚拟化技术里面的一台安装了 OS 的虚拟机，但是开销更小，部署更为便捷。

Linux Namespaces 机制本身就是为了实现 container based virtualizaiton 开发的。它提供了一套轻量级、高效率的系统资源隔离方案，远比传统的虚拟化技术开销小，不过它也不是完美的，它为内核的开发带来了更多的复杂性，它在隔离性和容错性上跟传统的虚拟化技术比也还有差距。

namespace 还拥有层次关系。图 3 中，一个 parent namespace 下有两个 child namespace。parent namespace 和它的两个 child namespace 都有三个进程号为 1,2,3 的进程，同时 child namespace 的每个进程被映射到了 parent namespace 中的 4,5,6,7,8,9。虽然只有 9 个进程，但需要 15 个进程号来表示它们。



7.2.2.2 FWaaS L3-agent 如何实现的

FWaaS L3-agent 实际是利用 linux 的 namespace 机制实现的逻辑路由器。每个租户至少一个。

neutron\services\firewall\drivers\linux\iptables_fwaas.py:

def create_firewall(self, apply_list, firewall): //apply_list 就是指 namespace（某个租户）

def delete_firewall(self, apply_list, firewall):

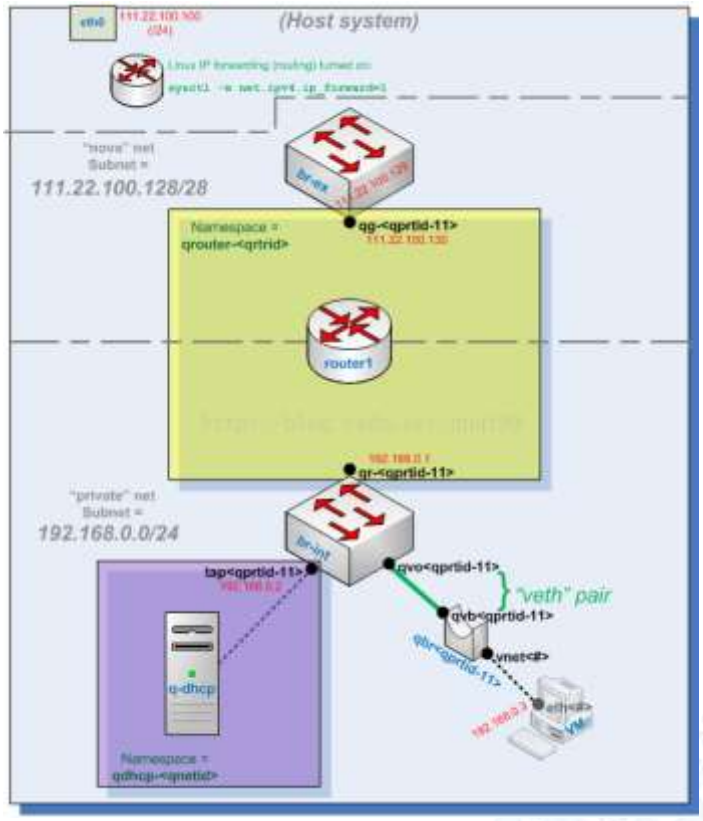
def update_firewall(self, apply_list, firewall):

def apply_default_policy(self, apply_list, firewall):

def _setup_firewall(self, apply_list, firewall):

7.2.2.3 FWaaS 的租户私有网络 VPC 及安全防护

下图是一个节点上的内部逻辑图示，利用 linux namespace 机制实现了逻辑路由器和逻辑 dhcp 服务器，利用 iptabales 实现了 VM 的南北向流量控制。



8 vMware

vMware 作为私有云整体解决方案的软件大佬，在最近也积极参与到 OpenStack 的生态圈之中，所以研究 vMware 如何融入到 OpenStack 对于有志于国内的云解决方案提供商具有较大的参考价值。

网上有很多声音在探讨 vMware 为什么这么积极的参与到 OpenStack 的阵营之中呢？有人说 vMware 布局了一个阴谋，也有人说 vMware 更多从实用的角度出发，希望将自己倡导的“软件定义数据中心”与 OpenStack 擅长“云管理和自动化部署”相结合，使得 vMware 老客户继续使用 vShpere，新客户接受自己的 NSX 等 SDN 概念和产品，从而扩大自己在云计算领域的占有率。

8.1 vMware 与 OpenStack 融合方案探讨

vMware 与 OpenStack 在“云”的理念方面有着一个本质在于运行在云上的应用：

vMware 认为运行在云上的应用的可靠性和可用性可以通过底层的虚拟化技术比如 vMotion、HA 来保障，而 OpenStack 在一开始就认为云应用本身应该由应用本身来保障，有很多的应用本身就支持集群和分布式的运行，所以 OpenStack 在虚机层面没有 FT 等类似的功能，包括后续也没有计划在这方面加大投入。

vMware 与 OpenStack 的融合方案有：

1）vMware 为 OpenStack 贡献一些关键的 Plugin、agent、drivers 等，从而使得 OpenStack 可以选择这些 vMware 贡献的组件，从而扩大 vMware 在 OpenStack 生态圈的影响力

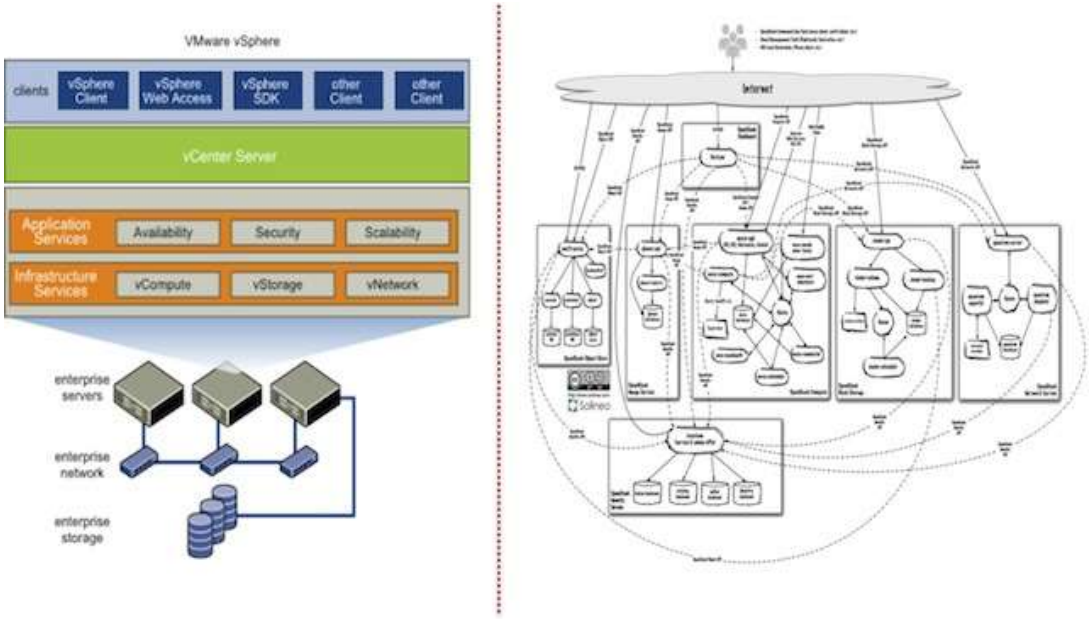
2）孤岛型解决方案

3）异构虚拟机集成管理方案

4）类似 Rackspace 的混合解决方案

8.1.1 孤岛型解决方案

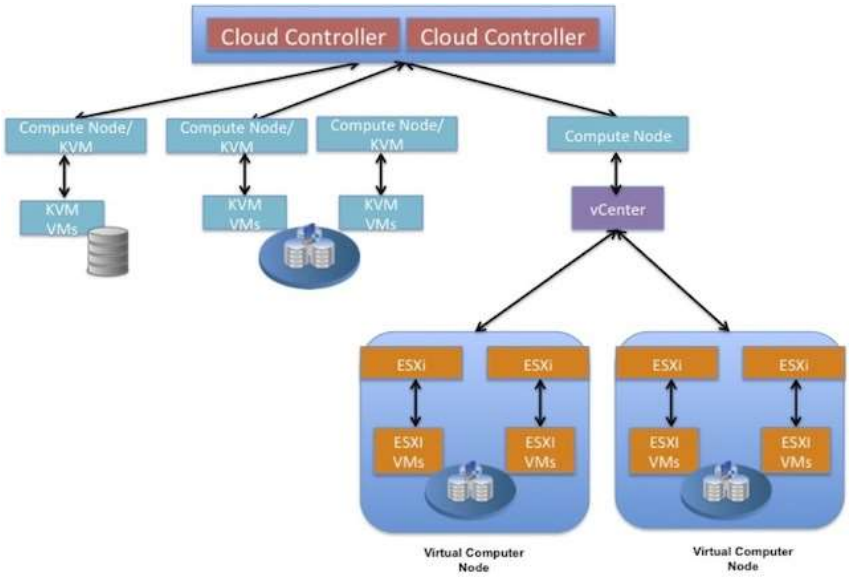
孤岛型架构是用户选择最多的。通常，这种方案涉及到保存在 vSphere 上的现有遗留传统应用和在独立OpenStack云上建立新应用的抉择。虽然这是最最无痛的融合OpenStack的解决方案，但是它保持了 IT 基础架构的孤岛劣势，并且增加了运维和复杂性，通常我们需要两个独立团队去维护这两套独立系统，这也会带来额外的开销。



8.1.2 异构虚机集成管理解决方案

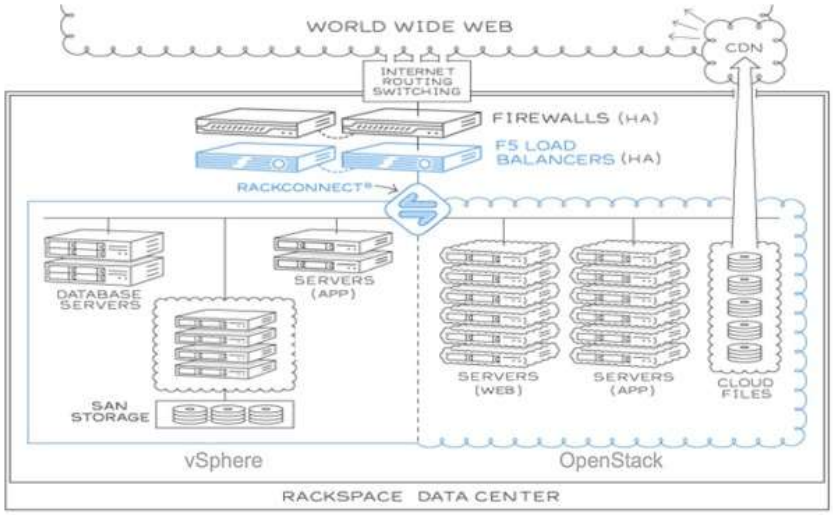
基于 VMware 已完成的工作将 vSphere 与 OpenStack 相集成。这种方案类似于孤岛型解决方案，传统型应用仍然运行在 vSphere 之上，而新的下一代应用则运行在新的虚拟机管理器之上，如 KVM 或在 XEN。在这种情况下，OpenStack 成为多虚拟机管理器的控制平台，它将允许新创建的应用被分配到最适合他们的虚拟机管理平台之上。

这种架构的主要缺点是 vSphere 与 OpenStack 整合这种方案非常新，这带来了很多问题，比如两平台的整合边缘过于粗糙仍需改进，比如平台的资源如何调度等问题仍然需要解决。



8.1.3 混合解决方案

Rackspace 给出的 OpenStack 与 vSphere 混合解决方案。在分离管理平台这方面，这种架构非常像孤岛型架构，此外，它还保证不同类型应用可以被部署到合适的虚拟机管理平台之上。这里的目标是保持架构的独立,整合两个环境的运维团队使他们可以共同工作来打造一个集成的平台。这其中的一个关键是应用技术，如 Rackspace 的 RackConnect 把这些基础架构连接起来,使每个架构都可以与其他架构协同工作。这里举一个例子,一个应用运行在基于 OpenStack 的 Rackspace 私有云之上，通过 RackConnect 与运行在 vSphere 集群上的 Oracle 数据库相连接。



8.2 vMware 对 OpenStack 在计算虚拟化方面的贡献

vSphere 延伸对 Nova 的支持，并加入了新的功能，其中包括配置-驱动程序、卷启动、更多磁

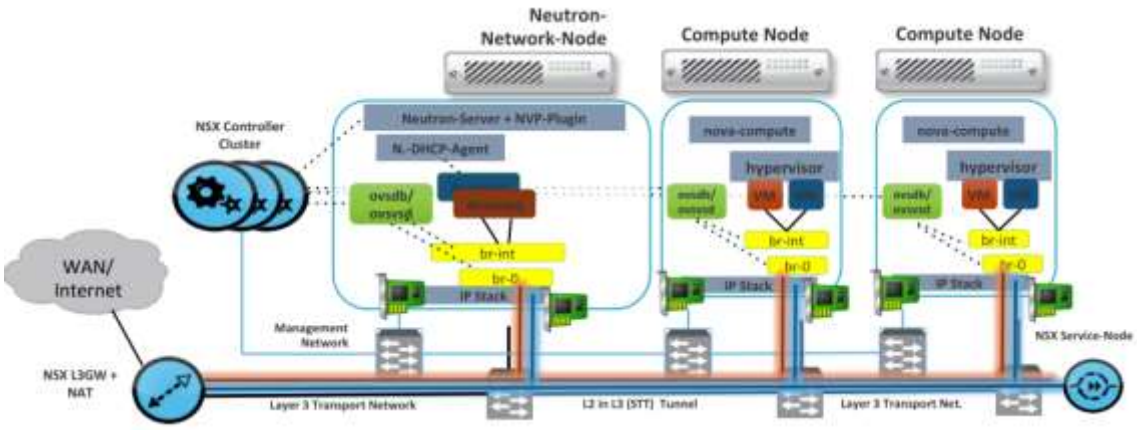
盘类型、VNC 控制台、实例快照等。

8.3 vMware 对 OpenStack 在网络虚拟化方面的贡献

VMware 延续了在 Neutron 网络项目中的领导地位。Neutron 可支持 VMwareNSX 网络虚拟化，其中包括 VMware vSphere 5.5 中的内核中虚拟交换机、负载均衡即服务（LBaaS）和全状态防火墙（FWaaS）。下图是 vmware 的 NSX controller 融入到 Neutron 框架中涉及的方面：

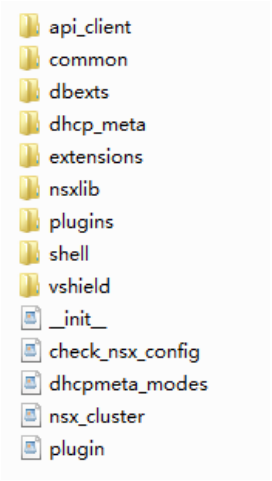
NVP-Plugin、OVS agent 和 OVS。

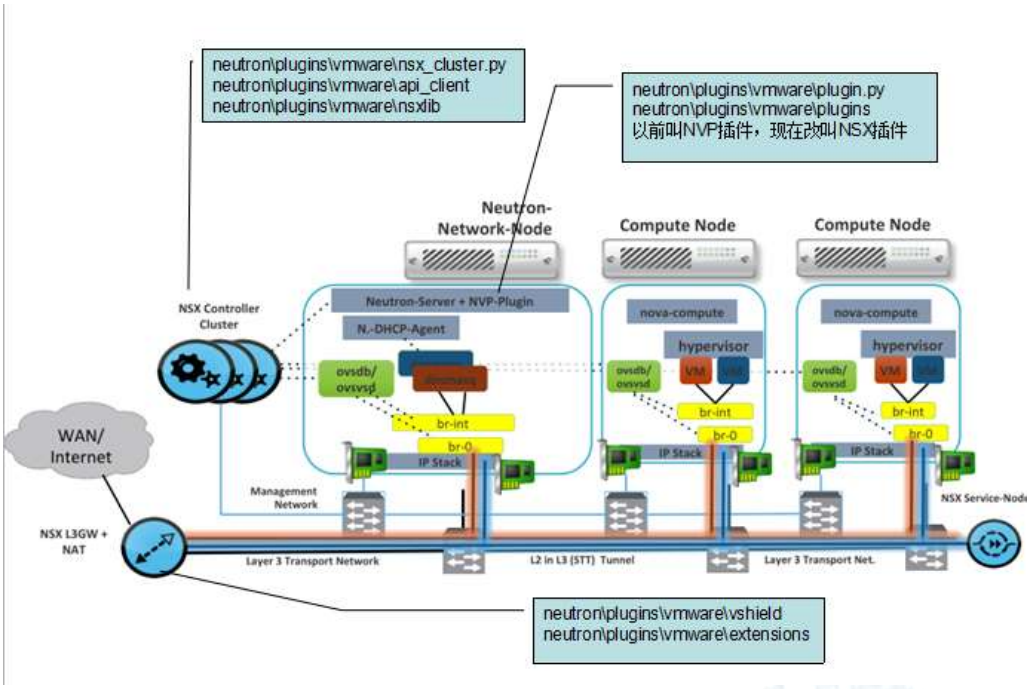
NVP-Plugin 运行在网络节点上，OVS agent 和 OVS 本身运行在计算节点上。



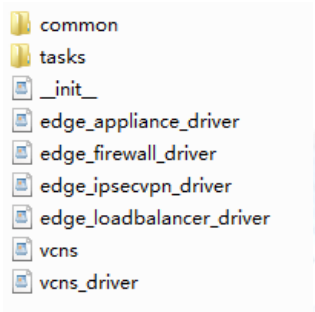
- Centralized scale-out controller cluster controls all Open vSwitches in all Compute- and Network Nodes. It configures the tunnel interfaces and programs the flow tables of OVS
- NSX L3 Gateway Service (scale-out) is taking over the L3 routing and NAT functions
- NSX Service-Node relieves the Compute Nodes from the task of replicating broadcast, unknown unicast and multicast traffic sourced by VMs
- Security-Groups are implemented natively in OVS, instead of iptables/ebtables

neutron\plugins\vmware：





neutron\plugins\vmware\vshield：



vCloud Network and Security：

neutron\plugins\vmware\vshield\vcns.py--->neutron\plugins\vmware\vshield\vcns_driver.py-->

edge_firewall_driver.py

edge_loadbalancer_driver.py

edge_ipsecvpn_driver.py

edge_appliance_driver.py

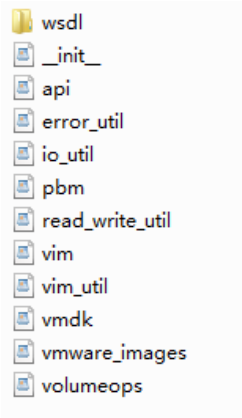
```
neutron\plugins\vmware\vshield\vcns_driver.py:
from neutron.plugins.vmware.vshield import edge_appliance_driver
from neutron.plugins.vmware.vshield import edge_firewall_driver
from neutron.plugins.vmware.vshield import edge_ipsecvpn_driver
from neutron.plugins.vmware.vshield import edge_loadbalancer_driver
from neutron.plugins.vmware.vshield.tasks import tasks
from neutron.plugins.vmware.vshield import vcns

class VcnsDriver(edge_appliance_driver.EdgeApplianceDriver,
                 edge_firewall_driver.EdgeFirewallDriver,
                 edge_loadbalancer_driver.EdgeLbDriver,
                 edge_ipsecvpn_driver.EdgeIPsecVpnDriver):
```

8.4 vMware 对 OpenStack 在存储虚拟化方面的贡献

为 Cinder 提供 VMware 贡献的 VMDK 驱动程序,确保 OpenStack 云可以充分利用 VMware 新的软件定义存储解决方案 VMware Virtual SAN，从本地链接的磁盘以及所有支持 vSphere 的存储阵列中创建一个分布式共享数据存储。

cinder\volume\drivers\vmware



volumeops.py:卷相关操作方法

vmware_images.py:虚机镜像相关管理

vmdk.py:Volume driver for VMware vCenter/ESX managed datastores.

vim.py:Classes for making VMware VI SOAP calls

pbm.py:Class for making VMware PBM SOAP calls.used for storage policy based placement of volumes

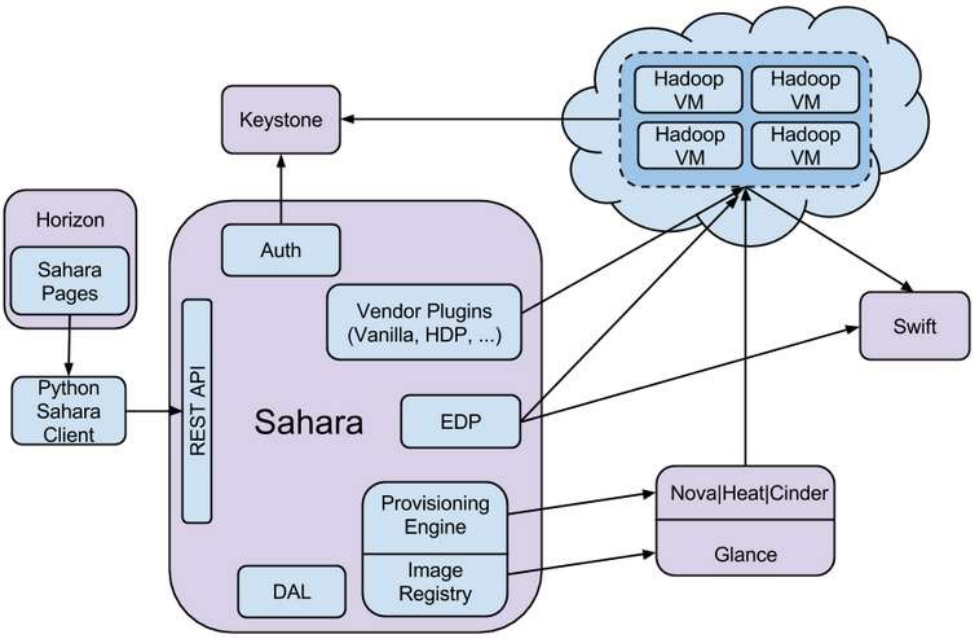
api.py:Session and API call management for VMware ESX/VC server.Provides abstraction over cinder.volume.drivers.vmware.vim.Vim SOAP calls.

9 Sahara & Hadoop

The Sahara project provides a simple means to provision a Hadoop cluster on top of OpenStack. It's ex Savanna project, renamed due to the potential trademark issues.

9.1 Sahara 体系结构

下面是 Sahara 的体系结构图：



The Sahara architecture consists of several components:

- Auth component - responsible for client authentication & authorization, communicates with Keystone
- DAL - Data Access Layer, persists internal models in DB
- Provisioning Engine - component responsible for communication with Nova, Heat, Cinder and Glance
- Vendor Plugins - pluggable mechanism responsible for configuring and launching Hadoop on provisioned VMs; existing management solutions like Apache Ambari and Cloudera Management Console could be utilized for that matter
- EDP - *Elastic Data Processing (EDP)* responsible for scheduling and managing Hadoop jobs on clusters provisioned by Sahara
- REST API - exposes Sahara functionality via REST
- Python Sahara Client - similar to other OpenStack components Sahara has its own python client
- Sahara pages - GUI for the Sahara is located on Horizon

Auth：sahara\middleware\auth_valid.py

DAL:sahara\db

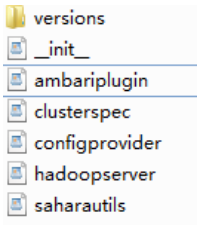
Provisioning Engine:

sahara\topology：计算、存储的拓扑结构发现与创建

sahara\swift：与 Swift 对象存储模块的交互

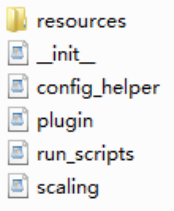
Vendor Plugins:sahara\plugins

sahara\plugins\hdp



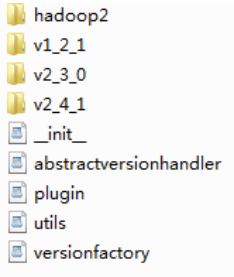
The Apache Ambari project is aimed at making Hadoop management simpler by developing software for provisioning, managing, and monitoring Apache Hadoop cluster.

sahara\plugins\spark



Spark 是 UC Berkeley AMP lab 所开源的类 Hadoop MapReduce 的通用的并行计算框架，Spark 基于 map reduce 算法实现的分布式计算，拥有 Hadoop MapReduce 所具有的优点；但不同于 MapReduce 的是 Job 中间输出结果可以保存在内存中，从而不再需要读写 HDFS，因此 Spark 能更好地适用于数据挖掘与机器学习等需要迭代的 map reduce 的算法。

sahara\plugins\vanilla

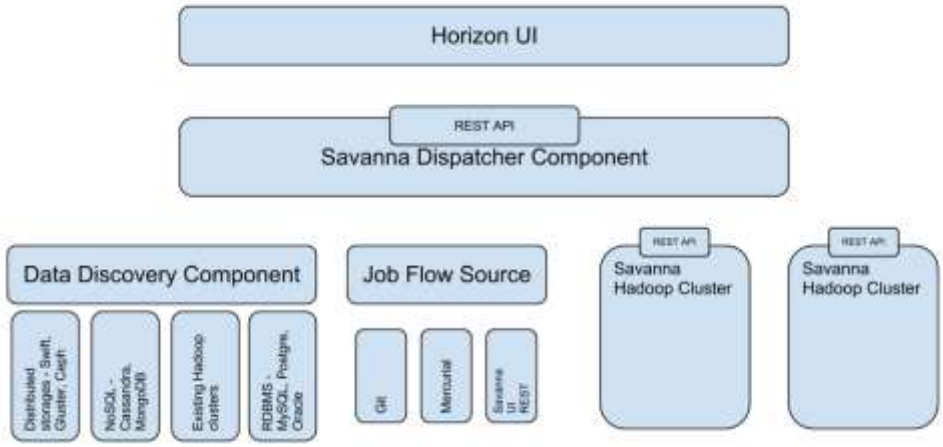


The Vanilla Hadoop plugin is a reference plugin implementation that enables the user to launch Apache Hadoop clusters without any management consoles. This plugin is based on images with pre-installed instances of Apache Hadoop. Almost all Hadoop cluster topologies have been supported by the Vanilla plugin, with ability to manually scale existing clusters and use OpenStack Swift as input/output for MapReduce jobs.

EDP：sahara\service\edp

9.2 EDP 运行机理

Sahara EDP 的运行结构组件如下：



Horizon UI Component

Integration into OpenStack Dashboard - Horizon. It should provide instruments for job

creation, monitoring etc. Hue already provides part of this functionality: submit jobs (Java, Hive, Pig, MapReduce), view job status and output.

源代码位置：sahara\service\edp\api.py

Dispatcher Component

This component is responsible for provisioning a new cluster, scheduling job on new or existing cluster, resizing cluster and gathering information from clusters about current jobs and utilization. Also, it should provide information to help to make a right decision where to schedule job, create a new cluster or use existing one. For example, current loads on clusters, their proximity to the data location etc.

源代码位置：

sahara\service\edp\spark\engine.py

sahara\service\edp\job_manager.py

Data discovery component

EDP can have several sources of data for processing. Data can be pulled from Swift, GlusterFS or NoSQL database like Cassandra or HBase. To provide an unified access to this data we'll introduce a component responsible for discovering data location and providing right configuration for Hadoop cluster. It should have a pluggable system.

源代码位置：

sahara\service\edp\binary_retrievers\dispatch.py

sahara\service\edp\binary_retrievers\internal_swift.py

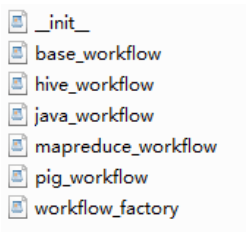
sahara\service\edp\binary_retrievers\sahara_db.py

Cluster Level Coordination Component

Expose information about jobs on a specific cluster. Possible this component should be represent by existing Hadoop projects Hue and Oozie.

源代码位置：sahara\service\edp\oozie

sahara\service\edp\oozie\workflow_creator



Oozie is a workflow scheduler system to manage Apache Hadoop jobs.

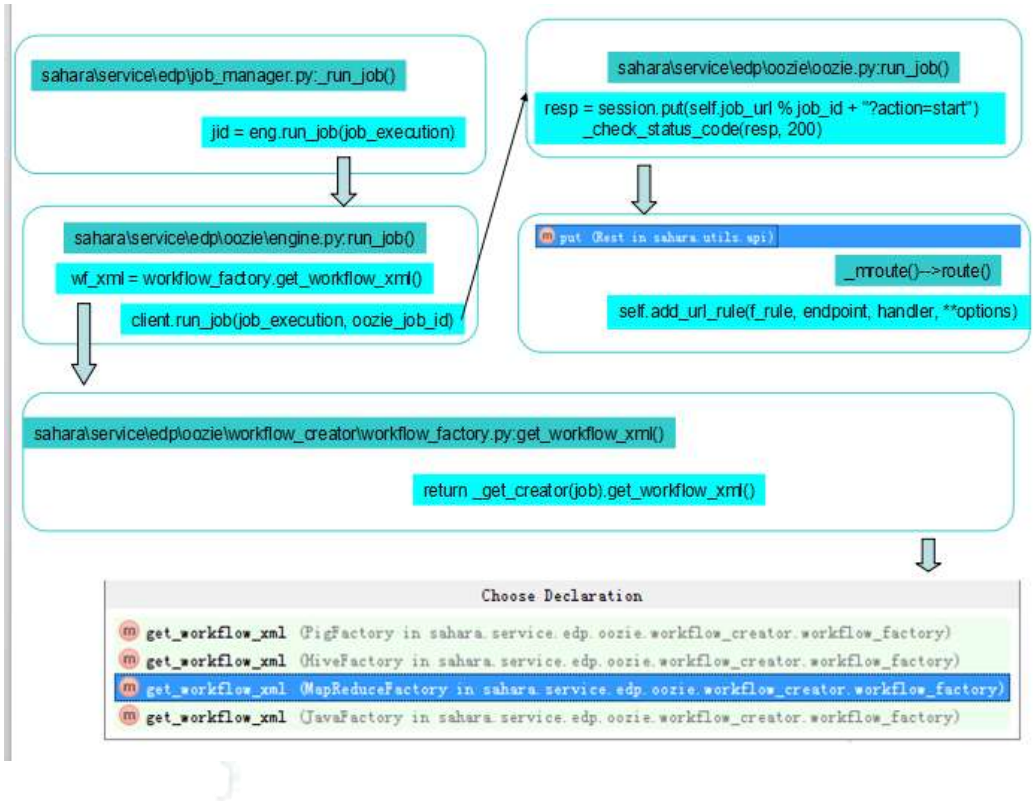
Oozie Workflow jobs are Directed Acyclical Graphs (DAGs) of actions.

Oozie Coordinator jobs are recurrent Oozie Workflow jobs triggered by time (frequency) and data availabilty.

Oozie is integrated with the rest of the Hadoop stack supporting several types of Hadoop jobs out of the box (such as Java map-reduce, Streaming map-reduce, Pig, Hive, Sqoop and Distcp) as well as system specific jobs (such as Java programs and shell scripts).

Oozie is a scalable, reliable and extensible system.

下面是 EDP 运行一个 job 的运行的调用过程：



10 Murano

Murano Project introduces an application catalog, which allows application developers and cloud administrators to publish various cloud-ready applications in a browsable categorized catalog, which may be used by the cloud users (including the inexperienced ones) to pick-up the needed applications and services and composes the reliable environments out of them in a "push-the-button" manner. The key goal is to provide UI and API which allows to compose and deploy composite environments on the Application abstraction level and then manage their lifecycle. The Service should be able to orchestrate complex circular dependent cases in order to setup complete environments with many dependant applications and services. However, the actual deployment itself

will be done by the existing software orchestration tools (such as Heat), while the Murano project will become an integration point for various applications and services.
（待跟踪）。

11 KeyStone

Keystone is the identity service used by OpenStack for authentication (authN) and high-level authorization (authZ). It currently supports token-based authN and user-service authorization. It has recently been rearchitected to allow for expansion to support proxying external services and AuthN/AuthZ mechanisms such as oAuth, SAML and openID in future versions.
（重点关注 tenant 的实现，待分析）。

12 OpenStack 部署管理

Fuel 是 Mirantis 公司的一个为 openstack 端到端”一键部署“设计的工具，其功能含盖自动的 PXE 方式的操作系统安装，DHCP 服务，Orchestration 服务 和 puppet 配置管理相关服务等，此外还有 openstack 关键业务健康检查和 log 实时查看等非常好用的服务。
类似的工具：

Cobbler is a Linux installation server that allows for rapid setup of network installation environments powered by RedHat. It glues together and automates many associated Linux tasks so you do not have to hop between lots of various commands and applications when rolling out new systems, and, in some cases, changing existing ones. It can help with installation, DNS, DHCP, package updates, power management, configuration management orchestration, and much more.

xCAT (Extreme Cloud Administration Toolkit) is open-source distributed computing management software developed by IBM, used for the deployment and administration of Linux or AIX based clusters.

Packstack is utility to install OpenStack on Red Hat based operating system.

Ironic is an Incubated OpenStack project which aims to provision bare metal (as opposed to virtual) machines by leveraging common technologies such as PXE boot and IPMI to cover a wide range of hardware, while supporting pluggable drivers to allow vendor-specific functionality to be added.
（待分析）。

13 总结

本文档还在不断的更新和完善之中，暂未总结。

14 附录

14.1 本子系统用到的缩写词、定义和术语

VETP：A virtual tunnel endpoint

SDN：Softeware Defined Network

14.2 参考资料

a. 软件包 cinder-2014.1.1
b. 软件包 flashcache-3.1.2
c. 软件包 neutron-2014.2.b2
d. 软件包 sahara-2014.2.b2
e. 软件包 openvswitch-2.1.2
f. 《VMware-VXLAN-Deployment-Guide.pdf》
g. 《分布式文件系统_-_RADOS.pdf》
h. 微软官方网站
i. vMware 官方网站
j. OpenStack 维基：https://wiki.openstack.org/
k. OpenStack 官网网站：http://www.openstack.org
l. OpenStack 中文社区：http://http://www.openstack.cn
m. Ceph 官方网站：http://ceph.com/