

Table of Contents

前言	1.1
概述	1.2
基本概念	1.3
GRE 模式	1.4
计算节点	1.4.1
网络节点	1.4.2
VLAN 模式	1.5
计算节点	1.5.1
网络节点	1.5.2
VXLAN 模式	1.6
计算节点	1.6.1
br-int	1.6.1.1
br-tun	1.6.1.2
网络节点	1.6.2
br-tun	1.6.2.1
br-int	1.6.2.2
br-ex	1.6.2.3
网络命名空间	1.7
DHCP 服务	1.7.1
路由服务	1.7.2
安全组	1.8
INPUT	1.8.1
OUTPUT	1.8.2
FORWARD	1.8.3
整体逻辑	1.8.4
快速查找安全组规则	1.8.5
其它	1.8.6

LBaaS (负载均衡即服务)	1.9
典型场景	1.9.1
实现细节	1.9.2
其它问题	1.9.3
FWaaS (防火墙即服务)	1.10
典型场景	1.10.1
实现细节	1.10.2
其它问题	1.10.3
DVR (分布式路由)	1.11
典型场景	1.11.1
网络节点	1.11.2
计算节点	1.11.3
配置	1.11.4
工作流程	1.11.5
实现细节	1.11.6
工具	1.12
easyOVS	1.12.1
参考	1.13
附：安装配置	1.14

深入理解 Neutron -- OpenStack 网络实现

Neutron 是 OpenStack 项目中负责提供网络服务的组件，它基于软件定义网络的思想，实现了网络虚拟化下的资源管理。

本书将剖析 Neutron 组件的原理和实现。

在线阅读：[GitBook](#) 或 [Github](#)。

- pdf 版本 [下载](#)
- epub 版本 [下载](#)

本书源码在 Github 上维护，欢迎参与：

https://github.com/yeasy/openstack_understand_Neutron。

感谢所有的 贡献者。

更新历史：

- V0.9: 2015-06-29
 - 添加对 DVR 更多细节分析。
- V0.8: 2015-03-24
 - 添加 LBaaS 服务分析；
- V0.7: 2015-03-23
 - 添加 VXLAN 模式分析；
 - 添加 DVR 服务分析。
- V0.6: 2014-04-04
 - 修正系统结构的图表；
 - 修正部分描述。
- V0.5: 2014-03-24
 - 添加对 vlan 模式下具体规则的分析；
 - 更新 GRE 模式下 answerfile 的 IP 信息。
- V0.4: 2014-03-20
 - 添加对安全组实现的完整分析，添加整体逻辑图表；
 - 添加 vlan 模式下的 RDO answer 文件（更新 IP 信息）。
- V0.3: 2014-03-10

- 添加 GRE 模式下对流表规则分析；
 - 添加 GRE 模式下的 answer 文件。 *V0.2: 2014-03-06
 - 修正图表引用错误；
 - 添加对 GRE 模式下流表细节分析。
- V0.1: 2014-02-20
 - 开始整体结构。

参加步骤

- 在 GitHub 上 fork 到自己的仓库，如 user/openstack_understand_Neutron，然后 clone 到本地，并设置用户信息。

```
$ git clone git@github.com:user/openstack_understand_Neutron  
.git  
$ cd openstack_understand_Neutron  
$ git config user.name "User"  
$ git config user.email user@email.com
```

- 修改代码后提交，并推送到自己的仓库。

```
$ #do some change on the content  
$ git commit -am "Fix issue #1: change helo to hello"  
$ git push
```

- 在 GitHub 网站上提交 pull request。
- 定期使用项目仓库内容更新自己仓库内容。

```
$ git remote add upstream https://github.com/yeasy/openstack  
_understand_Neutron  
$ git fetch upstream  
$ git checkout master  
$ git rebase upstream/master  
$ git push -f origin master
```


概述

Neutron 的设计目标是实现“网络即服务”，为了达到这一目标，在设计上遵循了基于“软件定义网络”实现网络虚拟化的原则，在实现上充分利用了 Linux 系统上的各种网络相关的技术。

理解了 Linux 系统上的这些概念将有利于快速理解 Neutron 的原理和实现。

涉及的 Linux 网络技术

- **bridge**：网桥，Linux 中用于表示一个能连接不同网络设备的虚拟设备，linux 中传统实现的网桥类似一个hub设备，而ovs管理的网桥一般类似交换机。
- **br-int**：bridge-integration，综合网桥，常用于表示实现主要内部网络功能的网桥。
- **br-ex**：bridge-external，外部网桥，通常表示负责跟外部网络通信的网桥。
- **GRE**：General Routing Encapsulation，一种通过封装来实现隧道的方式。在 openstack 中一般是基于L3的gre，即original pkt/GRE/IP/Ethernet
- **VETH**：虚拟ethernet接口，通常以pair的方式出现，一端发出的网包，会被另一端接收，可以形成两个网桥之间的通道。
- **qvb**：neutron veth, Linux Bridge-side
- **qvo**：neutron veth, OVS-side
- **TAP设备**：模拟一个二层的网络设备，可以接受和发送二层网包。
- **TUN设备**：模拟一个三层的网络设备，可以接受和发送三层网包。
- **iptables**：Linux 上常见的实现安全策略的防火墙软件。
- **Vlan**：虚拟 Lan，同一个物理 Lan 下用标签实现隔离，可用标号为 1-4094。
- **VXLAN**：一套利用 UDP 协议作为底层传输协议的 Overlay 实现。一般认为作为 VLan 技术的延伸或替代者。
- **namespace**：用来实现隔离的一套机制，不同 namespace 中的资源之间彼此不可见。

基本概念

Neutron管理下面的实体：

- 网络：隔离的 L2 域，可以是虚拟、逻辑或交换。
- 子网：隔离的 L3 域，IP 地址块。其中每个机器有一个 IP，同一个子网的主机彼此 L3 可见。
- 端口：网络上虚拟、逻辑或交换端口。所有这些实体都是虚拟的，拥有自动生成的唯一标识 id，支持CRUD功能，并在数据库中跟踪记录状态。

网络

隔离的 L2 广播域，一般是创建它的用户所有。用户可以拥有多个网络。网络是最基础的，子网和端口都需要关联到网络上。

网络上可以有多个子网。同一个网络上的主机一般可以通过交换机或路由器连通起来。

子网

隔离的 L3 域，子网代表了一组分配了 IP 的虚拟机。每个子网必须有一个 CIDR 和关联到一个网络。IP 可以从 CIDR 或者用户指定池中选取。

子网可能会有一个网关、一组 DNS 和主机路由。不同子网之间 L3 是互相不可见的，必须通过一个三层网关（即路由器）经过 L3 上进行通信。

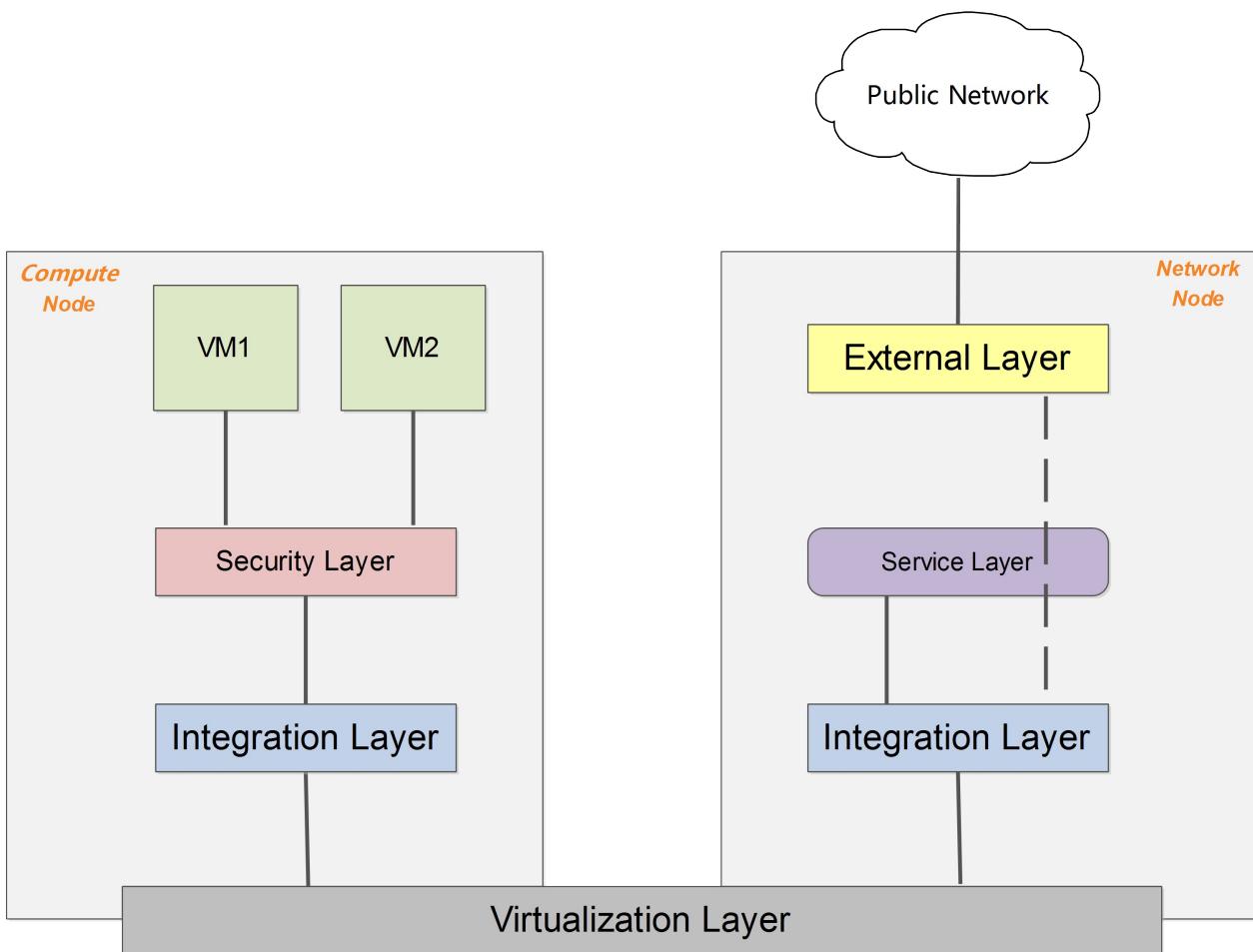
端口

可以进出流量的接口，往往绑定上若干 MAC 地址和 IP 地址，以进行寻址。一般为虚拟交换机上的虚拟接口。

虚拟机挂载网卡到端口上，通过端口访问网络。当端口有 IP 的时候，意味着它属于某个子网。

抽象系统架构

无论哪种具体的网络虚拟化实现，一个简化和抽象后的系统架构可以表述为下图所示。

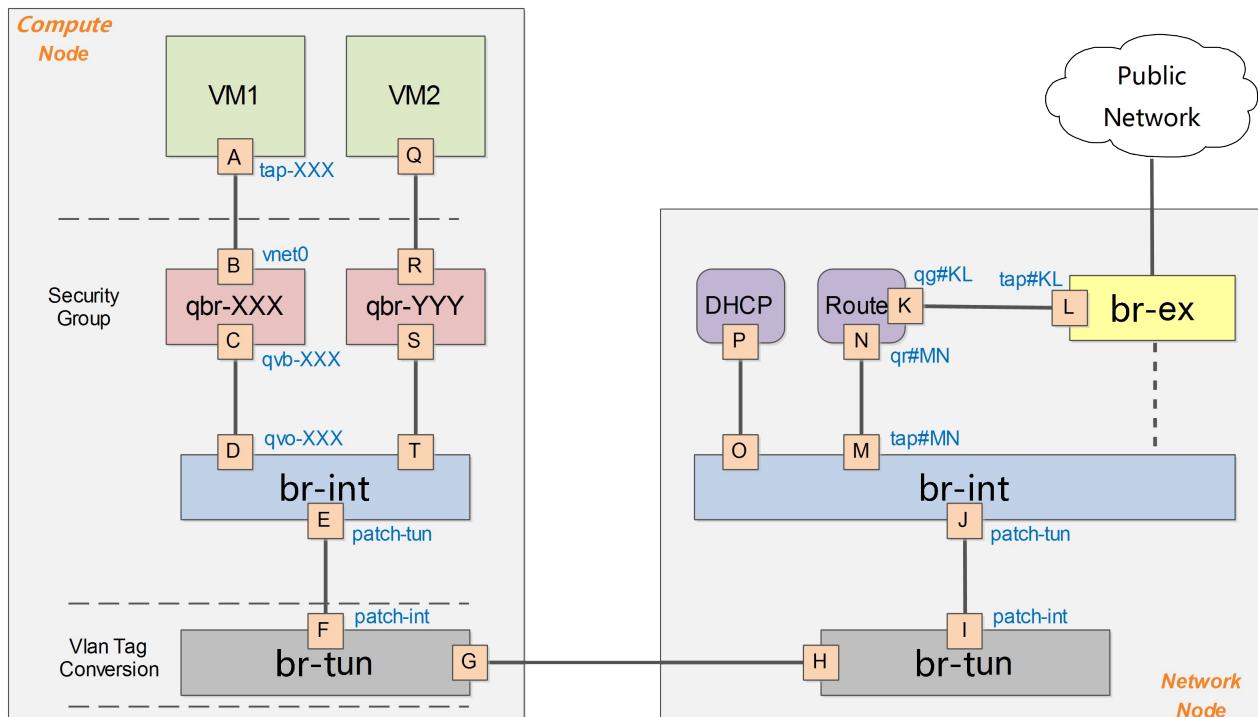


在启用 DVR 特性（J 版本以后支持）之前，所有流量（东西向、南北向）都需要经过网络节点的转发；DVR 特性则允许东西向流量和带有 Floating IP 的南北向流量不经过网络节点的转发，直接从计算节点的外部网络出去。

GRE 模式

下图给出了在OpenStack中网络实现的一个简化的架构示意。

一般的，OpenStack中网络实现包括vlan、gre、vxlan 等模式，此处以gre模式为例。



在OpenStack中，所有网络有关的逻辑管理均在Network节点中实现，例如DNS、DHCP以及路由等。Compute节点上只需要对所部属的虚拟机提供基本的网络功能支持，包括隔离不同租户的虚拟机和进行一些基本的安全策略管理（即security group）。

计算节点

以抽象系统架构的图表为例，Compute 节点上包括两台虚拟机 VM1 和 VM2，分别经过一个网桥（如 qbr-XXX）连接到 br-int 网桥上。br-int 网桥再经过 br-tun 网桥（物理网络是 GRE 实现）连接到物理主机外部网络。

对于物理网络通过 `vlan` 来隔离的情况，则一般会存在一个 br-eth 网桥，替代 br-tun 网桥。

qbr

在 VM1 中，虚拟机的网卡实际上连接到了物理机的一个 TAP 设备（即 A，常见名称如 `tap-XXX`）上，A 则进一步通过 VETH pair (A-B) 连接到网桥 `qbr-XXX` 的端口 `vnet0`（端口 B）上，之后再通过 VETH pair (C-D) 连到 `br-int` 网桥上。一般 C 的名字格式为 `qvb-XXX`，而 D 的名字格式为 `qvo-XXX`。注意它们的名称除了前缀外，后面的 id 都是一样的，表示位于同一个虚拟机网络到物理机网络的连接上。

之所以 TAP 设备 A 没有直接连接到网桥 `br-int` 上，是因为 OpenStack 需要通过 `iptables` 实现 `security group` 的安全策略功能。目前 `openvswitch` 并不支持应用 `iptables` 规则的 Tap 设备。

因为 `qbr` 的存在主要是为了辅助 `iptables` 来实现 `security group` 功能，有时候也被称为 安全网桥。详见 `security group` 部分的分析。

br-int

一个典型的 `br-int` 的端口如下所示：

```
# ovs-vsctl show
Bridge br-int
    Port "qvo-XXX"
        tag: 1
        Interface "qvo-XXX"
    Port patch-tun
        Interface patch-tun
            type: patch
            options: {peer=patch-int}
    Port br-int
        Interface br-int
            type: internal
```

其中，

- br-int 为内部端口。
- patch-tun（即端口E，端口号为1）连接到 br-tun 上，实现到外部网络的隧道。
- qvo-XXX（即端口D，端口号为2）带有 tag1，说明这个口是一个1号 vlan 的 access 端口。虚拟机发出的从该端口到达br-int的网包将被自动带上vlan tag 1，而其他带有 vlan tag 1 的网包则可以在去掉 vlan tag 后从该端口发出（即 vlan access 端口）。这个 vlan tag 是用来实现不同网络相互隔离的，比如租户创建一个网络（neutron net-create），则会被分配一个唯一的 vlan tag。

br-int 在 GRE 模式中作为一个 NORMAL 交换机使用，因此有效规则只有一条正常转发。如果两个在同一主机上的 vm 属于同一个 tenant 的（同一个 vlan tag），则它们之间的通信只需要经过 br-int 即可。

```
# ovs-ofctl dump-flows br-int
NXST_FLOW reply (xid=0x4):
cookie=0x0, duration=10727.864s, table=0, n_packets=198, n_bytes
s=17288, idle_age=13, priority=1 actions=NORMAL
```

br-tun

一个典型的 br-tun 上的端口类似：

```

Bridge br-tun
    Port patch-int
        Interface patch-int
            type: patch
            options: {peer=patch-tun}
    Port "gre-1"
        Interface "gre-1"
            type: gre
            options: {in_key=flow, local_ip="10.0.0.101", out_key=flow, remote_ip="10.0.0.100"}
    Port br-tun
        Interface br-tun
            type: internal

```

其中，

- patch-int（即端口 F，端口号为1）是连接到 br-int 上的 veth pair 的端口
- gre-1 端口（即端口 G，端口号为2）对应vm到外面的隧道。gre-1 端口是虚拟 gre 端口，当网包发送到这个端口的时候，会经过内核封包，然后从 10.0.0.101 发送到 10.0.0.100，即从本地的物理网卡（10.0.0.101）发出。

br-tun将带有 vlan tag 的 vm 跟外部通信的流量转换到对应的 gre 隧道，这上面要实现主要的转换逻辑，规则要复杂，一般通过多张表来实现。

典型的转发规则为：

```
# ovs-ofctl dump-flows br-tun
NXST_FLOW reply (xid=0x4):
  cookie=0x0, duration=10970.064s, table=0, n_packets=189, n_bytes=16232, idle_age=16, priority=1, in_port=1 actions=resubmit(,1)
  cookie=0x0, duration=10906.954s, table=0, n_packets=29, n_bytes=5736, idle_age=16, priority=1, in_port=2 actions=resubmit(,2)
  cookie=0x0, duration=10969.922s, table=0, n_packets=3, n_bytes=230, idle_age=10962, priority=0 actions=drop
  cookie=0x0, duration=10969.777s, table=1, n_packets=26, n_bytes=5266, idle_age=16, priority=0, dl_dst=00:00:00:00:00:00/01:00:00:00:00:00 actions=resubmit(,20)
  cookie=0x0, duration=10969.631s, table=1, n_packets=163, n_bytes=10966, idle_age=21, priority=0, dl_dst=01:00:00:00:00:00/01:00:00:00:00:00 actions=resubmit(,21)
  cookie=0x0, duration=688.456s, table=2, n_packets=29, n_bytes=5736, idle_age=16, priority=1, tun_id=0x1 actions=mod_vlan_vid:1, resubmit(,10)
  cookie=0x0, duration=10969.488s, table=2, n_packets=0, n_bytes=0, idle_age=10969, priority=0 actions=drop
  cookie=0x0, duration=10969.343s, table=3, n_packets=0, n_bytes=0, idle_age=10969, priority=0 actions=drop
  cookie=0x0, duration=10969.2s, table=10, n_packets=29, n_bytes=5736, idle_age=16, priority=1 actions=learn(table=20, hard_timeout=300, priority=1, NXM_OF_VLAN_TCI[0..11], NXM_OF_ETH_DST[] = NXM_OF_ETH_SRC[], load:0->NXM_OF_VLAN_TCI[], load:NXM_NX_TUN_ID[] -> NXM_NX_TUN_ID[], output:NXM_OF_IN_PORT[]), output:1
  cookie=0x0, duration=682.603s, table=20, n_packets=26, n_bytes=5266, hard_timeout=300, idle_age=16, hard_age=16, priority=1, vlan_tci=0x0001/0x0fff, dl_dst=fa:16:3e:32:0d:db actions=load:0->NXM_OF_VLAN_TCI[], load:0x1->NXM_NX_TUN_ID[], output:2
  cookie=0x0, duration=10969.057s, table=20, n_packets=0, n_bytes=0, idle_age=10969, priority=0 actions=resubmit(,21)
  cookie=0x0, duration=688.6s, table=21, n_packets=161, n_bytes=10818, idle_age=21, priority=1, dl_vlan=1 actions=strip_vlan, set_tunnel:0x1, output:2
  cookie=0x0, duration=10968.912s, table=21, n_packets=2, n_bytes=148, idle_age=689, priority=0 actions=drop
```

表 0

其中，表 0 中有 3 条规则：从内部端口 1（即 patch-int）来的，扔到表 1，从外部端口 2（即 gre-1）来的，扔到表2。

```
cookie=0x0, duration=10970.064s, table=0, n_packets=189, n_bytes=16232, idle_age=16, priority=1, in_port=1 actions=resubmit(,1)
cookie=0x0, duration=10906.954s, table=0, n_packets=29, n_bytes=5736, idle_age=16, priority=1, in_port=2 actions=resubmit(,2)
cookie=0x0, duration=10969.922s, table=0, n_packets=3, n_bytes=230, idle_age=10962, priority=0 actions=drop
```

表 1

表 1 处理内部过来的网包，有 2 条规则：如果是单播（00:00:00:00:00/01:00:00:00:00:00），则扔到表 20；如果是多播等（01:00:00:00:00/01:00:00:00:00:00），则扔到表 21。

```
cookie=0x0, duration=10969.777s, table=1, n_packets=26, n_bytes=5266, idle_age=16, priority=0, dl_dst=00:00:00:00:00:00/01:00:00:00:00:00 actions=resubmit(,20)
cookie=0x0, duration=10969.631s, table=1, n_packets=163, n_bytes=10966, idle_age=21, priority=0, dl_dst=01:00:00:00:00:00/01:00:00:00:00:00 actions=resubmit(,21)
```

表 2

表 2 处理外部过来的包。有 2 条规则：如果是 tunnel 1（合法的 tunnel id）的网包，则修改其 vlan id 为 1，并扔到表 10 学习记录来源；非 tunnel 1（非法的 tunnel id）的网包，则丢弃。

```
cookie=0x0, duration=688.456s, table=2, n_packets=29, n_bytes=5736, idle_age=16, priority=1, tun_id=0x1 actions=mod_vlan_vid:1,resubmit(,10)
cookie=0x0, duration=10969.488s, table=2, n_packets=0, n_bytes=0, idle_age=10969, priority=0 actions=drop
```

表 3

表 3 只有 1 条规则：丢弃。

表 10

表 10 负责学习。有一条规则，基于 `learn` 行动来创建反向（内部网包从 `gre` 端口发出去）的规则。`learn` 行动并非标准的 `openflow` 行动，是 `openvswitch` 自身的扩展行动，这个行动可以根据流内容动态来修改流表内容。

这条规则首先创建了一条新的流（该流对应 `vm` 从 `br-tun` 的 `gre` 端口发出的规则）：其中 `table=20` 表示规则添加在表 20；`NXM_OF_VLAN_TCI[0..11]` 表示匹配包自带的 `vlan id`；`NXM_OF_ETH_DST[]`=`NXM_OF_ETH_SRC[]` 表示 L2 目标地址需要匹配当前包的 L2 源地址；`load:0->NXM_OF_VLAN_TCI[]`，去掉 `vlan`，`load:NXM_NX_TUN_ID[]->NXM_NX_TUN_ID[]`，添加 tunnel 号为原始 tunnel 号；`output:NXM_OF_IN_PORT[]`，发出端口为原始包抵达的端口。

向表 20 添加完规则后，最后将匹配的当前网包从端口 1（即 `patch-int`）发出。

```
cookie=0x0, duration=10969.2s, table=10, n_packets=29, n_bytes=5736, idle_age=16, priority=1 actions=learn(table=20,hard_timeout=300,priority=1,NXM_OF_VLAN_TCI[0..11],NXM_OF_ETH_DST[]=NXM_OF_ETH_SRC[],load:0->NXM_OF_VLAN_TCI[],load:NXM_NX_TUN_ID[]->NXM_NX_TUN_ID[],output:NXM_OF_IN_PORT[]),output:1
```

表 20

表 20 中有两条规则，其中第一条即表 10 中规则利用 `learn` 行动创建的内部向外部发包的流表项，第 2 条提交其他流到表 21。

```
cookie=0x0, duration=682.603s, table=20, n_packets=26, n_bytes=5266, hard_timeout=300, idle_age=16, hard_age=16, priority=1,vlan_tci=0x0001/0x0fff,d1_dst=fa:16:3e:32:0d:db actions=load:0->NXM_OF_VLAN_TCI[],load:0x1->NXM_NX_TUN_ID[],output:2
cookie=0x0, duration=10969.057s, table=20, n_packets=0, n_bytes=0, idle_age=10969, priority=0 actions=resubmit(,21)
```

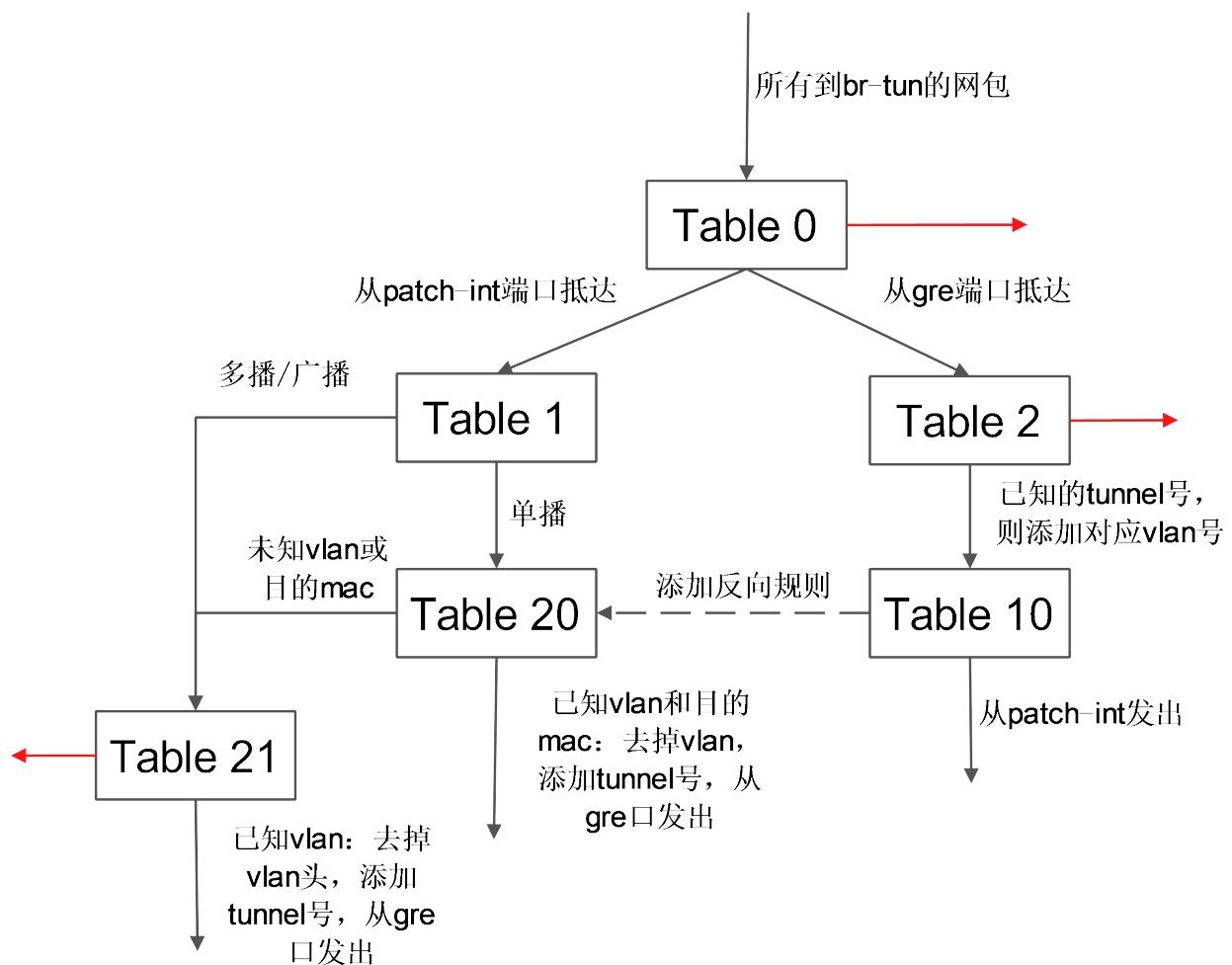
表 21 有 2 条规则，第一条是匹配所有目标 `vlan` 为 1 的网包，去掉 `vlan`，然后从端口 2（`gre` 端口）发出。第二条是丢弃。

```

cookie=0x0, duration=688.6s, table=21, n_packets=161, n_bytes=10
818, idle_age=21, priority=1, dl_vlan=1 actions=strip_vlan, set_tu
nnel:0x1, output:2
cookie=0x0, duration=10968.912s, table=21, n_packets=2, n_bytes
=148, idle_age=689, priority=0 actions=drop

```

这些规则所组成的整体转发逻辑如下图所示。



网络节点

br-tun

```
Bridge br-tun
    Port br-tun
        Interface br-tun
            type: internal
    Port patch-int
        Interface patch-int
            type: patch
            options: {peer=patch-tun}
    Port "gre-2"
        Interface "gre-2"
            type: gre
            options: {in_key=flow, local_ip="10.0.0.100", ou
t_key=flow, remote_ip="10.0.0.101"}
```

Compute 节点上发往 GRE 隧道的网包最终抵达 Network 节点上的 br-tun，该网桥的规则包括：

```
# ovs-ofctl dump-flows br-tun
NXST_FLOW reply (xid=0x4):
cookie=0x0, duration=19596.862s, table=0, n_packets=344, n_bytes
=66762, idle_age=4, priority=1, in_port=1 actions=resubmit(,1)
  cookie=0x0, duration=19537.588s, table=0, n_packets=625, n_byt
s=125972, idle_age=4, priority=1, in_port=2 actions=resubmit(,2)
  cookie=0x0, duration=19596.602s, table=0, n_packets=2, n_bytes=
140, idle_age=19590, priority=0 actions=drop
  cookie=0x0, duration=19596.343s, table=1, n_packets=323, n_byte
s=65252, idle_age=4, priority=0, dl_dst=00:00:00:00:00:00/01:00:0
0:00:00:00 actions=resubmit(,20)
  cookie=0x0, duration=19596.082s, table=1, n_packets=21, n_bytes
=1510, idle_age=5027, priority=0, dl_dst=01:00:00:00:00:00/01:00:
00:00:00:00 actions=resubmit(,21)
  cookie=0x0, duration=9356.289s, table=2, n_packets=625, n_bytes
=125972, idle_age=4, priority=1, tun_id=0x1 actions=mod_vlan_vid:
1,resubmit(,10)
  cookie=0x0, duration=19595.821s, table=2, n_packets=0, n_bytes=
0, idle_age=19595, priority=0 actions=drop
  cookie=0x0, duration=19595.554s, table=3, n_packets=0, n_bytes=
0, idle_age=19595, priority=0 actions=drop
  cookie=0x0, duration=19595.292s, table=10, n_packets=625, n_byt
es=125972, idle_age=4, priority=1 actions=learn(table=20,hard_ti
meout=300,priority=1,NXM_OF_VLAN_TCI[0..11],NXM_OF_ETH_DST[])=NXM
_OF_ETH_SRC[],load:0->NXM_OF_VLAN_TCI[],load:NXM_NX_TUN_ID[]->NX
M_NX_TUN_ID[],output:NXM_OF_IN_PORT[],output:1
  cookie=0x0, duration=9314.338s, table=20, n_packets=323, n_byt
s=65252, hard_timeout=300, idle_age=4, hard_age=3, priority=1,vl
an_tci=0x0001/0x0fff,dl_dst=fa:16:3e:cb:11:f6 actions=load:0->NX
M_OF_VLAN_TCI[],load:0x1->NXM_NX_TUN_ID[],output:2
  cookie=0x0, duration=19595.026s, table=20, n_packets=0, n_bytes
=0, idle_age=19595, priority=0 actions=resubmit(,21)
  cookie=0x0, duration=9356.592s, table=21, n_packets=9, n_bytes=
586, idle_age=5027, priority=1,dl_vlan=1 actions=strip_vlan, set_
tunnel:0x1,output:2
  cookie=0x0, duration=19594.759s, table=21, n_packets=12, n_byt
s=924, idle_age=5057, priority=0 actions=drop
```

这些规则跟 Compute 节点上 br-tun 的规则相似，完成 tunnel 跟 vlan 之间的转换。

br-int

```
Bridge br-int
    Port "qr-ff19a58b-3d"
        tag: 1
        Interface "qr-ff19a58b-3d"
            type: internal
    Port br-int
        Interface br-int
            type: internal
    Port patch-tun
        Interface patch-tun
            type: patch
            options: {peer=patch-int}
    Port "tap4385f950-8b"
        tag: 1
        Interface "tap4385f950-8b"
            type: internal
```

该集成网桥上挂载了很多进程来提供网络服务，包括路由器、DHCP服务器等。这些进程不同的租户可能都需要，彼此的地址空间可能冲突，也可能跟物理网络的地址空间冲突，因此都运行在独立的网络名字空间中。规则跟computer节点的br-int规则一致，表现为一个正常交换机。

```
# ovs-ofctl dump-flows br-int
NXST_FLOW reply (xid=0x4):
  cookie=0x0, duration=18198.244s, table=0, n_packets=849, n_bytes=
  164654, idle_age=43, priority=1 actions=NORMAL
```

网络名字空间

在 Linux 中，网络名字空间可以被认为是隔离的拥有单独网络栈（网卡、路由转发表、iptables）的环境。网络名字空间经常用来隔离网络设备和服务，只有拥有同样网络名字空间的设备，才能看到彼此。可以用 ip netns list 命令来查看已经存在的名字空间。

```
# ip netns
qdhcp-88b1609c-68e0-49ca-a658-f1edff54a264
qrouter-2d214fde-293c-4d64-8062-797f80ae2d8f
```

qdhcp 开头的名字空间是 dhcp 服务器使用的，qrouter 开头的则是 router 服务使用的。可以通过 ip netns exec namespaceid command 来在指定的网络名字空间中执行网络命令，例如

```
# ip netns exec qdhcp-88b1609c-68e0-49ca-a658-f1edff54a264 ip ad
dr
71: ns-f14c598d-98: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 q
disc pfifo_fast state UP qlen 1000
    link/ether fa:16:3e:10:2f:03 brd ff:ff:ff:ff:ff:ff
    inet 10.1.0.3/24 brd 10.1.0.255 scope global ns-f14c598d-98
        inet6 fe80::f816:3eff:fe10:2f03/64 scope link
            valid_lft forever preferred_lft forever
```

可以看到，dhcp 服务的网络名字空间中只有一个网络接口“ns-f14c598d-98”，它连接到 br-int 的 tapf14c598d-98 接口上。

dhcp 服务

dhcp 服务是通过 dnsmasq 进程（轻量级服务器，可以提供 dns、dhcp、tftp 等服务）来实现的，该进程绑定到 dhcp 名字空间中的 br-int 的接口上。可以查看相关的进程。

```
# ps -fe | grep 88b1609c-68e0-49ca-a658-f1edff54a264
nobody    23195      1  0 Oct26 ?          00:00:00 dnsmasq --no-hos
ts --no-resolv --strict-order --bind-interfaces --interface=ns-f
14c598d-98 --except-interface=lo --pid-file=/var/lib/neutron/dhc
p/88b1609c-68e0-49ca-a658-f1edff54a264/pid --dhcp-hostsfile=/var
/lib/neutron/dhcp/88b1609c-68e0-49ca-a658-f1edff54a264/host --dh
cp-optfile=/var/lib/neutron/dhcp/88b1609c-68e0-49ca-a658-f1edff
54a264/opts --dhcp-script=/usr/bin/neutron-dhcp-agent-dnsmasq-le
ase-update --leasefile-ro --dhcp-range=tag0,10.1.0.0,static,120s
--conf-file= --domain=openstacklocal
root      23196 23195  0 Oct26 ?          00:00:00 dnsmasq --no-hos
ts --no-resolv --strict-order --bind-interfaces --interface=ns-f
14c598d-98 --except-interface=lo --pid-file=/var/lib/neutron/dhc
p/88b1609c-68e0-49ca-a658-f1edff54a264/pid --dhcp-hostsfile=/var
/lib/neutron/dhcp/88b1609c-68e0-49ca-a658-f1edff54a264/host --dh
cp-optfile=/var/lib/neutron/dhcp/88b1609c-68e0-49ca-a658-f1edff
54a264/opts --dhcp-script=/usr/bin/neutron-dhcp-agent-dnsmasq-le
ase-update --leasefile-ro --dhcp-range=tag0,10.1.0.0,static,120s
--conf-file= --domain=openstacklocal
```

router服务

首先，要理解什么是 router，router 是提供跨 subnet 的互联功能的。比如用户的内部网络中主机想要访问外部互联网的地址，就需要 router 来转发（因此，所有跟外部网络的流量都必须经过 router）。目前 router 的实现是通过 iptables 进行的。同样的，router 服务也运行在自己的名字空间中，可以通过如下命令查看：

```
# ip netns exec qrouter-2d214fde-293c-4d64-8062-797f80ae2d8f ip
addr
66: qg-d48b49e0-aa: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 q
disc pfifo_fast state UP qlen 1000
    link/ether fa:16:3e:5c:a2:ac brd ff:ff:ff:ff:ff:ff
        inet 172.24.4.227/28 brd 172.24.4.239 scope global qg-d48b49
e0-aa
            inet 172.24.4.228/32 brd 172.24.4.228 scope global qg-d48b49
e0-aa
                inet6 fe80::f816:3eff:fe5c:a2ac/64 scope link
                    valid_lft forever preferred_lft forever
68: qr-c2d7dd02-56: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 q
disc pfifo_fast state UP qlen 1000
    link/ether fa:16:3e:ea:64:6e brd ff:ff:ff:ff:ff:ff
        inet 10.1.0.1/24 brd 10.1.0.255 scope global qr-c2d7dd02-56
            inet6 fe80::f816:3eff:feeaa:646e/64 scope link
                valid_lft forever preferred_lft forever
```

可以看出，该名字空间中包括两个网络接口。

第一个接口 `qg-d48b49e0-aa`（即 K）是外部接口（`qg=q gateway`），将路由器的网关指向默认网关（通过 `router-gateway-set` 命令指定），这个接口连接到 `br-ex` 上的 `tapd48b49e0-aa`（即 L）。

第二个接口 `qr-c2d7dd02-56`（即 N，`qr=q bridge`）跟 `br-int` 上的 `tapc2d7dd02-56` 口（即 M）相连，将 `router` 进程连接到集成网桥上。

查看该名字空间中的路由表：

```
# ip netns exec qrouter-2d214fde-293c-4d64-8062-797f80ae2d8f ip
route
172.24.4.224/28 dev qg-d48b49e0-aa  proto kernel  scope link  sr
c 172.24.4.227
10.1.0.0/24 dev qr-c2d7dd02-56  proto kernel  scope link  src 10
.1.0.1
default via 172.24.4.225 dev qg-d48b49e0-aa
```

其中，第一条规则是将到 `172.24.4.224/28` 段的访问都从网卡 `qg-d48b49e0-aa`（即 K）发出。

第二条规则是将到 10.1.0.0/24 段的访问都从网卡 qr-c2d7dd02-56（即 N）发出。最后一条是默认路由，所有的通过 qg-d48b49e0-aa 网卡（即 K）发出。floating ip 服务同样在路由器名字空间中实现，例如如果绑定了外部的 floating ip 172.24.4.228 到某个虚拟机 10.1.0.2，则 nat 表中规则为：

```
# ip netns exec qrouter-2d214fde-293c-4d64-8062-797f80ae2d8f iptables -t nat -S
-P PREROUTING ACCEPT
-P POSTROUTING ACCEPT
-P OUTPUT ACCEPT
-N neutron-l3-agent-OUTPUT
-N neutron-l3-agent-POSTROUTING
-N neutron-l3-agent-PREROUTING
-N neutron-l3-agent-float-snat
-N neutron-l3-agent-snat
-N neutron-postrouting-bottom
-A PREROUTING -j neutron-l3-agent-PREROUTING
-A POSTROUTING -j neutron-l3-agent-POSTROUTING
-A POSTROUTING -j neutron-postrouting-bottom
-A OUTPUT -j neutron-l3-agent-OUTPUT
-A neutron-l3-agent-OUTPUT -d 172.24.4.228/32 -j DNAT --to-destination 10.1.0.2
-A neutron-l3-agent-POSTROUTING ! -i qg-d48b49e0-aa ! -o qg-d48b49e0-aa -m conntrack ! --ctstate DNAT -j ACCEPT
-A neutron-l3-agent-PREROUTING -d 169.254.169.254/32 -p tcp -m tcp --dport 80 -j REDIRECT --to-ports 9697
-A neutron-l3-agent-PREROUTING -d 172.24.4.228/32 -j DNAT --to-destination 10.1.0.2
-A neutron-l3-agent-float-snat -s 10.1.0.2/32 -j SNAT --to-source 172.24.4.228
-A neutron-l3-agent-snat -j neutron-l3-agent-float-snat
-A neutron-l3-agent-snat -s 10.1.0.0/24 -j SNAT --to-source 172.24.4.227
-A neutron-postrouting-bottom -j neutron-l3-agent-snat
```

其中 SNAT 和 DNAT 规则完成外部 floating ip 到内部 ip 的映射：

```
-A neutron-l3-agent-OUTPUT -d 172.24.4.228/32 -j DNAT --to-destination 10.1.0.2
-A neutron-l3-agent-PREROUTING -d 172.24.4.228/32 -j DNAT --to-destination 10.1.0.2
-A neutron-l3-agent-float-snat -s 10.1.0.2/32 -j SNAT --to-source 172.24.4.228
```

另外有一条 SNAT 规则把所有其他的内部IP出来的流量都映射到外部IP 172.24.4.227。这样即使在内部虚拟机没有外部IP的情况下，也可以发起对外网的访问。

```
-A neutron-l3-agent-snat -s 10.1.0.0/24 -j SNAT --to-source 172.24.4.227
```

br-ex

```
Bridge br-ex
  Port "eth1"
    Interface "eth1"
  Port br-ex
    Interface br-ex
      type: internal
  Port "qg-1c3627de-1b"
    Interface "qg-1c3627de-1b"
      type: internal
```

br-ex 上直接连接到外部物理网络，一般情况下网关在物理网络中已经存在，则直接转发即可。

```
# ovs-ofctl dump-flows br-ex
NXST_FLOW reply (xid=0x4):
  cookie=0x0, duration=23431.091s, table=0, n_packets=893539, n_bytes=504805376, idle_age=0, priority=0 actions=NORMAL
```

如果对外部网络的网关地址配置到了 br-ex（即br-ex作为一个网关）：

```
# ip addr add 172.24.4.225/28 dev br-ex
```

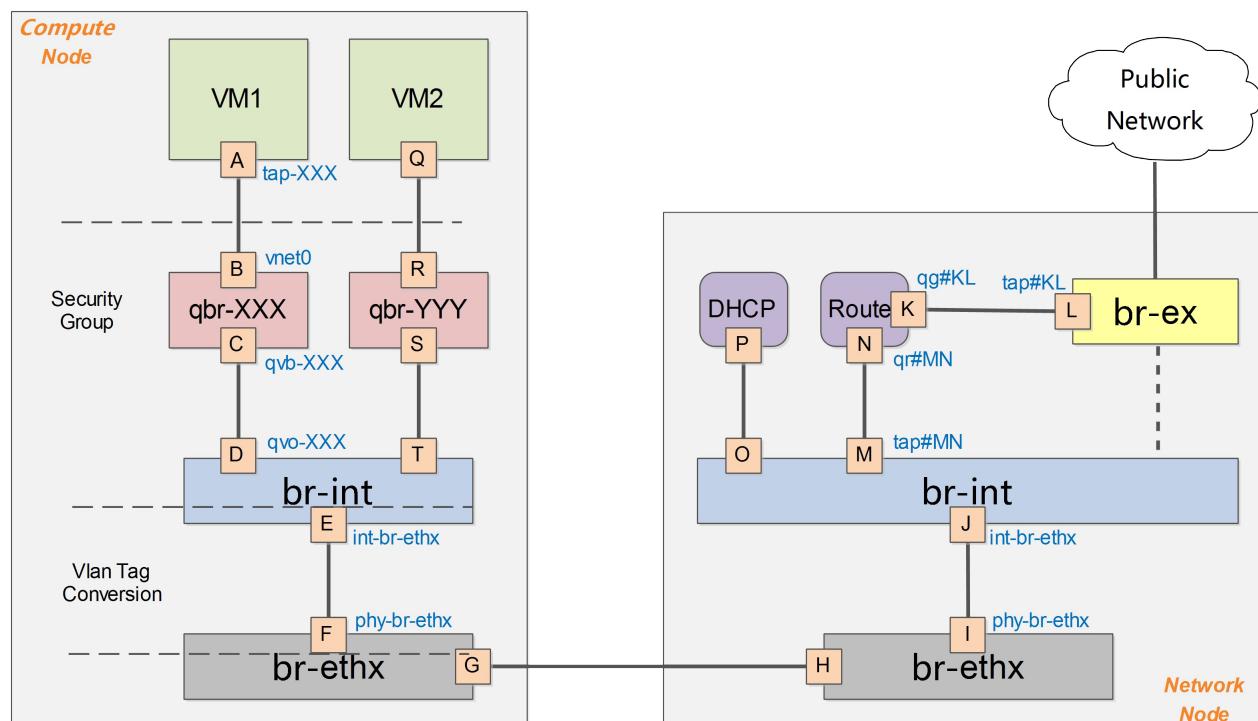
需要将内部虚拟机发出的流量进行 **SNAT**，之后发出。

```
# iptables -A FORWARD -d 172.24.4.224/28 -j ACCEPT  
# iptables -A FORWARD -s 172.24.4.224/28 -j ACCEPT  
# iptables -t nat -I POSTROUTING 1 -s 172.24.4.224/28 -j MASQUERADE
```

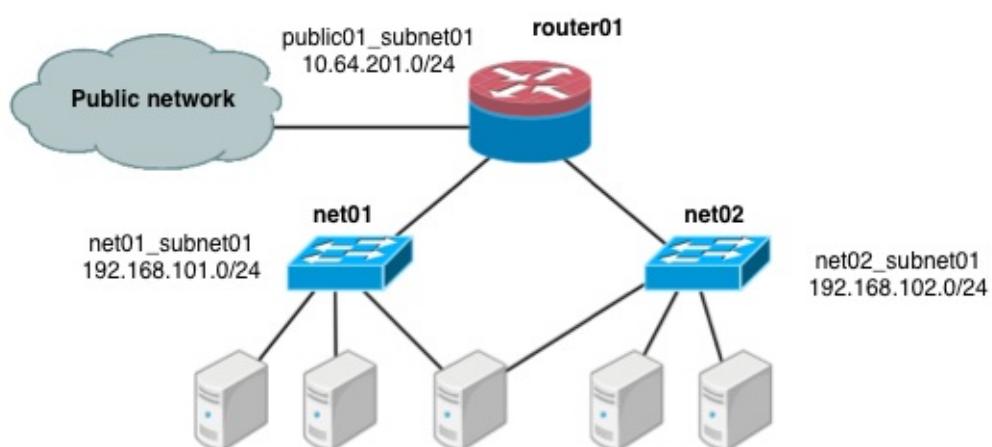
VLAN 模式

Vlan模式下的系统架构跟GRE模式下类似，如下图所示。

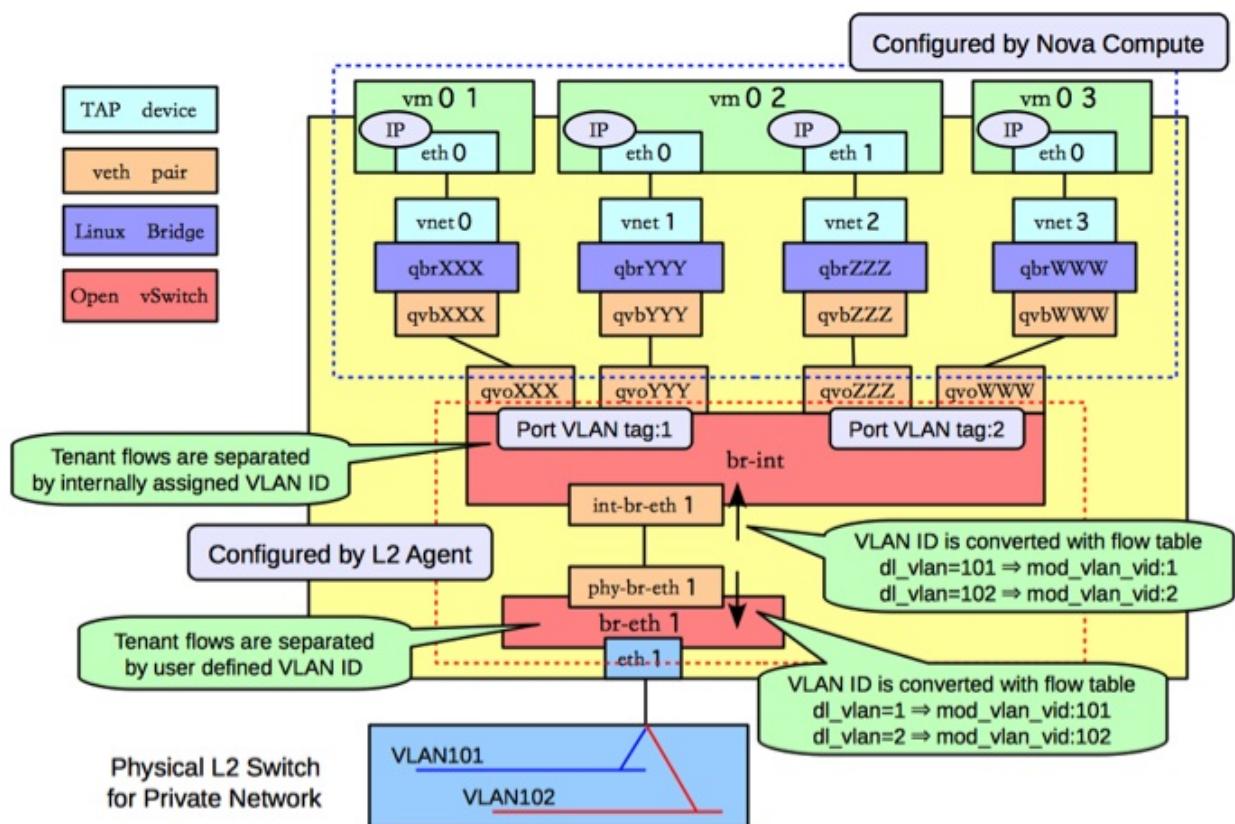
需要注意的是，在vlan模式下，vlan tag的转换需要在br-int和br-ethx两个网桥上进行相互配合。即br-int负责从int-br-ethX过来的包（带外部vlan）转换为内部vlan，而br-ethx负责从phy-br-ethx过来的包（带内部vlan）转化为外部的vlan。



下面进行一些细节的补充讨论，以Vlan作为物理网络隔离的实现。假如要实现同一个租户下两个子网，如下图所示：



计算节点



查看网桥信息，主要包括两个网桥：**br-int**和**br-eth1**：

```
[root@Compute ~]# ovs-vsctl show
f758a8b8-2fd0-4a47-ab2d-c49d48304f82
    Bridge "br-eth1"
        Port "phy-br-eth1"
            Interface "phy-br-eth1"
        Port "br-eth1"
            Interface "br-eth1"
                type: internal
        Port "eth1"
            Interface "eth1"
    Bridge br-int
        Port "qvoXXX"
            tag: 1
            Interface "qvoXXX"
        Port "qvoYYY"
            tag: 1
            Interface "qvoYYY"
        Port "qvoZZZ"
            tag: 2
            Interface "qvoZZZ"
        Port "qvoWWW"
            tag: 2
            Interface "qvoWWW"
        Port "int-br-eth1"
            Interface "int-br-eth1"
        Port br-int
            Interface br-int
                type: internal
```

类似GRE模式下，br-int负责租户隔离，br-eth1负责跟计算节点外的网络通信。在Vlan模式下，租户的流量隔离是通过vlan来进行的，因此此时包括两种vlan，虚拟机在Compute Node内流量带有的local vlan和在Compute Node之外物理网络上隔离不同租户的vlan。

br-int和br-eth1分别对从端口int-br-eth1和phy-br-eth1上到达的网包进行vlan tag的处理。此处有两个网，分别带有两个vlan tag（内部tag1对应外部tag101，内部tag2对应外部tag102）。其中，安全组策略仍然在qbr相关的iptables上实现。

br-int

与GRE模式不同的是，br-int完成从br-eth1上过来流量（从口int-br-eth1到达）的vlan tag转换，可能的规则为

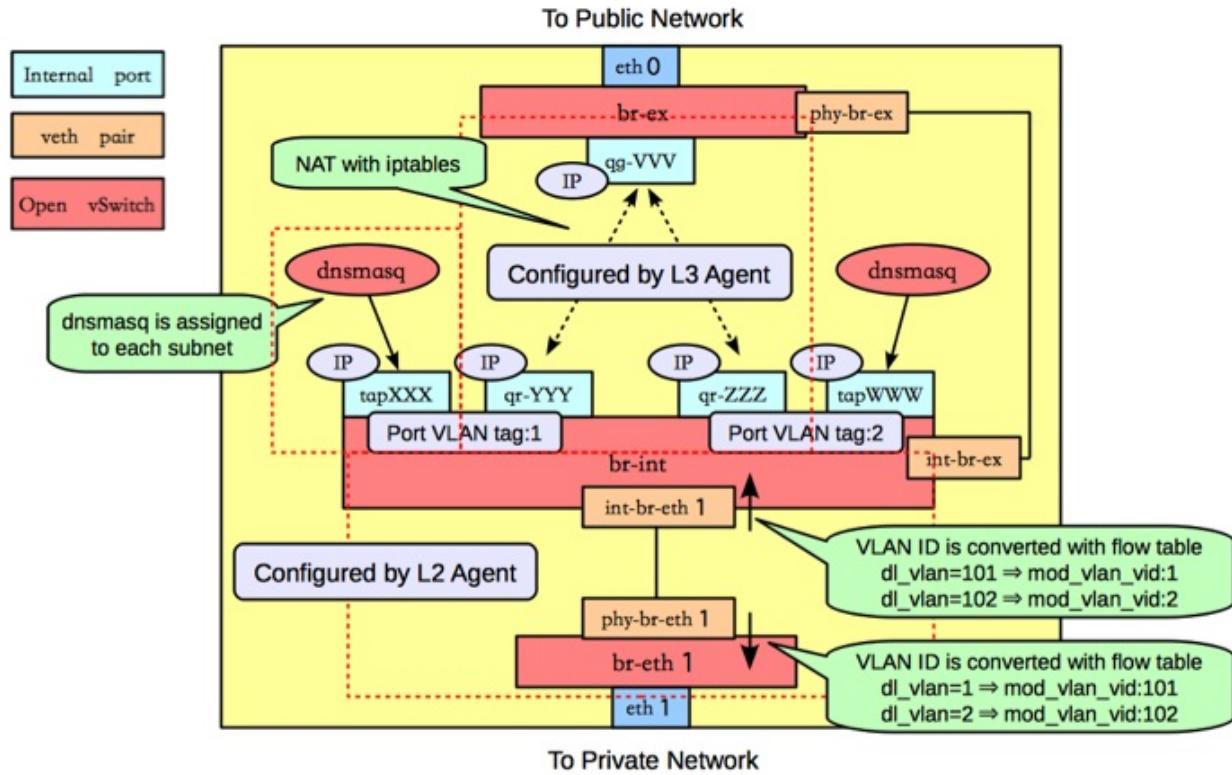
```
#ovs-ofctl dump-flows br-int
  cookie=0x0, duration=100.795s, table=0, n_packets=6, n_bytes=46
  8, idle_age=90, priority=2, in_port=3 actions=drop
  cookie=0x0, duration=97.069s, table=0, n_packets=22, n_bytes=66
  22, idle_age=31, priority=3, in_port=3, dl_vlan=101 actions=mod_vl
  an_vid:1, NORMAL
  cookie=0x0, duration=95.781s, table=0, n_packets=8, n_bytes=116
  5, idle_age=11, priority=3, in_port=3, dl_vlan=102 actions=mod_vla
  n_vid:2, NORMAL
  cookie=0x0, duration=103.626s, table=0, n_packets=47, n_bytes=1
  3400, idle_age=11, priority=1 actions=NORMAL
```

br-eth1

br-eth1上负责从br-int上过来的流量（从口phy-br-eth1到达），实现local vlan到外部vlan的转换。

```
#ovs-ofctl dump-flows br-eth0
NXST_FLOW reply (xid=0x4):
  cookie=0x0, duration=73.461s, table=0, n_packets=51, n_bytes=32
  403, idle_age=2, hard_age=65534, priority=4, in_port=4, dl_vlan=1
  actions=mod_vlan_vid:101, NORMAL
  cookie=0x0, duration=83.461s, table=0, n_packets=51, n_bytes=32
  403, idle_age=2, hard_age=65534, priority=4, in_port=4, dl_vlan=2
  actions=mod_vlan_vid:102, NORMAL
  cookie=0x0, duration=651.538s, table=0, n_packets=72, n_bytes=3
  908, idle_age=2574, hard_age=65534, priority=2, in_port=4 actions
  =drop
  cookie=0x0, duration=654.002s, table=0, n_packets=31733, n_byte
  s=6505880, idle_age=2, hard_age=65534, priority=1 actions=NORMAL
```


网络节点



类似GRE模式下，br-eth1收到到达的网包，int-br-eth1和phy-br-eth1上分别进行vlan转换，保证到达br-int上的网包都是带有内部vlan tag，到达br-eth1上的都是带有外部vlan tag。br-ex则完成到OpenStack以外网络的连接。查看网桥信息，包括三个网桥，br-eth1、br-int和br-ex。

```
#0VS
3bd78da8-d3b5-4112-a766-79506a7e2801
Bridge br-ex
  Port "qg-VVV"
    Interface "qg-VVV"
      type: internal
  Port br-ex
    Interface br-ex
      type: internal
  Port "eth0"
    Interface "eth0"
Bridge br-int
  Port br-int
```

```
        Interface br-int
            type: internal
        Port "int-br-eth1"
            Interface "int-br-eth0"
        Port "tapXXX"
            tag: 1
            Interface "tapXXX"
                type: internal
        Port "tapWWW"
            tag: 2
            Interface "tapWWW"
                type: internal
        Port "qr-YYY"
            tag: 1
            Interface "qr-YYY"
                type: internal
        Port "qr-ZZZ"
            tag: 2
            Interface "qr-ZZZ"
                type: internal
    Bridge "br-eth1"
        Port "phy-br-eth1"
            Interface "phy-br-eth1"
        Port "br-eth1"
            Interface "br-eth1"
                type: internal
        Port "eth1"
            Interface "eth1"
```

br-eth1

br-eth1主要负责把物理网络上外部vlan转化为local vlan。

```
#ovs-ofctl dump-flows br-eth1
NXST_FLOW reply (xid=0x4):
  cookie=0x0, duration=144.33s, table=0, n_packets=13, n_bytes=28
  404, idle_age=24, hard_age=65534, priority=4, in_port=5, dl_vlan=1
  01 actions=mod_vlan_vid:1, NORMAL
  cookie=0x0, duration=144.33s, table=0, n_packets=13, n_bytes=28
  404, idle_age=24, hard_age=65534, priority=4, in_port=5, dl_vlan=1
  02 actions=mod_vlan_vid:2, NORMAL
  cookie=0x0, duration=608.373s, table=0, n_packets=23, n_bytes=1
  706, idle_age=65534, hard_age=65534, priority=2, in_port=5 actions=drop
  cookie=0x0, duration=675.373s, table=0, n_packets=58, n_bytes=1
  0625, idle_age=24, hard_age=65534, priority=1 actions=NORMAL
```

br-int

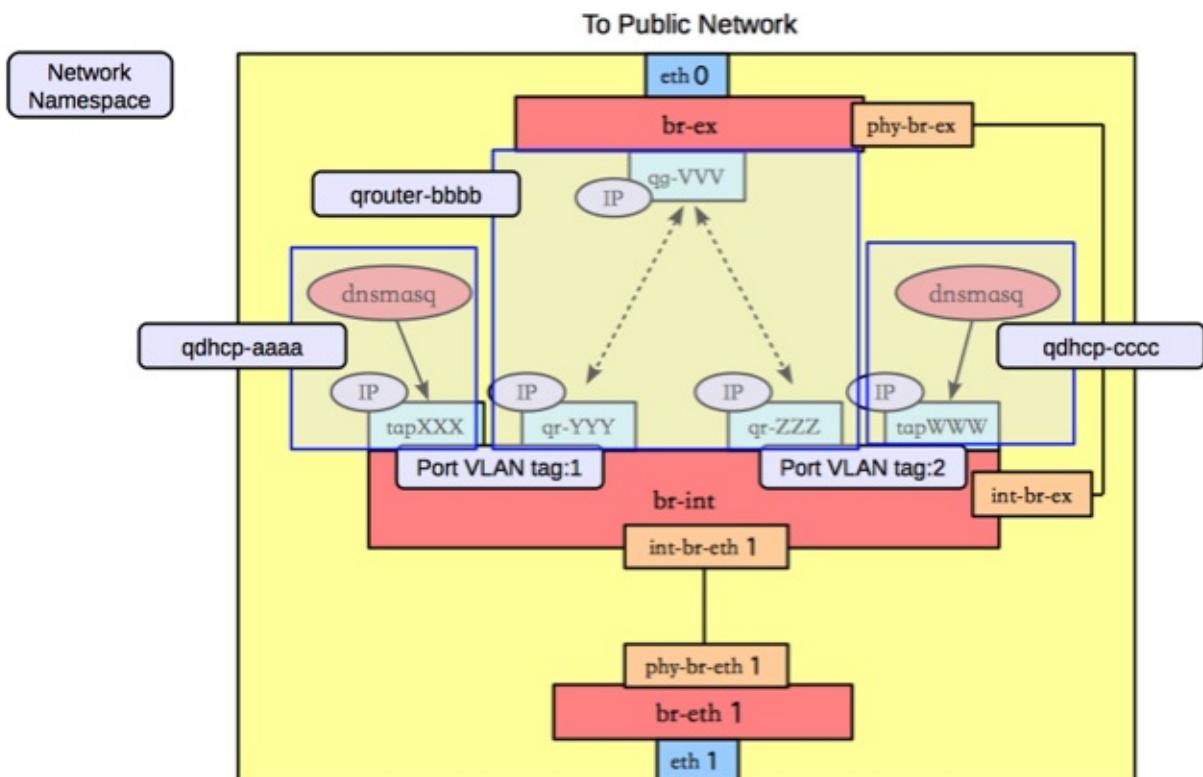
br-int上挂载了大量的agent来提供各种网络服务，另外负责对发往br-eth1的流量，实现local vlan转化为外部vlan。

```
#ovs-ofctl dump-flows br-int
NXST_FLOW reply (xid=0x4):
  cookie=0x0, duration=147294.121s, table=0, n_packets=224, n_bytes=33961, idle_age=13, hard_age=65534, priority=3, in_port=4, dl_vlan=1 actions=mod_vlan_vid:101, NORMAL
  cookie=0x0, duration=603538.84s, table=0, n_packets=19, n_bytes=2234, idle_age=18963, hard_age=65534, priority=2, in_port=4 actions=drop
  cookie=0x0, duration=603547.134s, table=0, n_packets=31901, n_bytes=6419756, idle_age=13, hard_age=65534, priority=1 actions=NORMAL
```

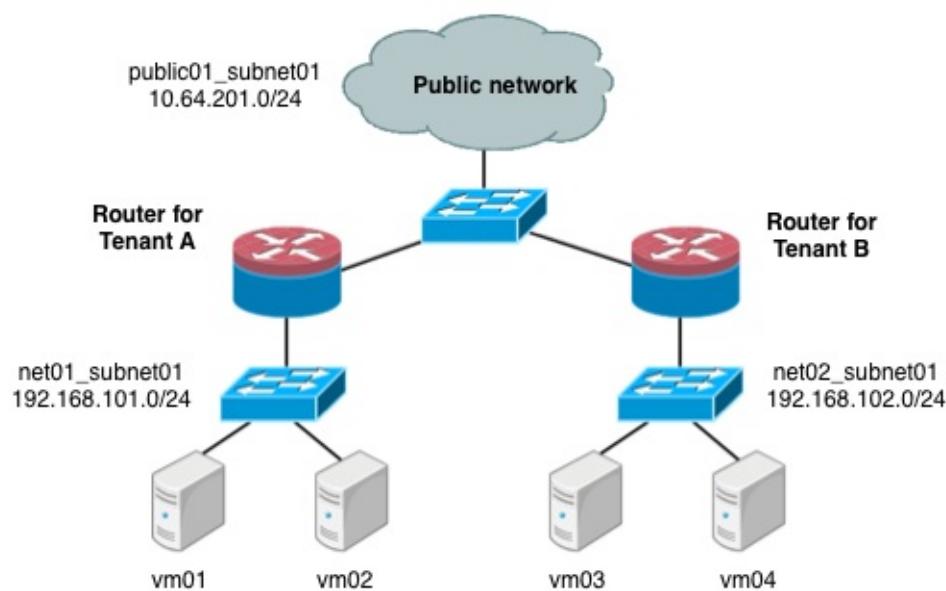
dnsmasq负责提供DHCP服务，绑定到某个特定的名字空间上，每个需要DHCP服务的租户网络有自己专属隔离的DHCP服务（图中的tapXXX和tapWWW上各自监听了一个dnsmasq）。

路由是L3 agent来实现，每个子网在br-int上有一个端口（qr-YYY和qr-ZZZ，已配置IP，分别是各自内部子网的网关），L3 agent绑定到上面。要访问外部的公共网络，需要通过L3 agent发出，而不是经过int-br-ex到phy-br-ex（实际上并没有网包从这个veth pair传输）。如果要使用外部可见的floating IP，L3 agent仍然需要通过iptables来进行NAT。

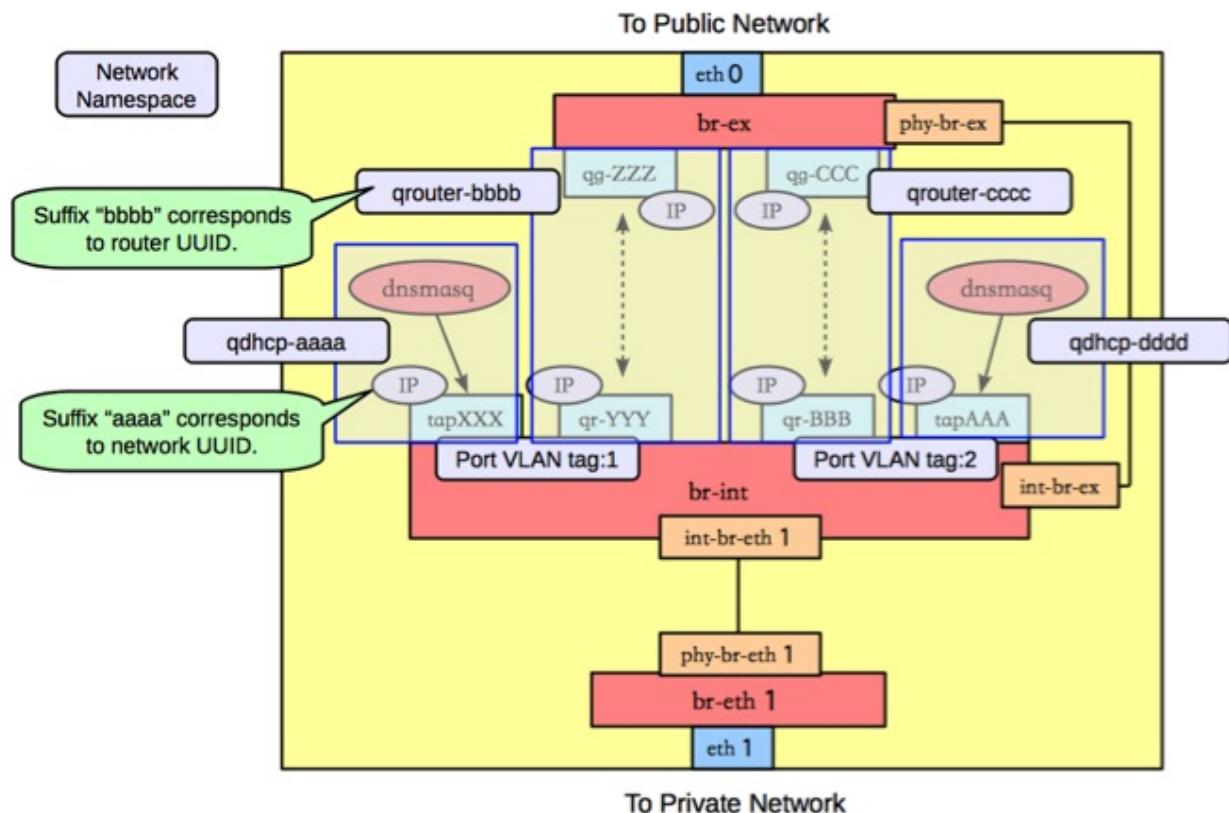
每个L3 agent或dnsmasq都在各自独立的名字空间中，如下图所示，其中同一租户的两个子网都使用了同一个路由器。



对于子网使用不同路由器的情况，多个路由器会在自己独立的名字空间中。例如要实现两个租户的两个子网的情况，如下图所示。



这种情况下，网络节点上的名字空间如下图所示。



br-ex

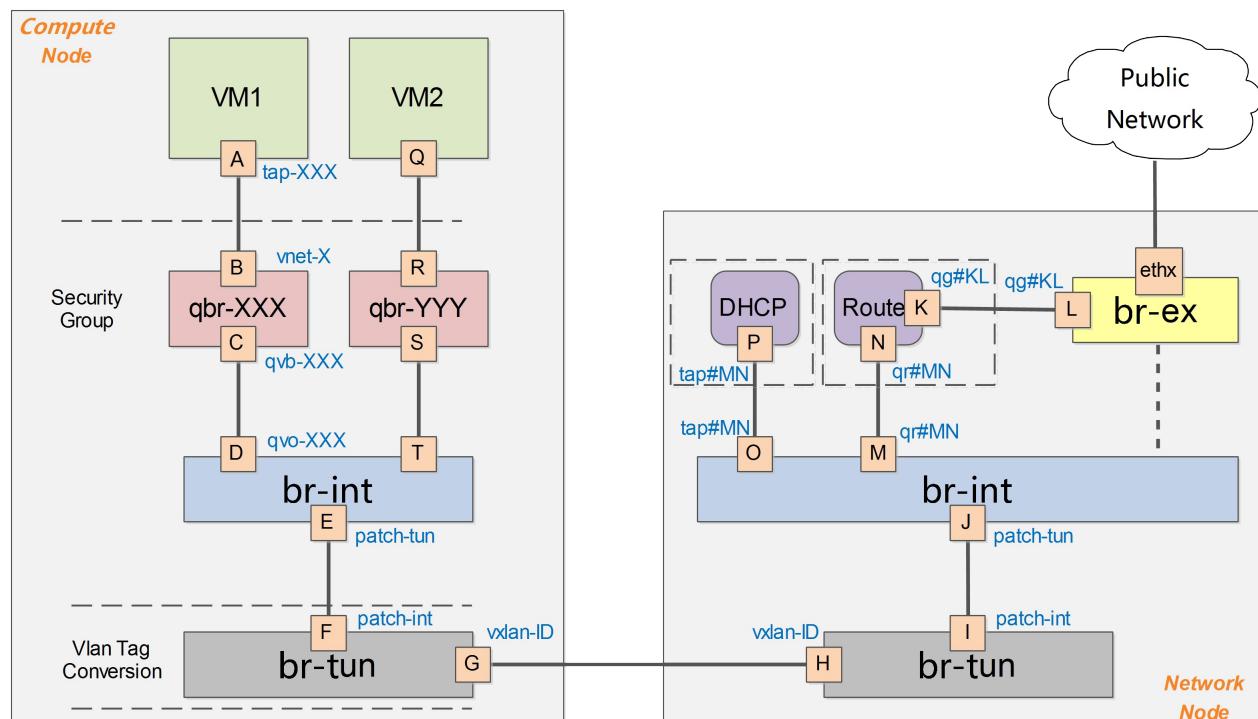
br-ex要做的事情很简单，只需要正常转发即可。

```
#ovs-ofctl dump-flows br-ex
NXST_FLOW reply (xid=0x4):
cookie=0x0, duration=6770.969s, table=0, n_packets=5411, n_byt
s=306944, idle_age=0, hard_age=65534, priority=0 actions=NORMAL
```

VXLAN 模式

VXLAN 模式下，网络的架构跟 GRE 模式类似，所不同的是，不同节点之间通过 VXLAN 隧道互通，即虚拟化层是采用的 VXLAN 协议。

基本结构如下图所示。



其中，节点网络配置如下所示，注意数据网络接口需要 IP 地址，因为是隧道协议需要底下的三层转发支持。

控制节点

- eth0: 9.186.100.77/24 作为管理网络（同时也是公共网络）。
- eth1: 10.0.100.77/24 作为数据网络接口。

计算节点

- eth0: 9.186.100.88/24 作为管理网络（同时也是公共网络）。
- eth1: 10.0.100.88/24 作为数据网络接口。

计算节点

主要包括两个网桥：集成网桥 br-int 和 隧道网桥 br-tun。

```
$ sudo ovs-vsctl show
225f3eb5-6059-4063-99c3-8666915c9c55
    Bridge br-int
        fail_mode: secure
        Port br-int
            Interface br-int
                type: internal
            Port "qvoc4493802-43"
                tag: 1
                Interface "qvoc4493802-43"
            Port patch-tun
                Interface patch-tun
                    type: patch
                    options: {peer=patch-int}
            Port "qvof47c62b0-db"
                tag: 1
                Interface "qvof47c62b0-db"
        Bridge br-tun
            fail_mode: secure
            Port "vxlan-0a00644d"
                Interface "vxlan-0a00644d"
                    type: vxlan
                    options: {df_default="true", in_key=flow, local_ip="10.0.100.88", out_key=flow, remote_ip="10.0.100.77"}
            Port patch-int
                Interface patch-int
                    type: patch
                    options: {peer=patch-tun}
            Port br-tun
                Interface br-tun
                    type: internal
    ovs_version: "2.0.2"
```

安全网桥可以通过 `brctl show` 命令看到，该网桥主要用于绑定控制组的 `iptables` 规则，跟转发无直接关系。

```
~$ brctl show
bridge name      bridge id          STP enabled    interfaces
es
qbrf47c62b0-db      8000.56a7904c418d    no
qvbf47c62b0-db
                    tapf47c6
2b0-db
```

br-int

集成网桥 br-int 规则比较简单，作为一个正常的二层交换机使用。无论下面虚拟化层是哪种技术实现，集成网桥是看不到的，只知道根据 vlan 和 mac 进行转发。

所连接接口除了从安全网桥过来的 qvo-xxx（每个虚拟机会有一个），就是一个往外的 patch-tun 接口，连接到 br-tun 网桥。

其中，qvo-xxx 接口上会为每个网络分配一个内部 vlan 号，比如这里是同一个网络启动了两台虚机，所以 tag 都为 1。

```
Bridge br-int
    fail_mode: secure
    Port br-int
        Interface br-int
            type: internal
    Port "qvoc4493802-43"
        tag: 1
        Interface "qvoc4493802-43"
    Port patch-tun
        Interface patch-tun
            type: patch
            options: {peer=patch-int}
    Port "qvof47c62b0-db"
        tag: 1
        Interface "qvof47c62b0-db"
```

转发规则表 0 中是对所有包进行 NORMAL，表 23 中是所有包直接丢弃（是否后面将安全组规则在这里实现？）。

```
$ sudo ovs-ofctl dump-flows br-int
NXST_FLOW reply (xid=0x4):
  cookie=0x0, duration=52889.682s, table=0, n_packets=161, n_bytes
  =39290, idle_age=13, priority=1 actions=NORMAL
  cookie=0x0, duration=52889.451s, table=23, n_packets=0, n_bytes
  =0, idle_age=52889, priority=0 actions=drop
```


br-tun

br-tun 作为虚拟化层网桥，规则就要复杂一些。要将内部过来的网包进行合理甄别，内部带着正确 `vlan tag` 过来的，从正确的 `tunnel` 扔出去；外面带着正确 `tunnel` 号过来的，要改到对应的内部 `vlan tag` 扔到里面。

```
Bridge br-tun
    fail_mode: secure
    Port "vxlan-0a00644d"
        Interface "vxlan-0a00644d"
            type: vxlan
            options: {df_default="true", in_key=flow, local_ip="10.0.100.88", out_key=flow, remote_ip="10.0.100.77"}
    Port patch-int
        Interface patch-int
            type: patch
            options: {peer=patch-tun}
    Port br-tun
        Interface br-tun
            type: internal
```

其中，端口 `br-tun` 是内部端口，`vxlan-0a00644d` 这样的端口是向其它节点发包时候的 `VXLAN` 隧道端点，`patch-int` 端口通过一条管道连接到 `br-int` 上的 `patch-tun` 端口。

正常情况下，虚拟机的流量经过 `br-int` 转发，经过 `patch-tun` 端口，抵达 `patch-int` 端口，从而到达 `br-tun` 网桥，该网桥根据自身规则将合适的网包经过 `VXLAN` 隧道送出去。

```
$ sudo ovs-ofctl dump-flows br-tun
NXST_FLOW reply (xid=0x4):
  cookie=0x0, duration=329.194s, table=0, n_packets=31, n_bytes=2
906, idle_age=29, priority=1,in_port=1 actions=resubmit(,2)
  cookie=0x0, duration=325.847s, table=0, n_packets=14, n_bytes=1
591, idle_age=33, priority=1,in_port=2 actions=resubmit(,4)
  cookie=0x0, duration=328.954s, table=0, n_packets=6, n_bytes=48
0, idle_age=321, priority=0 actions=drop
  cookie=0x0, duration=328.712s, table=2, n_packets=9, n_bytes=69
4, idle_age=33, priority=0,dl_dst=00:00:00:00:00:00/01:00:00:00:
00:00 actions=resubmit(,20)
  cookie=0x0, duration=328.465s, table=2, n_packets=22, n_bytes=2
212, idle_age=29, priority=0,dl_dst=01:00:00:00:00:00/01:00:00:0
0:00:00 actions=resubmit(,22)
  cookie=0x0, duration=328.223s, table=3, n_packets=0, n_bytes=0,
idle_age=328, priority=0 actions=drop
  cookie=0x0, duration=50.703s, table=4, n_packets=12, n_bytes=14
51, idle_age=33, priority=1,tun_id=0x3e9 actions=mod_vlan_vid:1,
resubmit(,10)
  cookie=0x0, duration=327.979s, table=4, n_packets=2, n_bytes=14
0, idle_age=94, priority=0 actions=drop
  cookie=0x0, duration=327.742s, table=10, n_packets=12, n_bytes=
1451, idle_age=33, priority=1 actions=learn(table=20,hard_timeou
t=300,priority=1,NXM_OF_VLAN_TCI[0..11],NXM_OF_ETH_DST[])=NXM_OF_
ETH_SRC[],load:0->NXM_OF_VLAN_TCI[],load:NXM_NX_TUN_ID[]->NXM_NX
_TUN_ID[],output:NXM_OF_IN_PORT[]),output:1
  cookie=0x0, duration=38.551s, table=20, n_packets=9, n_bytes=69
4, hard_timeout=300, idle_age=33, hard_age=33, priority=1,vlan_t
ci=0x0001/0x0fff,dl_dst=fa:16:3e:83:95:fa actions=load:0->NXM_OF
_VLAN_TCI[],load:0x3e9->NXM_NX_TUN_ID[],output:2
  cookie=0x0, duration=327.504s, table=20, n_packets=0, n_bytes=0
, idle_age=327, priority=0 actions=resubmit(,22)
  cookie=0x0, duration=50.94s, table=22, n_packets=11, n_bytes=13
34, idle_age=29, dl_vlan=1 actions=strip_vlan,set_tunnel:0x3e9,o
utput:2
  cookie=0x0, duration=327.261s, table=22, n_packets=10, n_bytes=
808, idle_age=51, priority=0 actions=drop
```

这些规则组成如下图所示的转发逻辑。

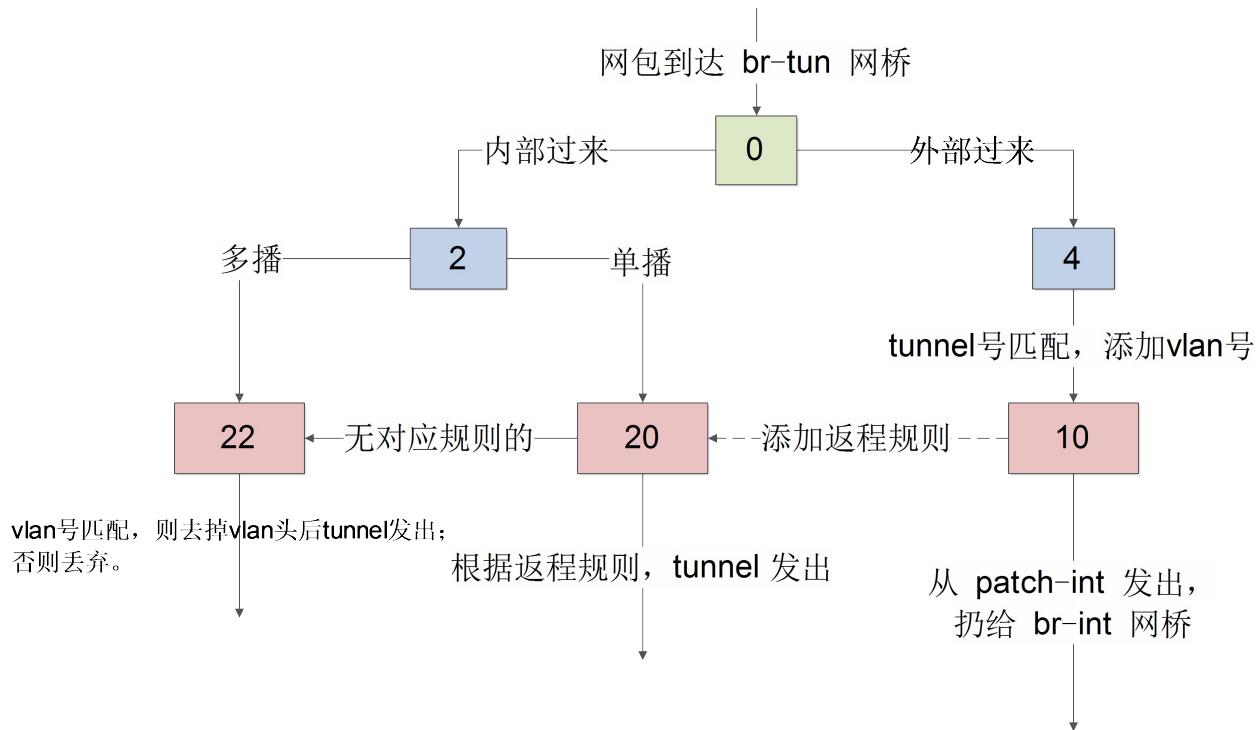


表 0

先看 table0 中的规则

```

cookie=0x0, duration=329.194s, table=0, n_packets=31, n_bytes=2
906, idle_age=29, priority=1, in_port=1 actions=resubmit(,2)
cookie=0x0, duration=325.847s, table=0, n_packets=14, n_bytes=1
591, idle_age=33, priority=1, in_port=2 actions=resubmit(,4)
cookie=0x0, duration=328.954s, table=0, n_packets=6, n_bytes=48
0, idle_age=321, priority=0 actions=drop
  
```

从 1 端口 (patch-int) 进来的网包，扔给表 2 处理，从 2 端口 (vxlan-0a00644d) 进来的网包，扔给表 4 处理。即一个处理来自内部 vm 的，一个处理来自外面的 vxlan 隧道的。

表 2

对于内部包，表 2 中规则为

```

cookie=0x0, duration=53316.397s, table=2, n_packets=0, n_bytes=0,
idle_age=53316, priority=0, dl_dst=00:00:00:00:00:00/01:00:00:00:00:00
actions=resubmit(,20)
cookie=0x0, duration=53316.162s, table=2, n_packets=161, n_bytes=39562,
idle_age=422, priority=0, dl_dst=01:00:00:00:00:00/00:01:00:00:00:00
actions=resubmit(,22)

```

即里面过来的单播包，扔给表 20 处理；多播和广播包，扔给表 22 处理。

表 3

丢弃所有包。

```

cookie=0x0, duration=328.223s, table=3, n_packets=0, n_bytes=0,
idle_age=328, priority=0 actions=drop

```

表 4

对于外部来的数据，表 4 中规则为

```

cookie=0x0, duration=50.703s, table=4, n_packets=12, n_bytes=1451,
idle_age=33, priority=1, tun_id=0x3e9 actions=mod_vlan_vid:1,
resubmit(,10)
cookie=0x0, duration=327.979s, table=4, n_packets=2, n_bytes=140,
idle_age=94, priority=0 actions=drop

```

匹配给定的 tunnel 号，添加对应的 vlan 号，扔给表 10 去学习一下后扔到 br-int 网桥。

表 10

```

cookie=0x0, duration=327.742s, table=10, n_packets=12, n_bytes=1451,
idle_age=33, priority=1 actions=learn(table=20, hard_timeout=300,
priority=1, NXM_OF_VLAN_TCI[0..11], NXM_OF_ETH_DST[] = NXM_OF_ETH_SRC[],
load:0->NXM_OF_VLAN_TCI[], load:NXM_NX_TUN_ID[] -> NXM_NX_TUN_ID[],
output:NXM_OF_IN_PORT[]), output:1

```

主要作用是学习外部（从 tunnel）进来的包，往表 20 中添加对返程包的正常转发规则，并且从 patch-int 扔给 br-int。

使用了 openvswitch 的 learn 动作。该动作能根据处理的流来动态修改其它表中的规则。

具体来看 learn 规则。

- `table=20` 说明是修改表 20 中的规则，后面是添加的规则内容；
- `NXM_OF_VLAN_TCI[0..11]`，匹配跟当前流同样的 VLAN 头，其中 NXM 是 Nicira Extensible Match 的缩写；
- `NXM_OF_ETH_DST[] = NXM_OF_ETH_SRC[]`，包的目的 mac 跟当前流的源 mac 匹配；
- `load:0->NXM_OF_VLAN_TCI[]`，将 vlan 号改为 0；
- `load:NXM_NX_TUN_ID[]->NXM_NX_TUN_ID[]`，将 tunnel 号修改为当前的 tunnel 号；
- `output:NXM_OF_IN_PORT[]`，从当前入口发出。

表 20

```
cookie=0x0, duration=38.551s, table=20, n_packets=9, n_bytes=694, hard_timeout=300, idle_age=33, hard_age=33, priority=1, vlan_tci=0x0001/0x0fff, dl_dst=fa:16:3e:83:95:fa actions=load:0->NXM_OF_VLAN_TCI[], load:0x3e9->NXM_NX_TUN_ID[], output:2
cookie=0x0, duration=327.504s, table=20, n_packets=0, n_bytes=0, idle_age=327, priority=0 actions=resubmit(,22)
```

其中，第一条规则就是表 10 学习来的结果。对于 vlan 号为 1，目标 mac 是 fa:16:3e:83:95:fa（之前，我们从虚拟机内 ping 10.0.0.1，这个 mac 作为源 mac 从 tunnel 来过）的网包，去掉 vlan 号，添加当时的 vxlan 号，并从 tunnel 发出。

对于没学习到规则的网包，则扔给表 22 处理。

表 22

```
cookie=0x0, duration=50.94s, table=22, n_packets=11, n_bytes=13  
34, idle_age=29, dl_vlan=1 actions=strip_vlan, set_tunnel:0x3e9, o  
utput:2  
cookie=0x0, duration=327.261s, table=22, n_packets=10, n_bytes=  
808, idle_age=51, priority=0 actions=drop
```

表 22 检查如果 `vlan` 号正确，则去掉 `vlan` 头后从 `tunnel` 扔出去。

网络节点

网络节点担负着进行网络服务的任务，包括DHCP、路由和高级网络服务等。一般包括三个网桥：br-tun、br-int 和 br-ex。

```
$ sudo ovs-vsctl show
49761e8e-031f-4a60-b838-28bb82aac7b7
  Bridge br-int
    fail_mode: secure
    Port br-int
      Interface br-int
        type: internal
    Port "qr-694450d6-f6"
      tag: 1
      Interface "qr-694450d6-f6"
        type: internal
    Port "tap13685e28-b0"
      tag: 1
      Interface "tap13685e28-b0"
        type: internal
    Port patch-tun
      Interface patch-tun
        type: patch
        options: {peer=patch-int}
  Bridge br-ex
    Port br-ex
      Interface br-ex
        type: internal
    Port "qg-e76de35e-90"
      Interface "qg-e76de35e-90"
        type: internal
  Bridge br-tun
    fail_mode: secure
    Port br-tun
      Interface br-tun
        type: internal
    Port "vxlan-0a006458"
      Interface "vxlan-0a006458"
```

```
    type: vxlan
        options: {df_default="true", in_key=flow, local_
ip="10.0.100.77", out_key=flow, remote_ip="10.0.100.88"}
    Port patch-int
        Interface patch-int
            type: patch
            options: {peer=patch-tun}
ovs_version: "2.0.2"
```

br-tun

跟计算节点类似，br-tun 作为虚拟化层网桥。要将内部过来的网包进行合理甄别，内部带着正确 `vlan tag` 过来的，从正确的 `tunnel` 扔出去；外面带着正确 `tunnel` 号过来的，要改到对应的内部 `vlan tag` 扔到里面。

包括两个接口，跟其它接点形成 `tunnel` 的 `vxlan-xxx` 端口，以及跟 `br-int` 互连的 `patch-int` 端口。

```
Bridge br-tun
    fail_mode: secure
    Port br-tun
        Interface br-tun
            type: internal
        Port "vxlan-0a006458"
            Interface "vxlan-0a006458"
                type: vxlan
                options: {df_default="true", in_key=flow, local_
ip="10.0.100.77", out_key=flow, remote_ip="10.0.100.88"}
        Port patch-int
            Interface patch-int
                type: patch
                options: {peer=patch-tun}
```

其中，端口 `br-tun` 是内部端口，`vxlan-0a00644d` 这样的端口是向其它节点发包时候的 `VXLAN` 隧道端点，`patch-int` 端口通过一条管道连接到 `br-int` 上的 `patch-tun` 端口。

查看 `br-tun` 上的转发规则。

```
$ sudo ovs-ofctl dump-flows br-tun
NXST_FLOW reply (xid=0x4):
  cookie=0x0, duration=422.153s, table=0, n_packets=1073, n_bytes
=107845, idle_age=96, priority=1,in_port=1 actions=resubmit(,2)
  cookie=0x0, duration=185.009s, table=0, n_packets=1076, n_bytes
=102922, idle_age=96, priority=1,in_port=2 actions=resubmit(,4)
  cookie=0x0, duration=421.853s, table=0, n_packets=6, n_bytes=48
0, idle_age=62414, priority=0 actions=drop
  cookie=0x0, duration=421.552s, table=2, n_packets=1047, n_bytes
=105725, idle_age=96, priority=0,dl_dst=00:00:00:00:00:00/01:00:
00:00:00:00 actions=resubmit(,20)
  cookie=0x0, duration=421.252s, table=2, n_packets=26, n_bytes=2
120, idle_age=61953, priority=0,dl_dst=01:00:00:00:00:00/01:00:0
0:00:00:00 actions=resubmit(,22)
  cookie=0x0, duration=420.939s, table=3, n_packets=0, n_bytes=0,
idle_age=62420, priority=0 actions=drop
  cookie=0x0, duration=394.249s, table=4, n_packets=1076, n_bytes
=102922, idle_age=96, priority=1,tun_id=0x3e9 actions=mod_vlan_v
id:1,resubmit(,10)
  cookie=0x0, duration=420.628s, table=4, n_packets=0, n_bytes=0,
idle_age=62420, priority=0 actions=drop
  cookie=0x0, duration=420.304s, table=10, n_packets=1076, n_byt
e=102922, idle_age=96, priority=1 actions=learn(table=20,hard_ti
meout=300,priority=1,NXM_OF_VLAN_TCI[0..11],NXM_OF_ETH_DST[]>NX
M_OF_ETH_SRC[],load:0->NXM_OF_VLAN_TCI[],load:NXM_NX_TUN_ID[]>NX
M_NX_TUN_ID[],output:NXM_OF_IN_PORT[]),output:1
  cookie=0x0, duration=1314.149s, table=20, n_packets=1006, n_byt
es=101338, hard_timeout=300, idle_age=96, hard_age=95, priority=
1,vlan_tci=0x0001/0x0fff,dl_dst=fa:16:3e:52:7a:f2 actions=load:0
->NXM_OF_VLAN_TCI[],load:0x3e9->NXM_NX_TUN_ID[],output:2
  cookie=0x0, duration=419.977s, table=20, n_packets=0, n_bytes=0
, idle_age=62419, priority=0 actions=resubmit(,22)
  cookie=0x0, duration=184.683s, table=22, n_packets=3, n_bytes=2
30, idle_age=61953, dl_vlan=1 actions=strip_vlan,set_tunnel:0x3e
9,output:2
  cookie=0x0, duration=419.668s, table=22, n_packets=23, n_bytes=
1890, idle_age=61961, priority=0 actions=drop
```

这些规则跟计算节点上的 br-tun 网桥规则类似，组成如下图所示的转发逻辑。

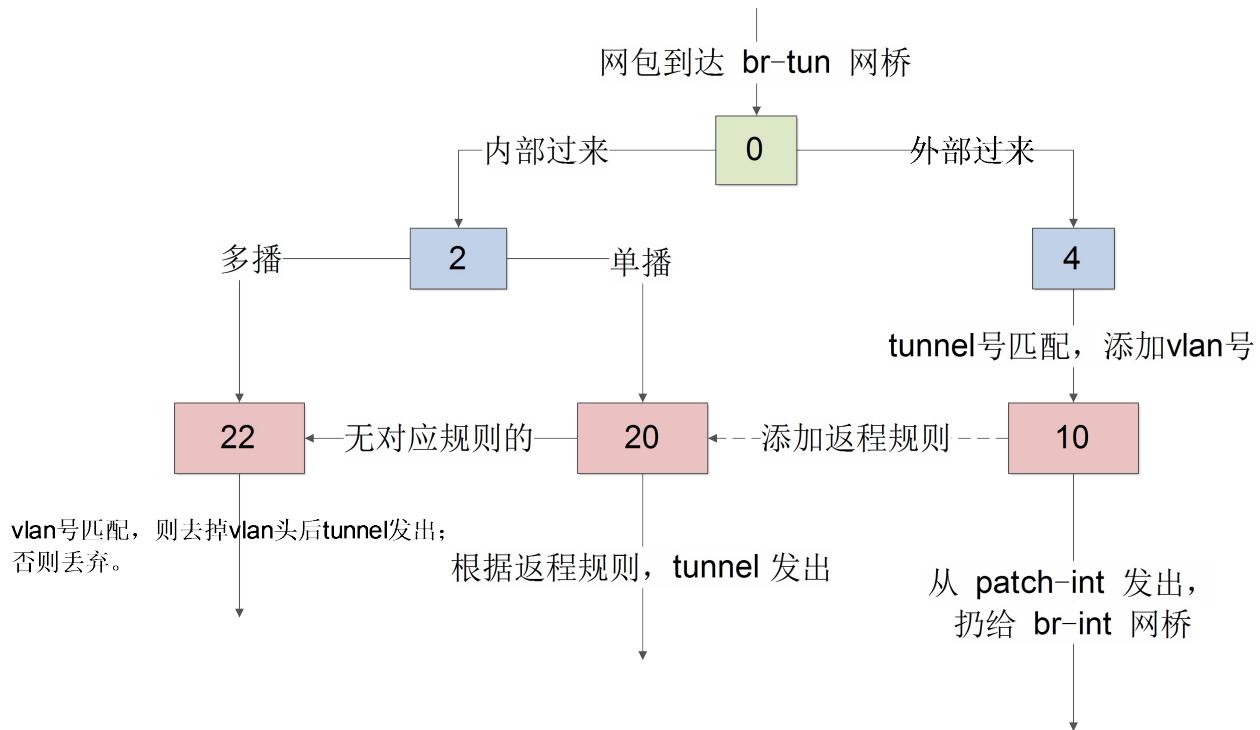


表 0

先看 table0 中的规则

```

cookie=0x0, duration=422.153s, table=0, n_packets=1073, n_bytes
=107845, idle_age=96, priority=1, in_port=1 actions=resubmit(,2)
cookie=0x0, duration=185.009s, table=0, n_packets=1076, n_bytes
=102922, idle_age=96, priority=1, in_port=2 actions=resubmit(,4)
cookie=0x0, duration=421.853s, table=0, n_packets=6, n_bytes=48
0, idle_age=62414, priority=0 actions=drop

```

从 1 端口 (patch-int) 进来的网包，扔给表 2 处理，从 2 端口 (vxlan-0a00644d) 进来的网包，扔给表 4 处理。即一个处理来自内部 br-int 的（这上面挂载着所有的网络服务，包括路由、DHCP 等），一个处理来自外面的 VXLAN 隧道的。

表 2

对于内部包，表 2 中规则为

```

cookie=0x0, duration=421.552s, table=2, n_packets=1047, n_bytes
=105725, idle_age=96, priority=0, dl_dst=00:00:00:00:00:00/01:00:
00:00:00 actions=resubmit(,20)
cookie=0x0, duration=421.252s, table=2, n_packets=26, n_bytes=2
120, idle_age=61953, priority=0, dl_dst=01:00:00:00:00:00/01:00:0
0:00:00 actions=resubmit(,22)

```

即里面过来的单播包，扔给表 20 处理；多播和广播包，扔给表 22 处理。

表 3

丢弃所有包。

```

cookie=0x0, duration=420.939s, table=3, n_packets=0, n_bytes=0,
idle_age=62420, priority=0 actions=drop

```

表 4

对于外部来的数据，表 4 中规则为

```

cookie=0x0, duration=394.249s, table=4, n_packets=1076, n_bytes
=102922, idle_age=96, priority=1, tun_id=0x3e9 actions=mod_vlan_v
id:1, resubmit(,10)
cookie=0x0, duration=420.628s, table=4, n_packets=0, n_bytes=0,
idle_age=62420, priority=0 actions=drop

```

匹配给定的 tunnel 号，添加对应的 vlan 号，扔给表 10 去学习一下后扔到 br-int 网桥。

表 10

```

cookie=0x0, duration=420.304s, table=10, n_packets=1076, n_byt
e=102922, idle_age=96, priority=1 actions=learn(table=20, hard_t
imeout=300, priority=1, NXM_OF_VLAN_TCI[0..11], NXM_OF_ETH_DST[] = NXM
_OF_ETH_SRC[], load:0->NXM_OF_VLAN_TCI[], load:NXM_NX_TUN_ID[] -> NX
M_NX_TUN_ID[], output:NXM_OF_IN_PORT[]), output:1

```

主要作用是学习外部（从 tunnel）进来的包，往表 20 中添加对返程包的正常转发规则，并且从 patch-int 扔给 br-int。

使用了 openvswitch 的 learn 动作。该动作能根据处理的流来动态修改其它表中的规则。

具体来看 learn 规则。

- `table=20` 说明是修改表 20 中的规则，后面是添加的规则内容；
- `NXM_OF_VLAN_TCI[0..11]`，匹配跟当前流同样的 VLAN 头，其中 NXM 是 Nicira Extensible Match 的缩写；
- `NXM_OF_ETH_DST[] = NXM_OF_ETH_SRC[]`，包的目的 mac 跟当前流的源 mac 匹配；
- `load:0->NXM_OF_VLAN_TCI[]`，将 vlan 号改为 0；
- `load:NXM_NX_TUN_ID[]->NXM_NX_TUN_ID[]`，将 tunnel 号修改为当前的 tunnel 号；
- `output:NXM_OF_IN_PORT[]`，从当前入口发出。

表 20

```
cookie=0x0, duration=1314.149s, table=20, n_packets=1006, n_bytes=
s=101338, hard_timeout=300, idle_age=96, hard_age=95, priority=1
,vlan_tci=0x0001/0xffff,d1_dst=fa:16:3e:52:7a:f2 actions=load:0-
->NXM_OF_VLAN_TCI[],load:0x3e9->NXM_NX_TUN_ID[],output:2
cookie=0x0, duration=419.977s, table=20, n_packets=0, n_bytes=0
, idle_age=62419, priority=0 actions=resubmit(,22)
```

其中，第一条规则就是表 10 学习来的结果。对于 vlan 号为 1，目标 mac 是 fa:16:3e:83:95:fa（之前，我们从虚拟机内 ping 10.0.0.1，这个 mac 作为源 mac 从 tunnel 来过）的网包，去掉 vlan 号，添加当时的 vxlan 号，并从 tunnel 发出。

对于没学习到规则的网包，则扔给表 22 处理。

表 22

```
cookie=0x0, duration=184.683s, table=22, n_packets=3, n_bytes=2  
30, idle_age=61953, dl_vlan=1 actions=strip_vlan, set_tunnel:0x3e  
9, output:2  
cookie=0x0, duration=419.668s, table=22, n_packets=23, n_bytes=  
1890, idle_age=61961, priority=0 actions=drop
```

表 22 检查如果 `vlan` 号正确，则去掉 `vlan` 头后从 `tunnel` 扔出去。

br-int

集成网桥 br-int 规则比较简单，作为一个正常的二层交换机使用。无论下面虚拟化层是哪种技术实现，集成网桥是看不到的，只知道根据 vlan 和 mac 进行转发。

所连接接口包括：

- tap-xxx，连接到网络 DHCP 服务的命名空间；
- qr-xxx，连接到路由服务的命名空间；
- 往外的 patch-tun 接口，连接到 br-tun 网桥。

其中网络服务接口上会绑定内部 vlan 号，每个号对应一个网络。

```
Bridge br-int
    fail_mode: secure
    Port br-int
        Interface br-int
            type: internal
    Port "qr-694450d6-f6"
        tag: 1
        Interface "qr-694450d6-f6"
            type: internal
    Port "tap13685e28-b0"
        tag: 1
        Interface "tap13685e28-b0"
            type: internal
    Port patch-tun
        Interface patch-tun
            type: patch
            options: {peer=patch-int}
```

转发规则表 0 中是对所有包进行 NORMAL，表 23 中是所有包直接丢弃（是否后面将安全组规则在这里实现？）。

```
$ sudo ovs-ofctl dump-flows br-int
NXST_FLOW reply (xid=0x4):
  cookie=0x0, duration=52889.682s, table=0, n_packets=161, n_bytes
  =39290, idle_age=13, priority=1 actions=NORMAL
  cookie=0x0, duration=52889.451s, table=23, n_packets=0, n_bytes
  =0, idle_age=52889, priority=0 actions=drop
```

br-ex

核心接口有两个。

一个是挂载的物理接口上，如 `eth0`，网包将从这个接口发送到外部网络上。

另外一个是 `qg-xxx` 这样的接口，是连接到 `router` 服务的网络名字空间中，里面绑定一个路由器的外部 IP，作为 `nAT` 时候的地址，另外，网络中的 `floating IP` 也放在这个网络名字空间中。

```
Bridge br-ex
    Port "eth0"
        Interface "eth0"
    Port br-ex
        Interface br-ex
            type: internal
    Port "qg-e76de35e-90"
        Interface "qg-e76de35e-90"
            type: internal
```

网桥的规则也很简单，作为一个正常的二层转发设备即可。

```
$ sudo ovs-ofctl dump-flows br-ex
NXST_FLOW reply (xid=0x4):
  cookie=0x0, duration=75072.257s, table=0, n_packets=352212, n_bytes=85641148, idle_age=0, hard_age=65534, priority=0 actions=NORMAL
```

网络名字空间

在 Linux 中，网络名字空间可以被认为是隔离的拥有单独网络栈（网卡、路由转发表、iptables）的环境。网络名字空间经常用来隔离网络设备和服务，只有拥有同样网络名字空间的设备，才能看到彼此。

可以用`ip netns list`命令来查看已经存在的名字空间。

```
$ ip net  
qdhcp-ea3928dc-b1fd-4a1a-940e-82b8c55214e6  
qrouter-40fff075-d3a2-477b-942c-6b1adb42e35e
```

`qdhcp`开头的名字空间是`dhcp`服务器使用的，`qrouter`开头的则是`router`服务使用的。可以通过 `ip netns exec namespaceid command` 来在指定的网络名字空间中执行网络命令，例如

```
# ip netns exec qdhcp-88b1609c-68e0-49ca-a658-f1edff54a264 ip ad  
dr  
71: ns-f14c598d-98: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 q  
disc pfifo_fast state UP qlen 1000  
    link/ether fa:16:3e:10:2f:03 brd ff:ff:ff:ff:ff:ff  
    inet 10.1.0.3/24 brd 10.1.0.255 scope global ns-f14c598d-98  
        inet6 fe80::f816:3eff:fe10:2f03/64 scope link  
            valid_lft forever preferred_lft forever
```

可以看到，`dhcp`服务的网络名字空间中只有一个网络接口“`ns-f14c598d-98`”，它连接到`br-int`的`tapf14c598d-98`接口上。

DHCP 服务

dhcp服务是通过dnsmasq进程（轻量级服务器，可以提供dns、dhcp、tftp等服务）来实现的，该进程绑定到dhcp名字空间中的br-int的接口上。可以查看相关的进程。

```
# ps -fe | grep 88b1609c-68e0-49ca-a658-f1edff54a264
nobody    23195      1  0 Oct26 ?          00:00:00 dnsmasq --no-hos
ts --no-resolv --strict-order --bind-interfaces --interface=ns-f
14c598d-98 --except-interface=lo --pid-file=/var/lib/neutron/dhc
p/88b1609c-68e0-49ca-a658-f1edff54a264/pid --dhcp-hostsfile=/var
/lib/neutron/dhcp/88b1609c-68e0-49ca-a658-f1edff54a264/host --dh
cp-optfile=/var/lib/neutron/dhcp/88b1609c-68e0-49ca-a658-f1edff
54a264/opts --dhcp-script=/usr/bin/neutron-dhcp-agent-dnsmasq-le
ase-update --leasefile-ro --dhcp-range=tag0,10.1.0.0,static,120s
--conf-file= --domain=openstacklocal
root      23196 23195  0 Oct26 ?          00:00:00 dnsmasq --no-hos
ts --no-resolv --strict-order --bind-interfaces --interface=ns-f
14c598d-98 --except-interface=lo --pid-file=/var/lib/neutron/dhc
p/88b1609c-68e0-49ca-a658-f1edff54a264/pid --dhcp-hostsfile=/var
/lib/neutron/dhcp/88b1609c-68e0-49ca-a658-f1edff54a264/host --dh
cp-optfile=/var/lib/neutron/dhcp/88b1609c-68e0-49ca-a658-f1edff
54a264/opts --dhcp-script=/usr/bin/neutron-dhcp-agent-dnsmasq-le
ase-update --leasefile-ro --dhcp-range=tag0,10.1.0.0,static,120s
--conf-file= --domain=openstacklocal
```

路由服务

首先，要理解什么是 router，router是提供跨 subnet 的互联功能的。比如用户的内部网络中主机想要访问外部互联网的地址，就需要router来转发（因此，所有跟外部网络的流量都必须经过router）。目前router的实现是通过iptables进行的。

同样的，router服务也运行在自己的名字空间中，可以通过如下命令查看：

```
$ sudo ip net exec qrouter-40fff075-d3a2-477b-942c-6b1adb42e35e
ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
49: qr-694450d6-f6: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UNKNOWN group default
    link/ether fa:16:3e:5d:18:10 brd ff:ff:ff:ff:ff:ff
    inet 10.0.0.1/24 brd 10.0.0.255 scope global qr-694450d6-f6
        valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:fe5d:1810/64 scope link
        valid_lft forever preferred_lft forever
50: qg-e76de35e-90: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UNKNOWN group default
    link/ether fa:16:3e:70:24:92 brd ff:ff:ff:ff:ff:ff
    inet 9.186.100.2/24 brd 9.186.100.255 scope global qg-e76de35e-90
        valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:fe70:2492/64 scope link
        valid_lft forever preferred_lft forever
```

可以看出，该名字空间中包括两个网络接口。

第一个接口 qr-694450d6-f6 (10.0.0.1) 跟 br-int 上的接口相连。即任何从 br-int 来的找 10.0.0.1 (租户的私有网段) 的网包都会到达这个接口。

第一个接口 qg-e76de35e-90 连接到 br-ex 上的接口，即任何从外部来的网包，询问 9.186.100.2 (默认的静态 NAT 外部地址) 或 9.186.100.129 (租户申请的 floating IP 地址)，都会到达这个接口。

查看该名字空间中的路由表：

```
$ sudo ip net exec qrouter-40fff075-d3a2-477b-942c-6b1adb42e35e
ip route
default via 9.186.100.1 dev qg-e76de35e-90
9.186.100.0/24 dev qg-e76de35e-90 proto kernel scope link src
9.186.100.2
10.0.0.0/24 dev qr-694450d6-f6 proto kernel scope link src 10
.0.0.1
```

默认情况，以及访问外部网络的时候，体会从 qg-xxx 接口发出，经过 br-ex 发布到外网。

访问租户内网的时候，会从 qr-xxx 接口发出，发给 br-int。

```
$ sudo ip net exec qrouter-40fff075-d3a2-477b-942c-6b1adb42e35e
iptables -t nat -S
-P PREROUTING ACCEPT
-P INPUT ACCEPT
-P OUTPUT ACCEPT
-P POSTROUTING ACCEPT
-N neutron-postrouting-bottom
-N neutron-vpn-agen-OUTPUT
-N neutron-vpn-agen-POSTROUTING
-N neutron-vpn-agen-PREROUTING
-N neutron-vpn-agen-float-snat
-N neutron-vpn-agen-snat
-A PREROUTING -j neutron-vpn-agen-PREROUTING
-A OUTPUT -j neutron-vpn-agen-OUTPUT
-A POSTROUTING -j neutron-vpn-agen-POSTROUTING
-A POSTROUTING -j neutron-postrouting-bottom
-A neutron-postrouting-bottom -j neutron-vpn-agen-snat
-A neutron-vpn-agen-OUTPUT -d 9.186.100.129/32 -j DNAT --to-destination 10.0.0.2
-A neutron-vpn-agen-POSTROUTING ! -i qg-e76de35e-90 ! -o qg-e76de35e-90 -m conntrack ! --ctstate DNAT -j ACCEPT
-A neutron-vpn-agen-PREROUTING -d 169.254.169.254/32 -p tcp -m tcp --dport 80 -j REDIRECT --to-ports 9697
-A neutron-vpn-agen-PREROUTING -d 9.186.100.129/32 -j DNAT --to-destination 10.0.0.2
-A neutron-vpn-agen-float-snat -s 10.0.0.2/32 -j SNAT --to-source 9.186.100.129
-A neutron-vpn-agen-snat -j neutron-vpn-agen-float-snat
-A neutron-vpn-agen-snat -s 10.0.0.0/24 -j SNAT --to-source 9.186.100.2
```

其中SNAT和DNAT规则完成外部 floating ip (9.186.100.129) 到内部 ip (10.0.0.2) 的映射：

```
-A neutron-vpn-agen-OUTPUT -d 9.186.100.129/32 -j DNAT --to-destination 10.0.0.2  
-A neutron-vpn-agen-PREROUTING -d 9.186.100.129/32 -j DNAT --to-destination 10.0.0.2  
-A neutron-vpn-agen-float-snat -s 10.0.0.2/32 -j SNAT --to-source 9.186.100.129
```

另外有一条SNAT规则把所有其他的内部IP出来的流量都映射到外部IP

9.186.100.2。这样即使在内部虚拟机没有外部IP的情况下，也可以发起对外网的访问。

```
-A neutron-vpn-agen-snat -s 10.0.0.0/24 -j SNAT --to-source 9.186.100.2
```

安全组

Security group通过Linux IPtables来实现，为此，在Compute节点上引入了qbr*这样的Linux传统bridge（iptables规则目前无法加载到直接挂在到ovs的tap设备上）。首先在Control节点上用neutron port-list命令列出虚拟机的端口id，例如：

```
# neutron port-list
+-----+-----+
| id | name | mac_address |
| fixed_ips |
|       |
+-----+-----+
| 2a169bb4-4d8b-4c67-802c-a24bdaf1312 | fa:16:3e:2f:e9:7
2 | {"subnet_id": "a2456a2c-5eea-416d-8757-d10bc0aa2aaa", "ip_address": "192.168.0.1"} |
| 583c7038-d341-41ec-a0d1-0cd2c33866ca | fa:16:3e:9c:dc:3
a | {"subnet_id": "a2456a2c-5eea-416d-8757-d10bc0aa2aaa", "ip_address": "192.168.0.2"} |
| 9b2db4ac-3145-401c-8dc6-486ca6e303b6 | fa:16:3e:4e:f1:b
5 | {"subnet_id": "ea4ed31b-e05a-4735-8c3f-9b430e656b64", "ip_address": "192.168.122.200"} |
| c5a7d51b-9934-40bd-befa-adff840462d2 | fa:16:3e:21:1d:0
0 | {"subnet_id": "ea4ed31b-e05a-4735-8c3f-9b430e656b64", "ip_address": "192.168.122.201"} |
| db2f5a49-7c0d-45dd-acad-908931f9a654 | fa:16:3e:17:5c:3
6 | {"subnet_id": "a2456a2c-5eea-416d-8757-d10bc0aa2aaa", "ip_address": "192.168.0.3"} |
+-----+-----+
```

其中id的前10位数字被用作虚机对外连接的qbr（同时也是tap口）的id。i或o加上前9位数字被用作安全组chain的id。

所有的规则默认都在Compute节点上的filter表（默认表）中实现，分别来查看filter表的INPUT、OUTPUT、FORWARD三条链上的规则。

在Compute节点上，可以用 `iptables --line-numbers -vnL [CHAIN]` 来获得filter表（可以指定某个链上的）规则。

INPUT

```
#iptables --line-numbers -vnL INPUT
Chain INPUT (policy ACCEPT 0 packets, 0 bytes)
num  pkts bytes target     prot opt in      out      source
      destination
1    360K  56M neutron-openvswi-INPUT  all  --  *      *
      0.0.0.0/0          0.0.0.0/0
2    10583 2146K ACCEPT     tcp   --  *      *      192.168.122.
100      0.0.0.0/0          multiport dports 5666 /* 001 nagios
-nrpe incoming 192.168.122.100 */
3     846 50966 ACCEPT     tcp   --  *      *      192.168.122.
100      0.0.0.0/0          multiport dports 5900:5999 /* 001 n
ova compute incoming 192.168.122.100 */
4    1033K  894M ACCEPT     all  --  *      *      0.0.0.0/0
      0.0.0.0/0          state RELATED,ESTABLISHED
5     760 63840 ACCEPT     icmp  --  *      *      0.0.0.0/0
      0.0.0.0/0
6       1    60 ACCEPT     all  --  lo      *      0.0.0.0/0
      0.0.0.0/0
7     977 58620 ACCEPT     tcp   --  *      *      0.0.0.0/0
      0.0.0.0/0          state NEW tcp dpt:22
8    3899 1194K REJECT     all  --  *      *      0.0.0.0/0
      0.0.0.0/0          reject-with icmp-host-prohibited
```

可以看到，跟安全组相关的规则被重定向到neutron-openvswi-INPUT。查看其规则，只有一条。

```
#iptables --line-numbers -vnL neutron-openvswi-INPUT
Chain neutron-openvswi-INPUT (1 references)
num  pkts bytes target     prot opt in      out      source
      destination
1       0     0 neutron-openvswi-o583c7038-d  all  --  *      *
      0.0.0.0/0          0.0.0.0/0          PHYSDEV match --
physdev-in tap583c7038-d3 --physdev-is-bridged
```

重定向到neutron-openvswi-o583c7038-d。

```
#iptables --line-numbers -vnL neutron-openvswi-o583c7038-d
Chain neutron-openvswi-o583c7038-d (2 references)
num  pkts bytes target      prot opt in     out      source
      destination
1    3894 1199K RETURN      udp   --   *       *       0.0.0.0/0
      0.0.0.0/0          udp spt:68 dpt:67
2    4282 1536K neutron-openvswi-s583c7038-d  all   --   *       *
      0.0.0.0/0          0.0.0.0/0
3     0     0 DROP         udp   --   *       *       0.0.0.0/0
      0.0.0.0/0          udp spt:67 dpt:68
4     0     0 DROP         all   --   *       *       0.0.0.0/0
      0.0.0.0/0          state INVALID
5    3971 1510K RETURN      all   --   *       *       0.0.0.0/0
      0.0.0.0/0          state RELATED,ESTABLISHED
6    311  25752 RETURN      all   --   *       *       0.0.0.0/0
      0.0.0.0/0
7     0     0 neutron-openvswi-sg-fallback  all   --   *       *
      0.0.0.0/0          0.0.0.0/0
```

如果是vm发出的dhcp请求，直接通过，否则转到neutron-openvswi-s583c7038-d。

```
#iptables --line-numbers -vnL neutron-openvswi-s583c7038-d
Chain neutron-openvswi-s583c7038-d (1 references)
num  pkts bytes target      prot opt in     out      source
      destination
1    4284 1537K RETURN      all   --   *       *       192.168.0.2
      0.0.0.0/0          MAC FA:16:3E:9C:DC:3A
2     0     0 DROP         all   --   *       *       0.0.0.0/0
      0.0.0.0/0
```

这条chain主要检查从vm发出来的网包，是否是openstack所分配的IP和MAC，如果不匹配，则禁止通过。这将防止利用vm上进行一些伪装地址的攻击。

OUTPUT

```
#iptables --line-numbers -vnL OUTPUT
Chain OUTPUT (policy ACCEPT 965K packets, 149M bytes)
num  pkts bytes target     prot opt in     out      source
      destination
1    481K  107M neutron-filter-top  all  --  *      *
      0.0.0/0          0.0.0.0/0
2    481K  107M neutron-openvswi-OUTPUT  all  --  *      *
      0.0.0.0/0          0.0.0.0/0
```

分别跳转到neutron-filter-top和neutron-openvswi-OUTPUT。

```
#iptables --line-numbers -vnL neutron-filter-top
Chain neutron-filter-top (2 references)
num  pkts bytes target     prot opt in     out      source
      destination
1    497K  112M neutron-openvswi-local  all  --  *      *
      0.0.0.0/0          0.0.0.0/0
```

跳转到neutron-openvswi-local。

```
#iptables --line-numbers -vnL neutron-openvswi-OUTPUT
Chain neutron-openvswi-OUTPUT (1 references)
num  pkts bytes target     prot opt in     out      source
      destination
```

该chain目前无规则。

```
#iptables --line-numbers -vnL neutron-openvswi-local
Chain neutron-openvswi-local (1 references)
num  pkts bytes target     prot opt in     out      source
      destination
```

该chain目前也无规则。

OUTPUT

FORWARD

FORWARD chain上主要实现安全组的功能。用户在配置缺省安全规则时候（例如允许ssh到vm，允许ping到vm），影响该chain。

```
#iptables --line-numbers -vnL FORWARD
Chain FORWARD (policy ACCEPT 0 packets, 0 bytes)
num  pkts bytes target     prot opt in      out      source
      destination
1    16203 5342K neutron-filter-top  all  --  *      *      0.0
     .0.0/0          0.0.0.0/0
2    16203 5342K neutron-openvswi-FORWARD  all  --  *      *      *
     0.0.0.0/0          0.0.0.0/0
3      0      0 REJECT     all  --  *      *      0.0.0.0/0
     0.0.0.0/0          reject-with icmp-host-prohibited
```

同样跳转到neutron-filter-top，无规则。跳转到neutron-openvswi-FORWARD。

```
#iptables --line-numbers -vnL neutron-openvswi-FORWARD
Chain neutron-openvswi-FORWARD (1 references)
num  pkts bytes target     prot opt in      out      source
      destination
1    8170 2630K neutron-openvswi-sg-chain  all  --  *      *
     0.0.0.0/0          0.0.0.0/0          PHYSDEV match --phy
sdev-out tap583c7038-d3 --physdev-is-bridged
2    8156 2729K neutron-openvswi-sg-chain  all  --  *      *
     0.0.0.0/0          0.0.0.0/0          PHYSDEV match --phy
sdev-in tap583c7038-d3 --physdev-is-bridged
```

neutron-openvswi-FORWARD将匹配所有进出tap-XXX端口的流量。

```
#iptables --line-numbers -vnL neutron-openvswi-sg-chain
Chain neutron-openvswi-sg-chain (2 references)
num  pkts bytes target      prot opt in     out      source
      destination
1    8170 2630K neutron-openvswi-i583c7038-d  all  --  *      *
      0.0.0.0/0          0.0.0.0/0      PHYSDEV match --
physdev-out tap583c7038-d3 --physdev-is-bridged
2    8156 2729K neutron-openvswi-o583c7038-d  all  --  *      *
      0.0.0.0/0          0.0.0.0/0      PHYSDEV match --
physdev-in tap583c7038-d3 --physdev-is-bridged
3    12442 4163K ACCEPT      all  --  *      *      0.0.0.0/0
      0.0.0.0/0
```

如果是网桥从tap-XXX端口发出到VM的流量，则跳转到neutron-openvswi-i9LETTERID；如果是从tap-XXX端口进入到网桥的（即vm发出来的）流量，则跳转到neutron-openvswi-o9LETTERID。

```
#iptables --line-numbers -vnL neutron-openvswi-i583c7038-d
Chain neutron-openvswi-i583c7038-d (1 references)
num  pkts bytes target      prot opt in     out      source
      destination
1    0     0 DROP      all  --  *      *      0.0.0.0/0
      0.0.0.0/0      state INVALID
2    400  43350 RETURN      all  --  *      *      0.0.0.0/0
      0.0.0.0/0      state RELATED,ESTABLISHED
3    1     60 RETURN      tcp   --  *      *      0.0.0.0/0
      0.0.0.0/0      tcp dpt:22
4    1     84 RETURN      icmp  --  *      *      0.0.0.0/0
      0.0.0.0/0
5   3885 1391K RETURN      udp   --  *      *      192.168.0.3
      0.0.0.0/0      udp spt:67 dpt:68
6   3885 1197K neutron-openvswi-sg-fallback  all  --  *      *
      0.0.0.0/0      0.0.0.0/0
```

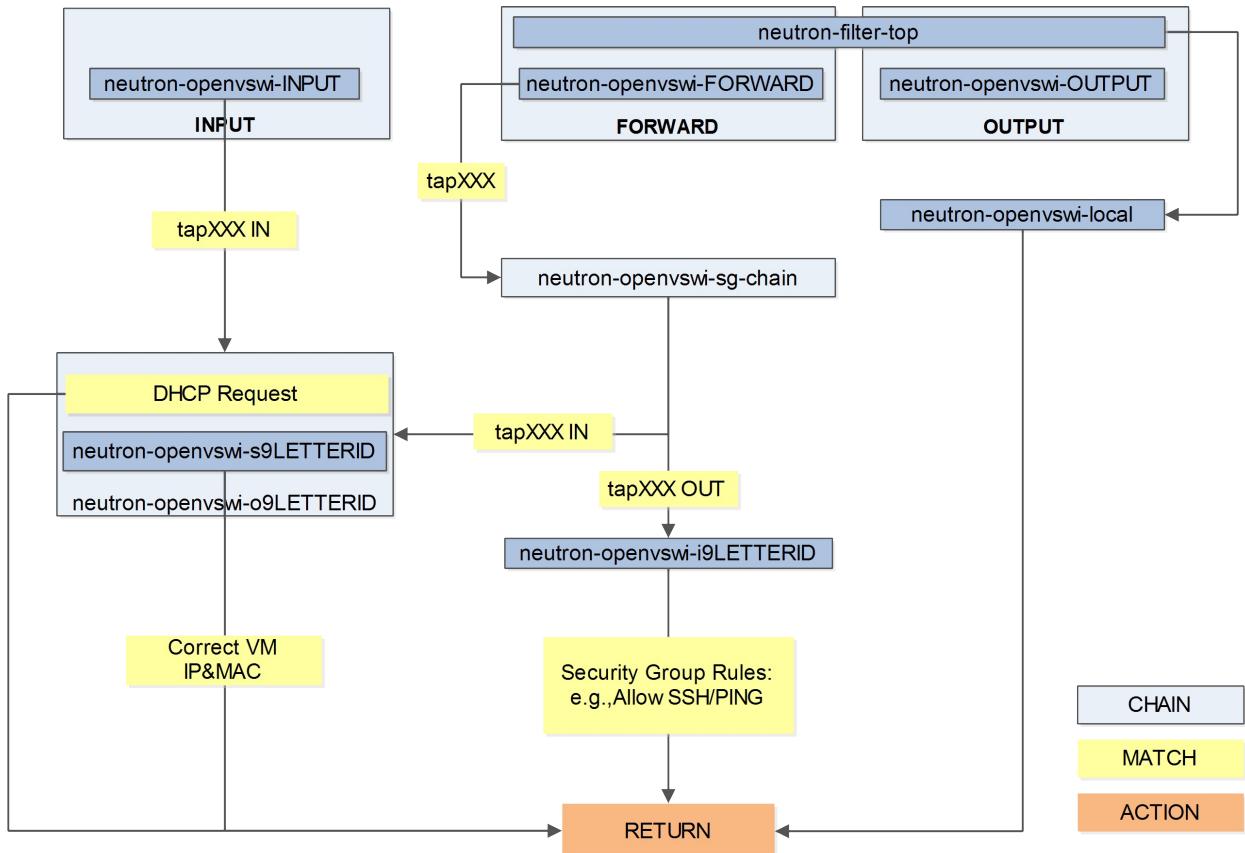
neutron-openvswi-i9LETTERID允许安全组中配置的策略（允许ssh、ping等）和dhcp reply通过。默认的neutron-openvswi-sg-fallback将drop所有流量。

```
#iptables --line-numbers -vnL neutron-openvswi-o583c7038-d
Chain neutron-openvswi-o583c7038-d (2 references)
num  pkts bytes target      prot opt in     out      source
      destination
1    3886  1197K RETURN      udp   --  *      *      0.0.0.0/0
      0.0.0.0/0          udp spt:68 dpt:67
2    4274  1533K neutron-openvswi-s583c7038-d  all   --  *      *
      0.0.0.0/0          0.0.0.0/0
3      0      0 DROP        udp   --  *      *      0.0.0.0/0
      0.0.0.0/0          udp spt:67 dpt:68
4      0      0 DROP        all   --  *      *      0.0.0.0/0
      0.0.0.0/0          state INVALID
5    3963  1507K RETURN      all   --  *      *      0.0.0.0/0
      0.0.0.0/0          state RELATED,ESTABLISHED
6    311   25752 RETURN      all   --  *      *      0.0.0.0/0
      0.0.0.0/0
7      0      0 neutron-openvswi-sg-fallback  all   --  *      *
      0.0.0.0/0          0.0.0.0/0
```

neutron-openvswi-o9LETTERID将跳转到neutron-openvswi-s583c7038-d，允许DHCP Request和匹配VM的源IP和源MAC的流量通过。

整体逻辑

整体逻辑如下图所示。



快速查找安全组规则

从前面分析可以看出，某个vm的安全组相关规则的chain的名字，跟vm的id的前9个字符有关。

因此，要快速查找qbr-XXX上相关的iptables规则，可以用iptables -S列出（默认是filter表）所有链上的规则，其中含有id的链即为虚拟机相关的安全组规则。其中--physdev-in表示即将进入某个网桥的端口，--physdev-out表示即将从某个网桥端口发出。

```
#iptables -S |grep tap583c7038-d3
-A neutron-openvswi-FORWARD -m physdev --physdev-out tap583c7038
-d3 --physdev-is-bridged -j neutron-openvswi-sg-chain
-A neutron-openvswi-FORWARD -m physdev --physdev-in tap583c7038-d
3 --physdev-is-bridged -j neutron-openvswi-sg-chain
-A neutron-openvswi-INPUT -m physdev --physdev-in tap583c7038-d3
--physdev-is-bridged -j neutron-openvswi-o583c7038-d
-A neutron-openvswi-sg-chain -m physdev --physdev-out tap583c7038
-d3 --physdev-is-bridged -j neutron-openvswi-i583c7038-d
-A neutron-openvswi-sg-chain -m physdev --physdev-in tap583c7038
-d3 --physdev-is-bridged -j neutron-openvswi-o583c7038-d
```

可以看出，进出tap-XXX口的FORWARD链上的流量都被扔到了neutron-openvswi-sg-chain这个链，neutron-openvswi-sg-chain上是security group具体的实现（两条规则，访问虚拟机的流量扔给neutron-openvswi-i583c7038-d；从虚拟机出来的扔给neutron-openvswi-o583c7038-d）。

其它

安全组在Havana版本中，默认是开启的，如果安装完毕后发现找不到qbr-*网桥，则可以检查在nova.conf里面是否设置以下内容：

```
libvirt_vif_driver=nova.virt.libvirt.vif.LibvirtHybridOVSBridgeDriver
```

负载均衡即服务

负载均衡即服务（Load Balance as a Service，LBaaS）是一项网络高级服务。

顾名思义，它允许租户动态的在自己的网络创建一个负载均衡设备。

负载均衡，可以说是分布式系统中比较基础的组件，它接收前端过来的请求，然后将请求按照某种均衡的策略转发给后端资源池中的某个处理单元，以完成处理。进而可以实现高可用性和横向的扩展性。

OpenStack Neutron 通过高级服务扩展的形式支持 LBaaS，目前默认是通过 [HAProxy](#) 软件来实现的。

典型场景

典型的场景，租户创建了一个网络，并在其上分配了一个子网 10.0.0.0/24，租户试图通过一个负载均衡设备来实现多个虚拟机呈现统一业务。

首先，启动两个虚机，分别分配到 IP 地址：10.0.0.2 和 10.0.0.4。

OpenStack 的 LBaaS 实现中有三个重要概念：

- Pool
- Member
- Monitor

其中，Pool 是要进行负载均衡的资源池（一般需要对应到某一个子网），可以指定负载均衡的提供机制（例如默认的 HAProxy）、均衡的协议（TCP、HTTP、HTTPS）和端口等、均衡的策略。

The screenshot shows the 'Pools' tab selected in the navigation bar. The main table displays one item:

Name	Description	Provider	Subnet	Protocol	Status	VIP	Actions
lb_pool		haproxy	10.0.0.0/24	TCP	ACTIVE	vip	<button>Edit Pool</button>

Below the table, a message says "Displaying 1 item".

定义完成 Pool 后可以往里面添加 Member（即虚拟机），并为各个成员分配权重，指定的目标端口等。

The screenshot shows the 'Members' tab selected in the navigation bar. The main table displays two items:

IP Address	Protocol Port	Weight	Pool	Status	Actions
10.0.0.4	22	1	lb_pool	ACTIVE	<button>Edit Member</button>
10.0.0.2	22	1	lb_pool	ACTIVE	<button>Edit Member</button>

Below the table, a message says "Displaying 2 items".

最后，定义一个 Monitor，负责定期探测成员的状态。

The screenshot shows the 'Monitors' tab selected in the navigation bar. The main table displays one item:

Monitor Type	Delay	Timeout	Max Retries	Details	Actions
PING	1	1	2	-	<button>Edit Monitor</button>

Below the table, a message says "Displaying 1 item".

例如，我们创建一个 Pool，添加子网 10.0.0.0/24，并添加启动的虚机 10.0.0.2 和 10.0.0.4 作为成员，创建一个 VIP（例如 10.0.0.250）来均衡 TCP 协议，端口 22 的请求。

这样，对地址 10.0.0.250 的 TCP 访问，只要是端口 22 的，会被自动负载均衡到虚机 10.0.0.2 和 10.0.0.4 上去。

实现细节

跟大部分的高级服务一样，LBaaS 在网络节点上实现。

qlbaas 命名空间

在启动了 LBaaS 之后，网络节点上会多一个 qlbaas-xxx 命名空间，其中一个 tap 类型端口，绑定了我们定义的 VIP。

```
$ sudo ip netns exec qlbaas-574a31da-a28a-449f-8c9d-3d3687c3c02a
ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
52: tapfb358342-59: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UNKNOWN group default
    link/ether fa:16:3e:68:77:37 brd ff:ff:ff:ff:ff:ff
    inet 10.0.0.250/24 brd 10.0.0.255 scope global tapfb358342-59
        valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:fe68:7737/64 scope link
        valid_lft forever preferred_lft forever
```

该 tap 类型端口，会连接到网络节点的 br-int 网桥上。

HAProxy

此时在网络节点上查看进程，会发现一个 haproxy 进程。

```
$ ps aux|grep haproxy
nobody 17331 0.0 0.0 20376 1104 ? Ss 15:08 0:00
haproxy -f /opt/stack/data/neutron/lbaas/574a31da-a28a-449f-8c9
d-3d3687c3c02a/conf -p /opt/stack/data/neutron/lbaas/574a31da-a2
8a-449f-8c9d-3d3687c3c02a/pid -sf 17323
```

查看 HAProxy 的配置文件

```
$ cat /opt/stack/data/neutron/lbaas/574a31da-a28a-449f-8c9d-3d36
87c3c02a/conf
global
    daemon
    user nobody
    group nogroup
    log /dev/log local0
    log /dev/log local1 notice
    stats socket /opt/stack/data/neutron/lbaas/574a31da-a28a
-449f-8c9d-3d3687c3c02a/sock mode 0666 level user
defaults
    log global
    retries 3
    option redispatch
    timeout connect 5000
    timeout client 50000
    timeout server 50000
frontend df25900f-53b1-4aab-8436-8411ec710445
    option tcplog
    bind 10.0.0.250:22
    mode tcp
    default_backend 574a31da-a28a-449f-8c9d-3d3687c3c02a
backend 574a31da-a28a-449f-8c9d-3d3687c3c02a
    mode tcp
    balance leastconn
    server 45481748-3104-462c-ae4b-fae0f4b08c20 10.0.0.2:22
    weight 1
    server 227b69fe-8685-4df6-b2a5-2f964c74886c 10.0.0.4:22
    weight 1
```

可见，里面的配置信息，跟我们之前配置的完全一致。

当有请求访问 10.0.0.250:22 的时候，最终会到达该 qlbaas 命名空间，请求被 HAProxy 收到，按照负载均衡策略，转发给合适的虚机。

可以通过 ssh 进行测试。

```
$ sudo ip netns exec qrouter-40fff075-d3a2-477b-942c-6b1adb42e35  
e ssh 10.0.0.250
```

在后端的虚机抓包，SSH 请求是 10.0.0.250 -> 10.0.0.2，说明实际上是 HAProxy 转发了请求。

其它问题

我们知道，虚拟机可以通过分配 Floating IP 的方式被外部直接访问到它内部的服务，那么经过负载均衡后，是否也能为 VIP 分配一个 floating IP 呢？

答案是肯定的。

可以先手动申请到一个 Floating IP，在绑定的时候选择 VIP 对应的端口即可。这样，我们就可以对外呈现我们的高可用、动态扩展的服务了。

防火墙即服务

熟悉防火墙的都知道，防火墙一般放在网关上，用来隔离子网之间的访问。因此，防火墙即服务（FireWall as a Service）也是在网络节点上（具体说来是在路由器命名空间中）来实现。

目前，OpenStack 中实现防火墙还是基于 Linux 系统自带的 `iptables`，所以大家对于其性能和功能就不要抱太大的期望了。

一个可能混淆的概念是安全组（Security Group），安全组的对象是虚拟网卡，由 L2 Agent 来实现，比如 `neutron_openvswitch_agent` 和 `neutron_linuxbridge_agent`，会在计算节点上通过配置 `iptables` 规则来限制虚拟网卡的进出访问。防火墙可以在安全组之前隔离外部过来的恶意流量，但是对于同个子网内部不同虚拟网卡间的通讯不能过滤（除非它要跨子网）。

可以同时部署防火墙和安全组实现双重防护。

典型场景

典型的场景，租户创建了一个网络，并在其上分配了一个子网 10.0.0.0/24，默认情况下，其它子网将不允许访问该子网。租户试图通过防火墙规则来允许外部网络对内部子网虚拟机 22 端口的访问。

OpenStack 的 FWaaS 实现中有三个重要概念：

- Firewall
- Policy
- Rule

其中，Firewall 会绑定到某个 Policy（因此必须先创建 Policy 之后才能创建一个 Firewall）。

The screenshot shows the 'Firewalls' tab selected in the top navigation bar. Below it, the 'Firewall Policies' tab is active. A table displays one policy entry:

Name	Policy	Status	Actions
fw1	p1	ACTIVE	<button>Edit Firewall</button>

At the bottom left, it says 'Displaying 1 item'.

Policy 中可以包括若干条 Rule。

The screenshot shows the 'Firewalls' tab selected in the top navigation bar. Below it, the 'Firewall Policies' tab is active. A table displays one rule entry:

Name	Rules	Audited	Actions
p1	r1	False	<button>Edit Policy</button>

At the bottom left, it says 'Displaying 1 item'.

Rule 即访问控制的规则，包括源和目的子网、源和目的端口、协议、行动等。

例如，我们创建一个 Rule，允许其它网络对内部子网中虚机端口 22 的 TCP 请求。

The screenshot shows the 'Firewalls' tab selected in the top navigation bar. Below it, the 'Firewall Rules' tab is active. A table displays one rule entry:

Name	Protocol	Source IP	Source Port	Destination IP	Destination Port	Action	Enabled	In Policy	Actions
r1	TCP	0.0.0.0/0	1:65535	10.0.0.0/24	22	ALLOW	True	p1	<button>Edit Rule</button>

At the bottom left, it says 'Displaying 1 item'.

实现细节

防火墙由 L3 Agent 通过修改 `iptables` 规则来实现，具体规则在网络节点的路由器命名空间中，作用到该租户所有路由器的 `qr-xxx` 接口上。

在网络节点上查看其中的 `iptables` 规则，会发现多了两个 `iptables` 链，分别处理进出两个方向的流量，被 `neutron-vpn-agen-FORWARD` 引用。

```
$ sudo ip netns exec qrrouter-40fff075-d3a2-477b-942c-6b1adb42e35  
e iptables -nvL  
Chain neutron-vpn-agen-iv4d8ed9898 (1 references)  
pkts bytes target      prot opt in      out      source  
destination  
0     0 DROP          all   --  *       *       0.0.0.0/0  
0.0.0.0/0           state INVALID  
0     0 ACCEPT         all   --  *       *       0.0.0.0/0  
0.0.0.0/0           state RELATED,ESTABLISHED  
0     0 ACCEPT         tcp    --  *       *       0.0.0.0/0  
10.0.0.0/24          tcp spts:1:65535 dpt:22  
  
Chain neutron-vpn-agen-ov4d8ed9898 (1 references)  
pkts bytes target      prot opt in      out      source  
destination  
0     0 DROP          all   --  *       *       0.0.0.0/0  
0.0.0.0/0           state INVALID  
0     0 ACCEPT         all   --  *       *       0.0.0.0/0  
0.0.0.0/0           state RELATED,ESTABLISHED  
0     0 ACCEPT         tcp    --  *       *       0.0.0.0/0  
10.0.0.0/24          tcp spts:1:65535 dpt:22
```

其它问题

其实像类似的高级服务，应该使用更为专业的实现，包括各种现成的防火墙硬件、软件实现。

但是很可惜的是，Neutron 中防火墙的位置跟路由器结合的过于紧密，造成要替换就要全部替换掉，即使用外部的网关实现。

其它高级服务也存在类似的问题，特别是支持了服务链之后，耦合的更严重了。

分布式路由

OpenStack 用户可能会发现，按照 Neutron 原先的设计，所有网络服务都在网络节点上进行，这意味着大量的流量和处理，给网络节点带来了很大的压力。

这些处理的核心是路由器服务。任何需要跨子网的访问都需要路由器进行路由。

很自然，能否让计算节点上也运行路由器服务？这个设计思路无疑是更为合理的，但具体实施起来需要诸多细节上的技术考量。

为了降低网络节点的负载，同时提高可扩展性，OpenStack 自 Juno 版本开始正式引入了分布式路由（Distributed Virtual Router，DVR）特性（用户可以选择使用与否），来让计算节点自己来处理原先的大量东西向流量和非 SNAT 南北流量（有 floating IP 的 vm 跟外面的通信）。

这样网络节点只需要处理占到一部分的 SNAT（无 floating IP 的 vm 跟外面的通信）流量，大大降低了负载和整个系统对网络节点的依赖。很自然的，FWaaS 也可以跟着放到计算节点上。

DHCP 服务、VPN 服务目前仍然需要集中在网络节点上进行。

典型场景

从网络的访问看，涉及到路由服务的至少是需要跨子网的访问，又包括是否是同一机器、是否是涉及到外网（东西向 vs 南北向）。

考虑下面几个跨子网路由的典型场景。

方向	同一机器	不同机器
东西	本地网桥处理	本地东西路由器
南北	本地南北路由器 floating 转发	网络节点 SNAT 转发

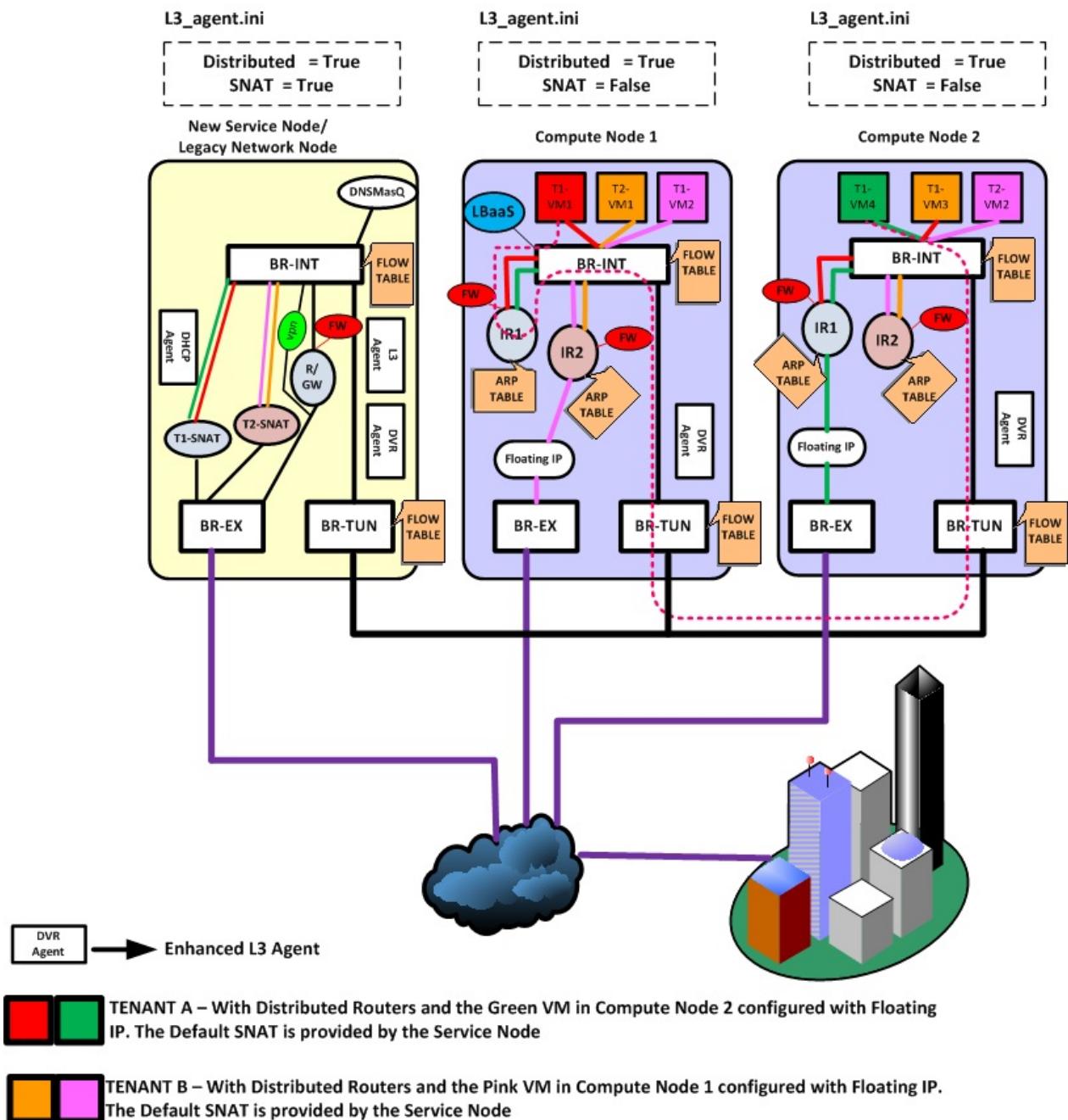
东西向

东西向意味着租户同一个数据中心内不同子网之间的互相访问。

同一机器

对于同一主机上的不同子网之间访问，路由器直接在 `br-int` 上转发即可，不需要经过外部网桥。

不同机器



如图所示，租户 T1 的两台虚拟机 VM1（计算节点 1）和 VM4（计算节点 2）分别属于不同子网，位于不同的计算节点。VM1 要访问 VM4，由计算节点 1 上的 IR1 起到路由器功能。返程的网包，则由计算节点 2 上的路由器 IR2 起作用。

两个路由器的 id、内部接口、功能等其实都是一样的。即同一个路由器，但是实际上在多个计算节点上同时存在。

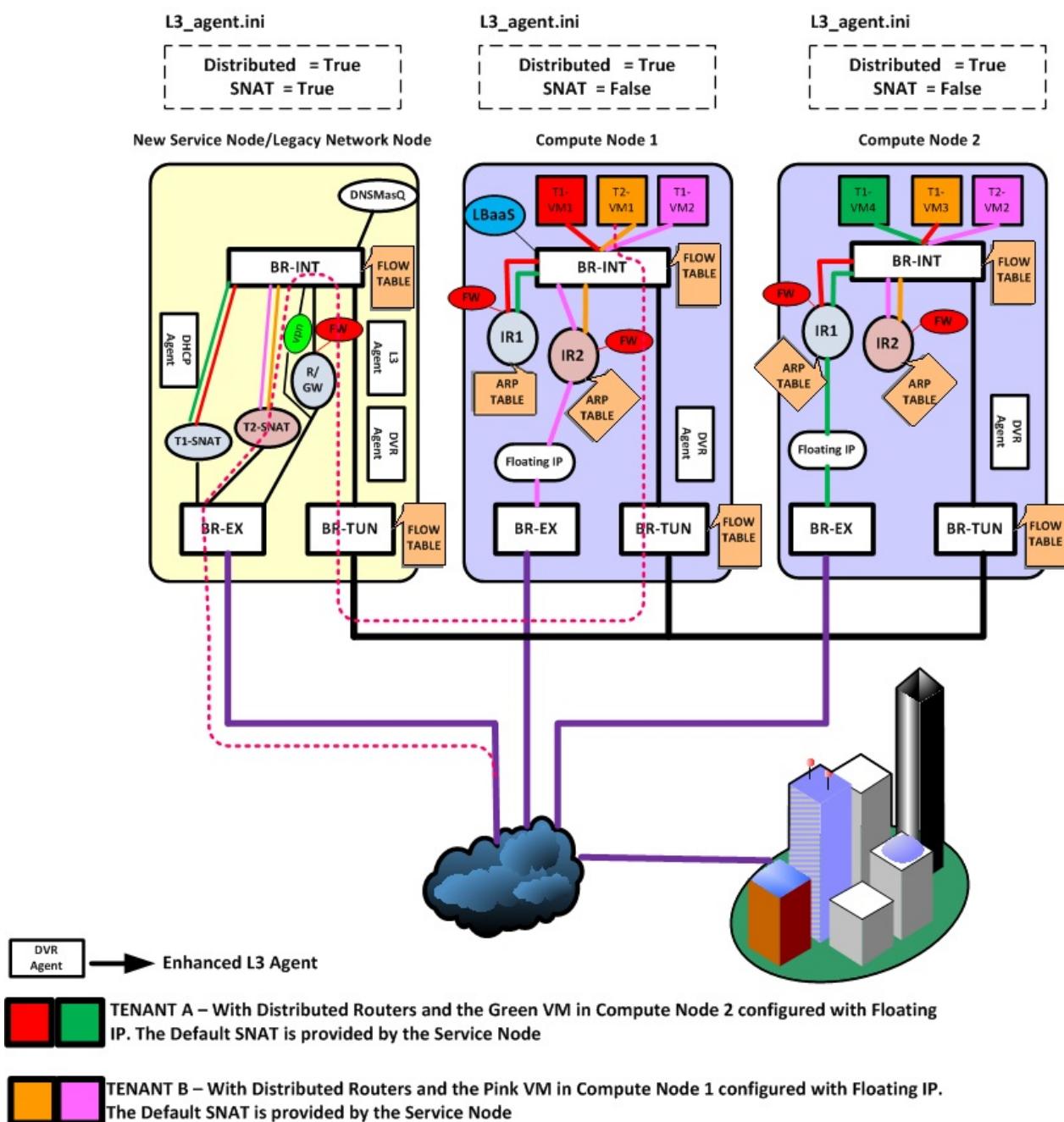
这里可能有人会想到，多台同样的路由器，如果都暴露在外部网络上，会出现冲突。例如当 VM1 的请求包离开计算节点 1 时，带的源 mac 是路由器目标接口的 mac，而这个 mac 在计算节点 2 上的路由器上同样存在。

因此，需要拦截路由器对外的暴露信息。一个是让每个路由器只应答本机的 mac 请求；另一个是绝对不让带着路由器 mac 地址的包直接扔出去。实现上在 br-int 上进行拦截，修改其源 mac 为 tunnel 端口的 mac。同样的，计算节点 2 在 br-int 上拦截源 mac 为这个 tunnel 端口的 mac，替换为正常的子网网关的 mac，直接扔给目标虚拟机所在的主机。

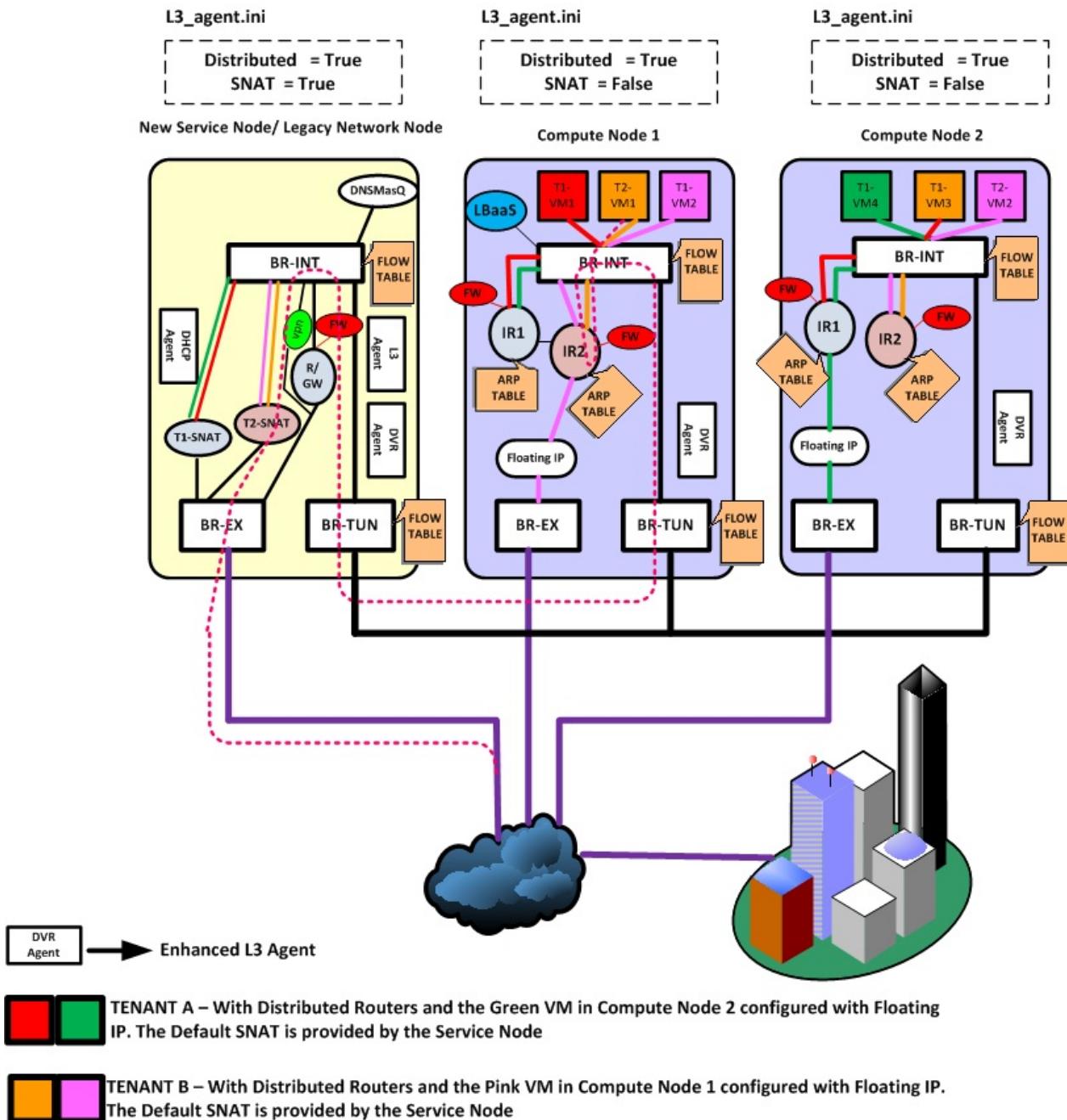
南北向

无 floating IP

这种情况（即 SNAT）下，跟传统模式很类似，首先来看外部访问内部子网。



租户 T2 在外部，通过默认的 SNAT 网关访问内部子网的 vm VM1。此时，网络节点上的 T2-SNAT 起到路由器的作用

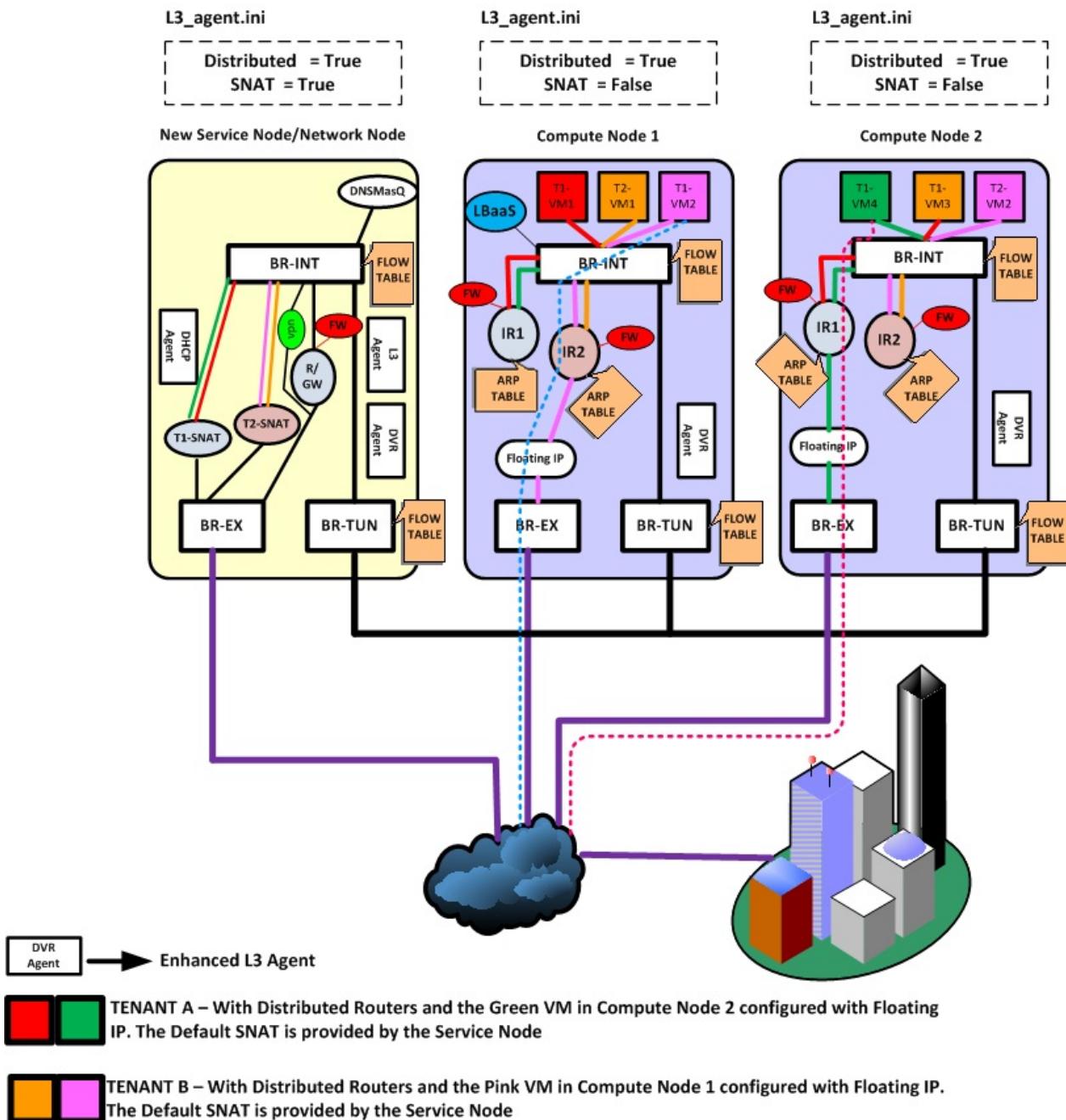


反过来，租户 T2 内部子网的 vm VM1 试图访问外部网络，则仍然经过网络节点上的 T2-SNAT 路由器。

为何这种情况下必须从网络节点走？这是因为对于外部网络来说，看到的都是外部接口的地址，这个地址只有一个。

当然，如果以后每个计算节点上都可以带有这样一个 SNAT 默认外部地址的话，这种情况下的流量也是可以直接从计算节点出去的。

有 floating IP



这种情况下，计算节点上的专门负责的外部路由器将负责进行转发，即计算节点 1 上的 IR2 和计算节点 2 上的 IR1。

网络节点

服务基本没变动，除了 L3 服务需要配置为 `dvr_snat` 模式。

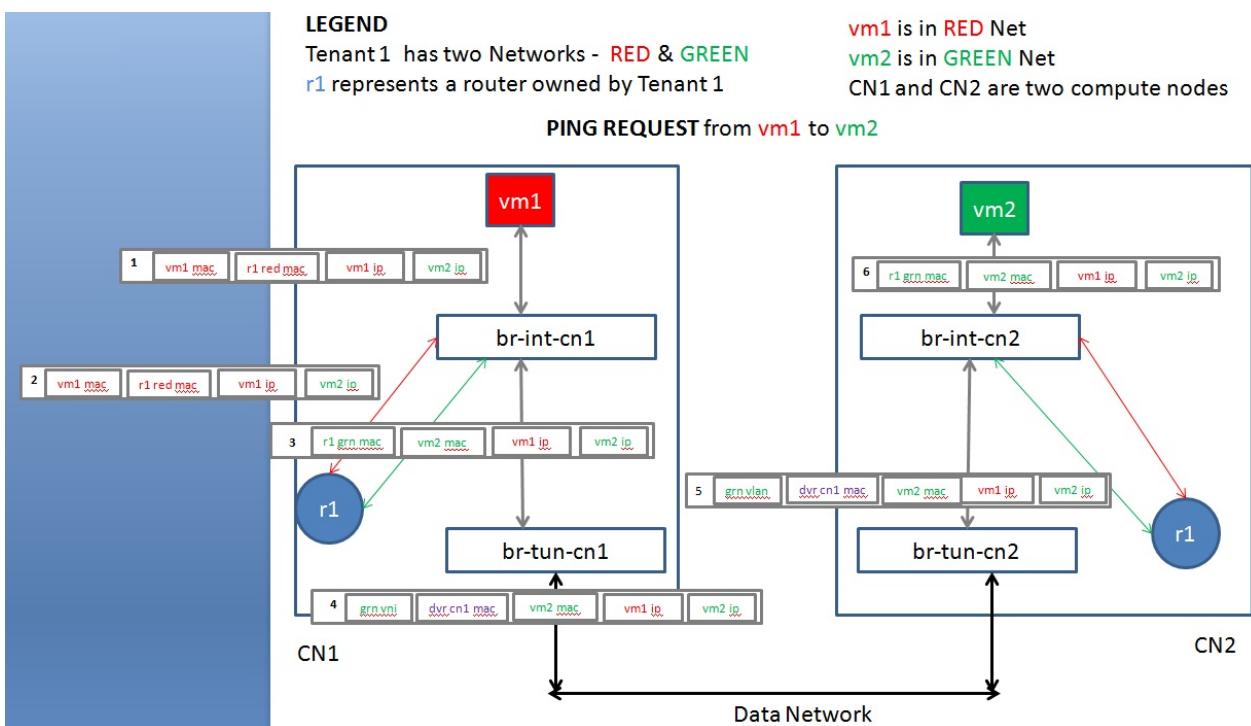
命名空间上会多一个专门的 `snat-xxx` 命名空间，处理来自计算节点的无 floating IP 的南北向流量。

计算节点

需要额外启用 `l3_agent` (dvr 模式) , 以及 `metadata agent`。

其实，跟传统情况下的网络节点十分类似。每个东西向路由器有自己的命名空间，负责跨子网的转发。另外，多一个 `floating` 路由器，专门负责经由 `floating` 地址的南北向转发。

东西流量



如上图所示，租户两个子网，红色和绿色，分别有 `vm1` 和 `vm2`，位于节点 `cn1` 和 `cn2` 上。

`vm1` 访问 `vm2` 的网包如步骤 1-6，整个过程 `ip` 保持不变。

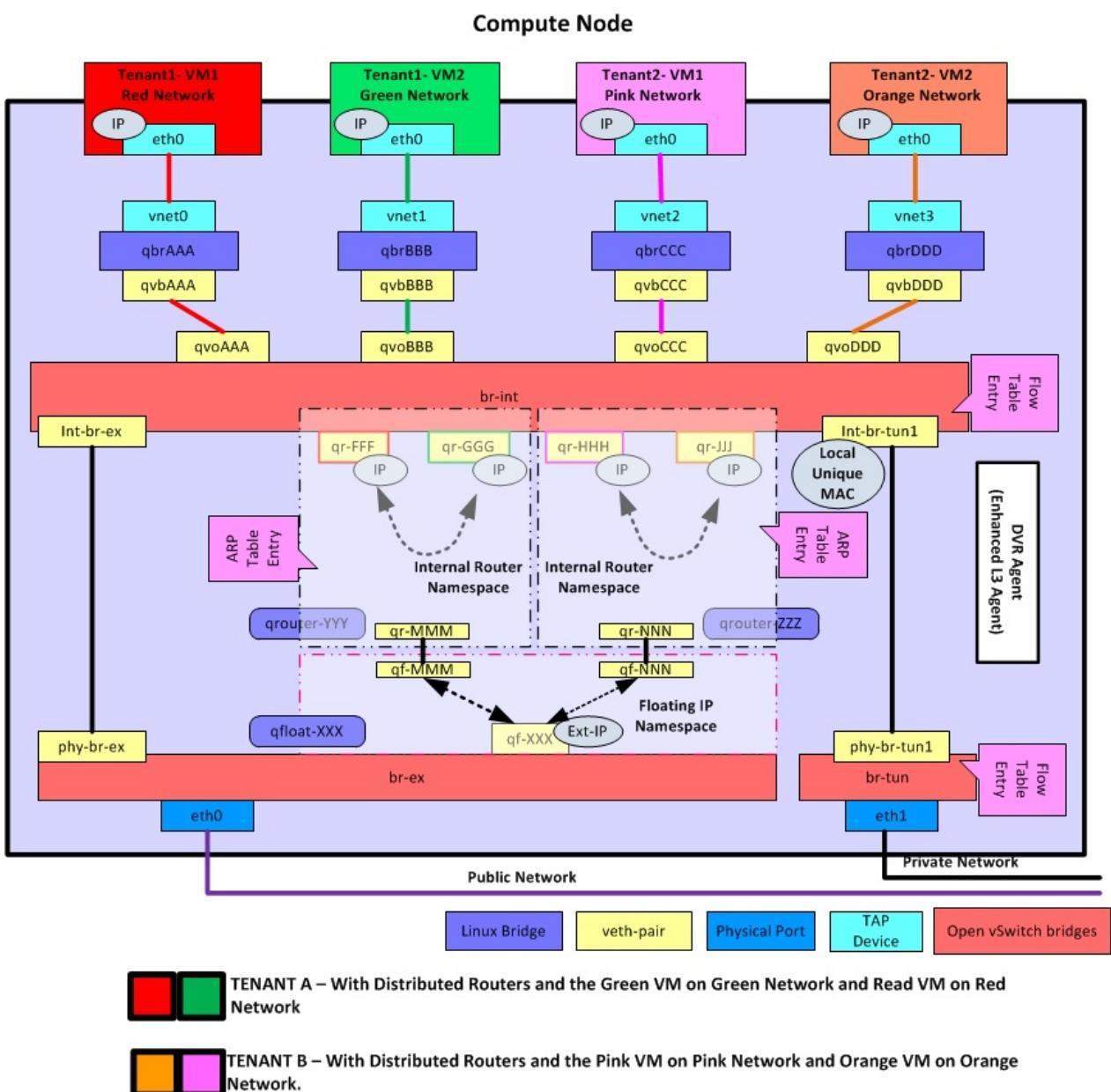
- 原始包，`vm1` 访问 `vm2`，目的 `mac` 为本地（红网）的路由器网关接口 `r1 red mac`；
- 经过 `br-int-cn1` 转发，该网包通过本地（红网）网关接口扔给本地路由器 `r1`。
- `r1` 根据路由规则，经过到绿网的接口发出，此时网包的源 `mac` 改为绿网的网关接口 `r1 grn mac`，目的 `mac` 改为 `vm2 mac`，并且带上绿网的本地 `vlan tag`；
- 网包发给 `br-tun-cn1` 进行 tunnel，扔出去之前，将源 `mac` 替换为跟节点相关

的特定 mac dvr cn1 mac，之后带着目标子网（绿网）的外部 tunnel id 扔出去（实现可以为 vlan、vxlan、gre 等，功能都是一样的）；

- 节点 cn2 的网桥 br-tun-cn2 会从 tunnel 收到这个包，解封包，带上本地 vlan tag，最终抵达网桥 br-int-cn2；
- br-int-cn2 上替换网包的源 mac（此时为 dvr-cn1-mac）为本地路由器的绿网网关接口，然后发给 vm2。

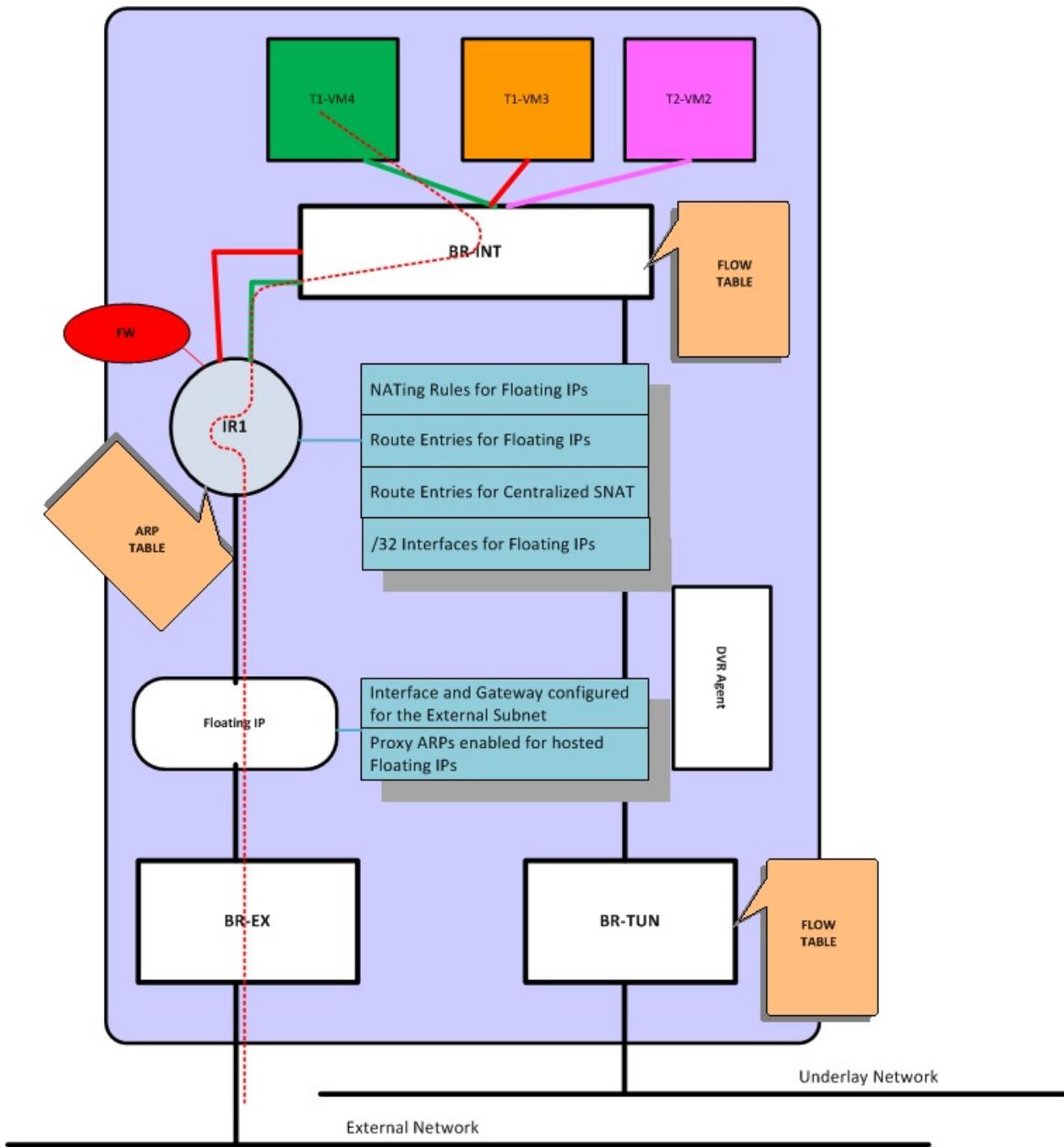
返回包的过程正好是反过来。虽然实现上略复杂，但整个过程还是比较清晰的，保证 vm1 和 vm2 感觉到的都是直接跟路由器的接口相连（分别为红网网关接口和绿网网关接口）。

南北流量



所不同的是，单独有一个 `qfloat-XXX` 路由器（也在一个独立命名空间中）来负责处理带有 floating IP 的南北向流量。

Floating IP Traffic Flow and Programming



流表规则

为了实现上面描述的这些功能，需要下面几种流表规则。

所有对网关的 **arp request** 不能出到外部网络

在 tunnel 网桥上看到的一律丢弃。

```
DVR PROCESS Table 1 (New table for dvr):  
table=1, priority=4, dl_vlan= red1-L-vlan, dl_type=arp, ar_tpa=  
r1-red-ip actions: drop  
table=1, priority=4, dl_vlan= grn1-L-vlan, dl_type=arp, ar_tpa=  
r1-grn-ip actions: drop
```

所有发往网关的包也不能出到外部网络

在 tunnel 网桥上看到的一律丢弃。

```
DVR PROCESS Table 1 (New table for dvr):  
table=1, priority=2, dl_vlan=red2_L_vlan, dl_dst=r1-red-mac, act  
ions: drop  
table=1, priority=2 , dl_vlan=grn2_L_vlan, dl_dst=r1-grn-mac, ac  
tions: drop
```

所有从路由器接口路由出去的包，需要修改源 **mac** 为特殊 **mac**

在 tunnel 网桥上进行处理。

```
DVR PROCESS Table 1 (New table for dvr):  
table=1, priority=1, dl_vlan=red2_L_vlan, dl_src=r1-red-mac, act  
ions: mod_dl_src=dvr-cn1-mac, resubmit(,2)  
table=1, priority=1, dl_vlan=grn2_L_vlan, dl_src=r1-grn-mac, act  
ions: mod_dl_src=dvr-cn1-mac, resubmit (,2)
```

同样，收到外部特殊源 **mac** 的包要修改为正确的路由器接口
mac

在 integration 网桥上进行处理。

```

Table 0: (Local switching table)
table=0, priority=2, in_port=patch-tun, dl_src=dvr-cn1-mac actions: goto table 1
table=0, priority=1, actions: output->NORMAL

Table 1: (DVR_TO_LOCALMAC table)
table=1, priority=2, dl_vlan=grn2-L-vlan, nw_dst=grn-subnet actions: strip_vlan, mod_dl_src=r1-grn-MAC,output->port-vm2
table=1, priority=1 actions: drop

```

此外，还有一些优化措施，包括：

采用 L2 pre-population 技术，提前把相关计算节点的地址关系放到本地的 FDB 表中，减少外部广播。

在 integration 网桥上，采用组表来调整规则顺序等。

```

Table 1: (DVR_TO_LOCALMAC table)
table=1, priority=2, dl_vlan=grn2-L-vlan, nw_dst=grn-subnet actions: strip_vlan, mod_dl_src=r1-grn-MAC,output->port-vm2

```

注意事项

这里面比较重要的地方，是允许有多个同样的网关存在于多个计算节点（同一子网跨多个物理节点导致）的情况下，保证：

- 要让本地的请求找到本地的路由器；
- 要避免路由器的接口 mac 地址直接暴露到外部网络上。

要解决这两个问题，首先是合理处理好本地的 ARP 请求，让本地对网关的请求拦截发给本地的路由器。同时，路由器路由后的网包，发到外部网络之前，先用一个跟节点绑定的特殊 mac 替换掉源 mac。这个 mac 地址是控制器为每个计算节点单独分配的唯一地址。

如果不允许同一子网跨多个物理节点，则每个节点上的路由器就不会冲突，设计起来就简单的多，比如 [kubernetes](#) 中的网络。

配置

要启用 DVR，比较简单，分别在各个节点的网络配置文件上做如下修改或添加。

Neutron Server

```
/etc/neutron/neutron.conf
```

```
router_distributed = True
```

L3 Agent

```
/etc/neutron/l3_agent.ini
```

```
agent_mode = [dvr_snat | dvr | legacy ]
```

网络节点上配置为 `dvr_snat`，计算节点上配置为 `dvr`。

L2 Agent

```
/etc/neutron/plugins/ml2/ml2_conf.ini
```

```
[ml2]
```

```
mechanism_drivers = openvswitch,linuxbridge,l2population
```

```
[agent]
```

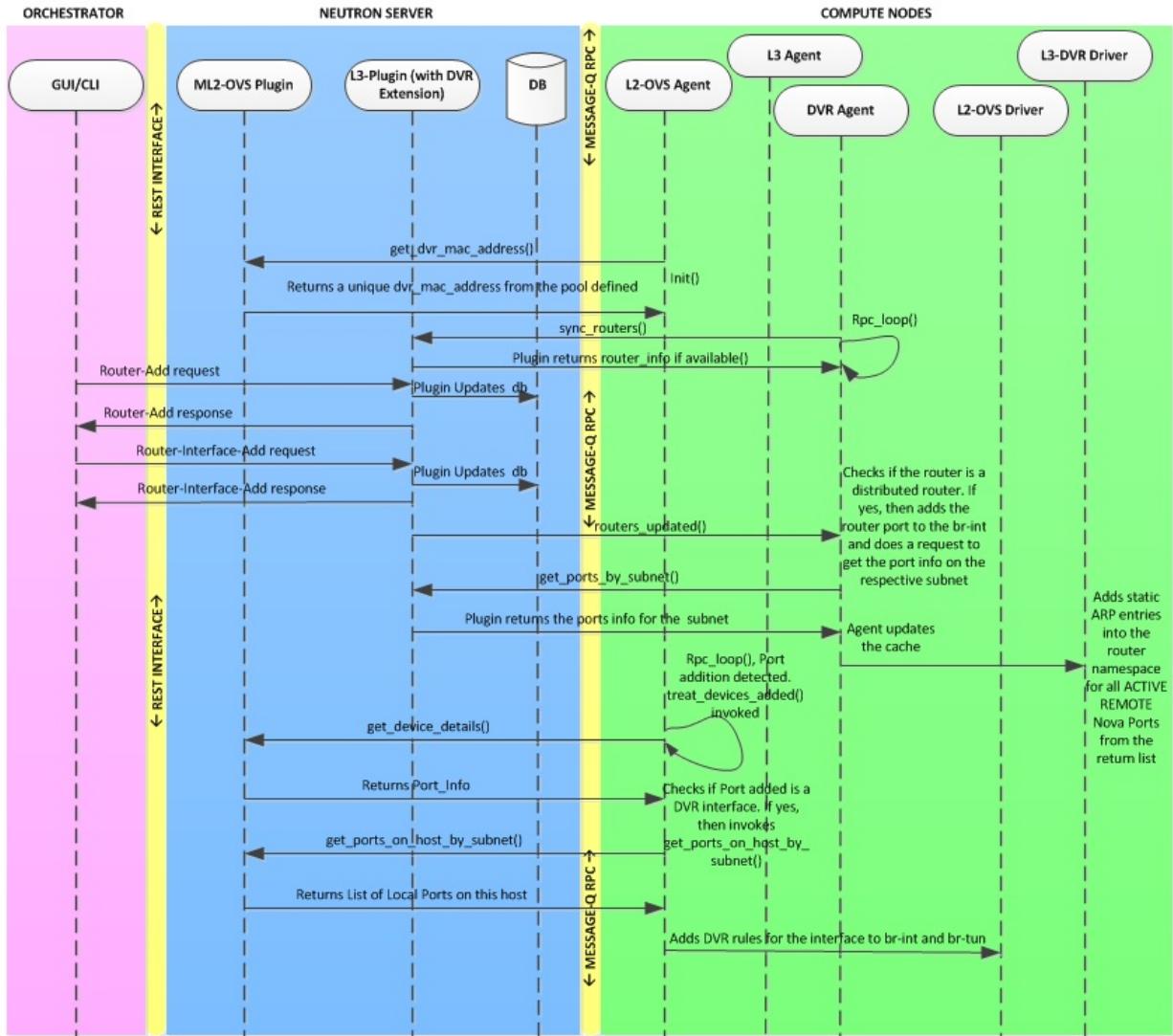
```
l2_population = True
```

```
tunnel_types = vxlan
```

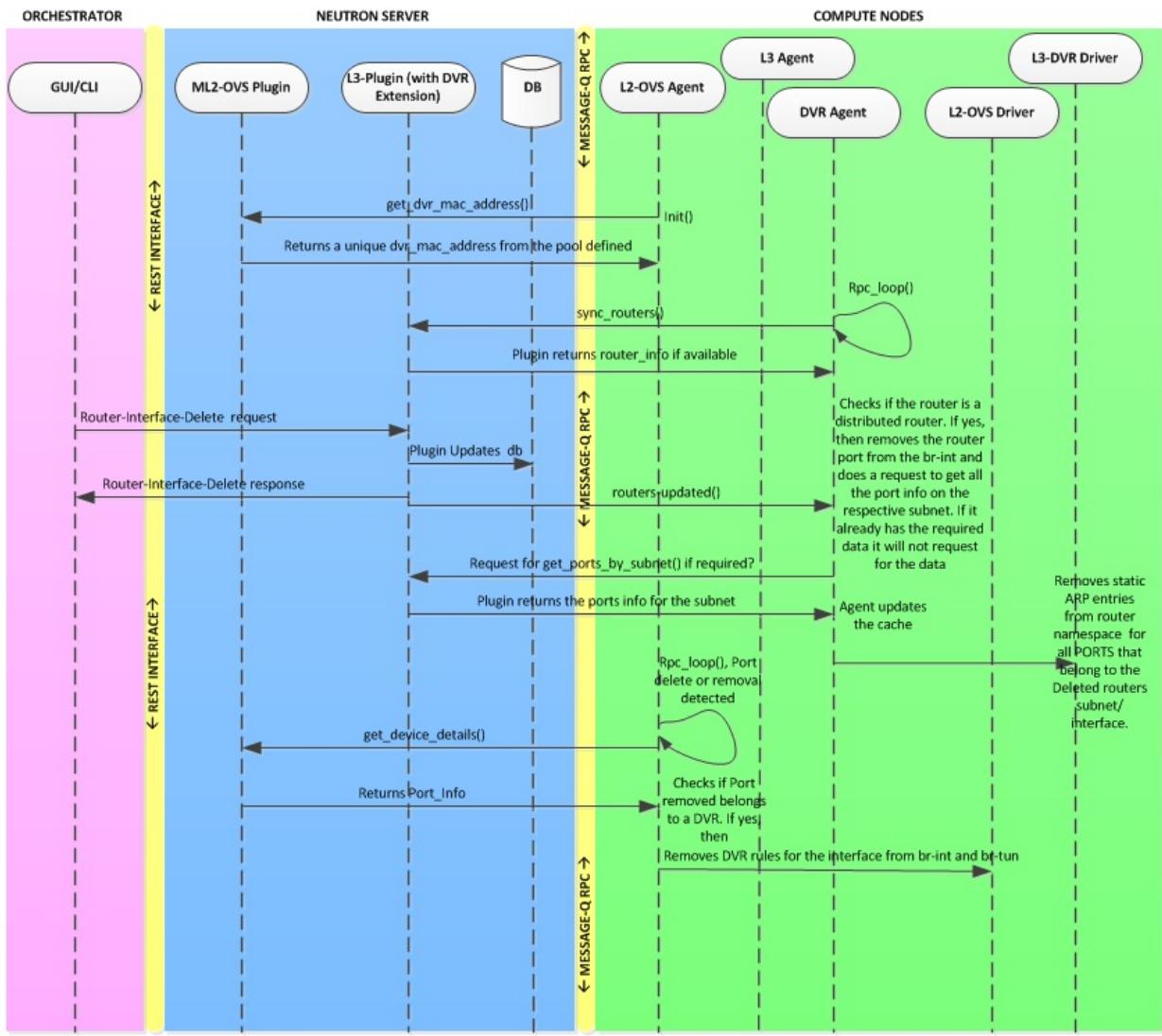
```
enable_distributed_routing = True
```

实现细节

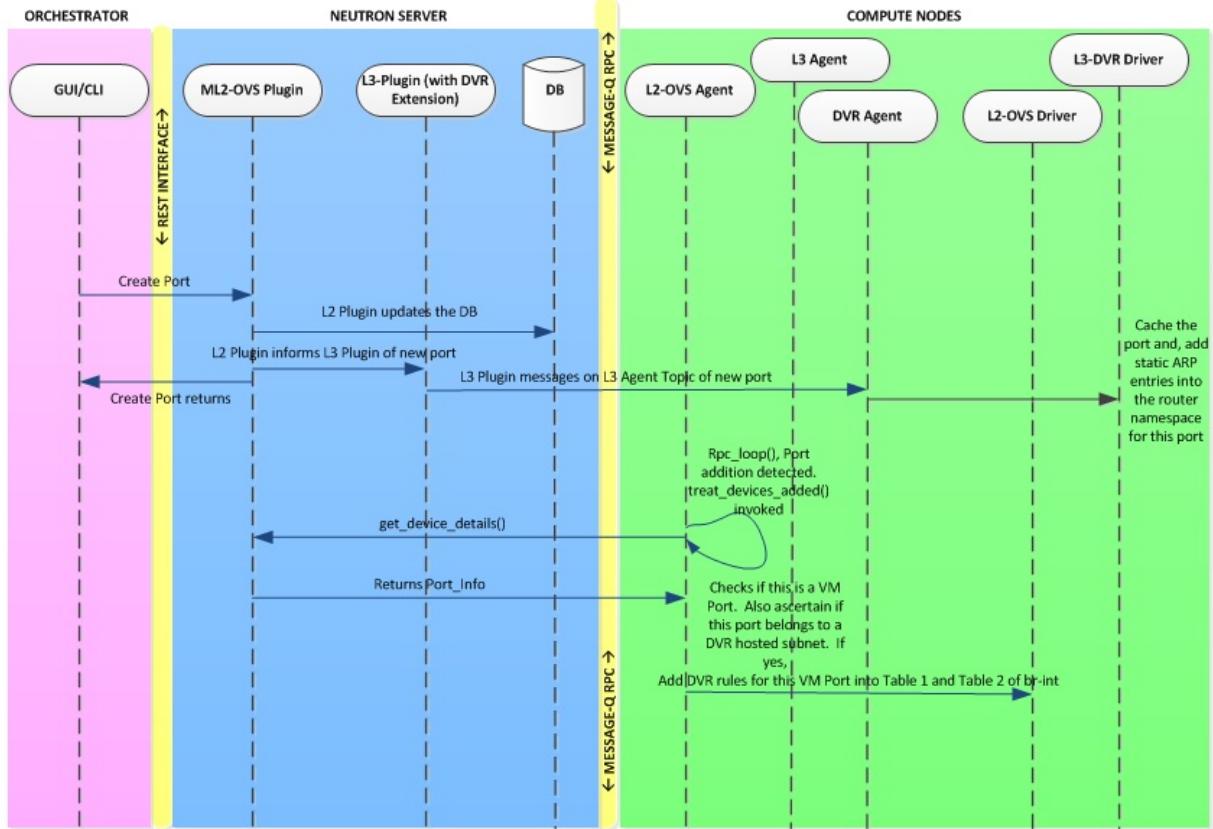
添加路由器接口



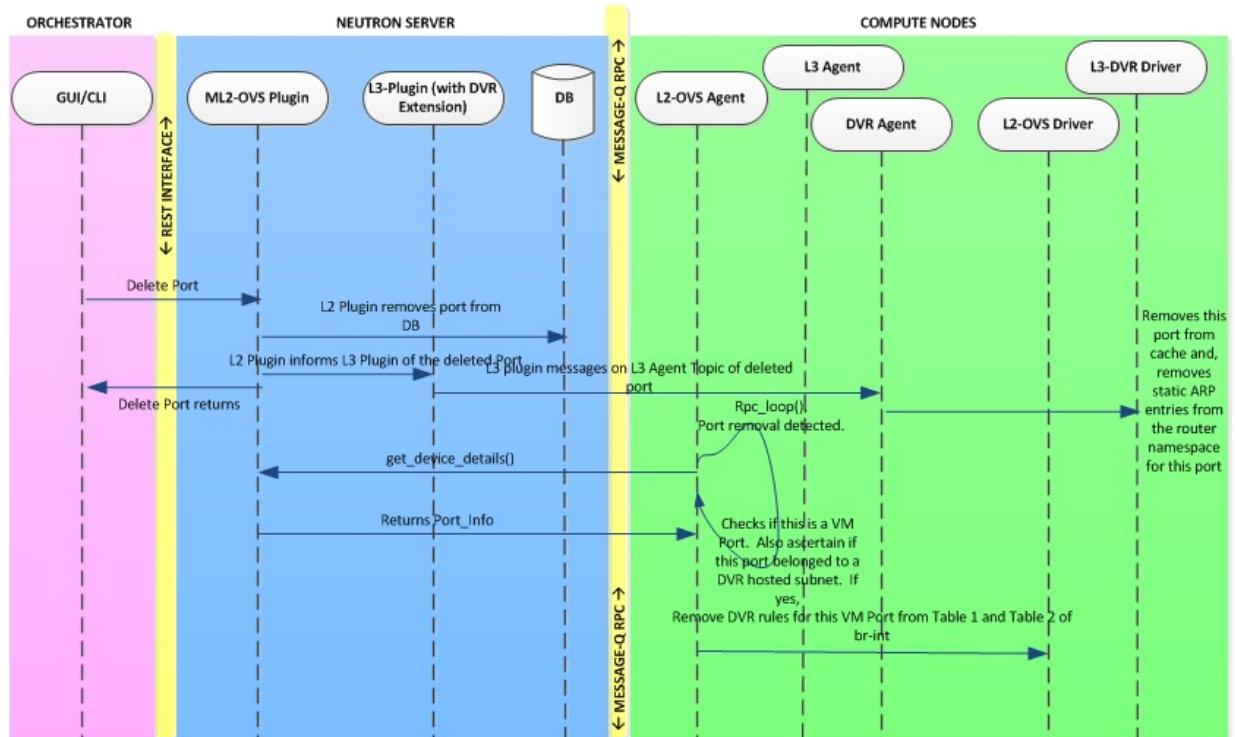
删除路由器接口



启动新的虚拟机节点



删除虚拟机节点



实现细节

以 OpenvSwitch plugin 为例，主要在

`neutron\plugins\openvswitch\agent\ovs_dvr_neutron_agent.py` 文件中。

`OVSDVRNeutronAgent` 类是本地处理的 agent，启动后会在三个网桥上添加初始化的规则，主要过程如下。

```
def setup_dvr_flows(self):
    self.setup_dvr_flows_on_integ_br()
    self.setup_dvr_flows_on_tun_br()
    self.setup_dvr_flows_on_phys_br()
    self.setup_dvr_mac_flows_on_all_brs()
```

其中，`br-int` 是本地交换网桥；`br-tun` 是跟其它计算节点通信的承载网桥；`br-phy` 是跟外部公共网络通信的网桥。

integration 网桥规则添加

`integrate` 网桥负责本地同一子网内的网包交换。

主要过程为

```

# Add a canary flow to int_br to track OVS restarts
    self.int_br.add_flow(table=constants.CANARY_TABLE, priority=0,
                           actions="drop")

        # Insert 'drop' action as the default for Table DVR_TO_SRC_MAC
        self.int_br.add_flow(table=constants.DVR_TO_SRC_MAC,
                           priority=1,
                           actions="drop")

        self.int_br.add_flow(table=constants.DVR_TO_SRC_MAC_VLAN
        ,
                           priority=1,
                           actions="drop")

        # Insert 'normal' action as the default for Table LOCAL_SWITCHING
        self.int_br.add_flow(table=constants.LOCAL_SWITCHING,
                           priority=1,
                           actions="normal")

    for physical_network in self.bridge_mappings:
        self.int_br.add_flow(table=constants.LOCAL_SWITCHING
        ,
                           priority=2,
                           in_port=self.int_ofports[physical_network],
                           actions="drop")

```

首先，网桥上采用了多个流表，定义在 `openvswitch/common` 目录下的 `constants.py`。主要流表包括：

- `LOCAL_SWITCHING` 表：0，顾名思义，用于本地的交换。
- `CANARY` 表：23
- `DVR_TO_SRC_MAC` 表：1
- `DVR_TO_SRC_MAC_VLAN` 表：2

上面的代码配置 LOCAL_SWITCHING 表默认规则为 NORMAL（作为本地的正常学习交换机），并不允许宿主机本地端口进来的包。配置其它表默认行为为丢弃。

tunnel 网桥规则添加

主要过程为

```
self.tun_br.add_flow(priority=1,
                      in_port=self.patch_int_ofport,
                      actions="resubmit(,%s)" %
                      constants.DVR_PROCESS)

# table-miss should be sent to learning table
self.tun_br.add_flow(table=constants.DVR_NOT_LEARN,
                     priority=0,
                     actions="resubmit(,%s)" %
                     constants.LEARN_FROM_TUN)

self.tun_br.add_flow(table=constants.DVR_PROCESS,
                     priority=0,
                     actions="resubmit(,%s)" %
                     constants.PATCH_LV_TO_TUN)
```

主要流表包括：

- DVR_PROCESS 表：1，顾名思义，用于 DVR 处理。
- DVR_NOT_LEARN 表：9。
- LEARN_FROM_TUN 表：10，学习从 tunnel 收到的网包。
- PATCH_LV_TO_TUN 表：2。

所有从 br-int 过来的网包，发到 DVR_PROCESS 表处理。

DVR_NOT_LEARN 表，默认将网包发给表 LEARN_FROM_TUN。

DVR_PROCESS 表，默认将网包发给表 PATCH_LV_TO_TUN。

physical 网桥规则添加

主要过程为

```

for physical_network in self.bridge_mappings:
    self.phys_brs[physical_network].add_flow(priority=2,
        in_port=self.phys_ofports[physical_network],
        actions="resubmit(,%s)" %
        constants.DVR_PROCESS_VLAN)
    self.phys_brs[physical_network].add_flow(priority=1,
        actions="resubmit(,%s)" %
        constants.DVR_NOT_LEARN_VLAN)
    self.phys_brs[physical_network].add_flow(
        table=constants.DVR_PROCESS_VLAN,
        priority=0,
        actions="resubmit(,%s)" %
        constants.LOCAL_VLAN_TRANSLATION)
    self.phys_brs[physical_network].add_flow(
        table=constants.LOCAL_VLAN_TRANSLATION,
        priority=2,
        in_port=self.phys_ofports[physical_network],
        actions="drop")
    self.phys_brs[physical_network].add_flow(
        table=constants.DVR_NOT_LEARN_VLAN,
        priority=1,
        actions="NORMAL")

```

对所有的连接到外部的物理网桥都添加类似规则。

主要流表包括：

- DVR_PROCESS_VLAN : 1
- LOCAL_VLAN_TRANSLATION : 2
- DVR_NOT_LEARN_VLAN : 3。

默认从 br-int 来的所有网包发送给表 DVR_PROCESS_VLAN。

DVR_PROCESS_VLAN 表，默认发给表 LOCAL_VLAN_TRANSLATION。

LOCAL_VLAN_TRANSLATION 表丢弃从 br-int 来的所有网包。

其他所有网包发送给表 DVR_NOT_LEARN_VLAN。

表 DVR_NOT_LEARN_VLAN 默认规则为 NORMAL。

所有网桥上添加 mac 规则

主要过程为

```

for physical_network in self.bridge_mappings:
    self.int_br.add_flow(table=constants.LOCAL_SWITCHING,
        priority=4,
        in_port=self.int_ofports[physical_network],
        dl_src=mac['mac_address'],
        actions="resubmit(,%s)" %
            constants.DVR_TO_SRC_MAC_VLAN)
    self.phys_brs[physical_network].add_flow(
        table=constants.DVR_NOT_LEARN_VLAN,
        priority=2,
        dl_src=mac['mac_address'],
        actions="output:%s" %
            self.phys_ofports[physical_network])

if self.enable_tunneling:
    # Table 0 (default) will now sort DVR traffic from other
    # traffic depending on in_port
    self.int_br.add_flow(table=constants.LOCAL_SWITCHING,
        priority=2,
        in_port=self.patch_tun_ofport,
        dl_src=mac['mac_address'],
        actions="resubmit(,%s)" %
            constants.DVR_TO_SRC_MAC)

    # Table DVR_NOT_LEARN ensures unique dvr macs in the cloud
    # are not learnt, as they may
    # result in flow explosions
    self.tun_br.add_flow(table=constants.DVR_NOT_LEARN,
        priority=1,
        dl_src=mac['mac_address'],
        actions="output:%s" %
            self.patch_int_ofport)
    self.registered_dvr_macs.add(mac['mac_address'])

```

这里主要是添加对其他主机上的 DVR 网关发过来的网包的处理。

外部网络为 **vlan** 的情况下

br-int 网桥上，表 LOCAL_SWITCHING 添加优先级为 4 的流，如果从外面过来的网包，MAC 源地址是其他的 DVR 网关地址，则扔给表 DVR_TO_SRC_MAC_VLAN 处理。

br-eth 网桥上，表 DVR_NOT_LEARN_VLAN 则添加优先级为 2 的规则，MAC 源地址是其他的 DVR 网关地址，则发给 br-int。

外部网络为 **tunnel** 的情况下

类似，分别在 br-int 网桥的 LOCAL_SWITCHING 表和 br-tun 网桥的 DVR_NOT_LEARN 表添加流，优先级分别改为 2 和 1。

工具

由于 OpenStack 自身的各个组件都是松耦合关系，常见的部署也都是分布式部署，造成要深入体会 Neutron 的工作过程，或者进行故障诊断，往往涉及到多个组件，是否繁琐。

本章介绍一些工具，提高操作和开发的效率。

easyOVS

easyOVS 是一个开源的 OpenvSwitch 虚拟交换机管理工具。使用它，用户可以很轻松的对 OpenvSwitch 的网桥、端口等进行查看，同时它深度整合了 OpenStack（支持 Havana 版本到 Juno 版本）中网络相关的信息，也是一个十分强大的 Neutron 中各个组件的监测工具。

主要功能一览

- 支持 OpenvSwitch 版本 1.4.6 ~ 2.0.2，OpenStack Havana 到 Juno 版本；
- 支持操作系统环境报 Ubuntu、Debian、CentOS、Fedora 和 Redhat；
- 输出结果经过处理，支持彩色输出，十分简洁易读；
- 开启 OpenStack 支持，可以获取端口的地址、mac、vlan 甚至虚拟机关联的 iptables 规则等信息；
- 对流表操作语法更加简洁，并支持通过 id 进行删除；
- 支持 tab 自动补全；
- 支持通过 `-m 'cmd'` 来直接运行命令，无需进入 CLI 操作。

安装

安装十分简单，一行代码搞定。

```
git clone https://github.com/yeasy/easyOVS.git && sudo bash ./easyOVS/util/install.sh
```

安装成功后，可以使用

```
sudo easyovs
```

进入操作界面。

打开 OpenStack 支持

由于 OpenStack 组件信息获取需要有相关的认证信息，因此需要在环境变量或者配置文件中进行指定。

环境变量

可以在用户目录的 `.bashrc` 文件中加入

```
export OS_USERNAME=demo  
export OS_TENANT_NAME=demo  
export OS_PASSWORD=admin  
export OS_AUTH_URL=http://127.0.0.1:5000/v2.0/
```

配置文件

默认的配置文件在 `/etc/easyovs.conf`，替换为相应的认证信息即可。

```
[OS]  
auth_url = http://127.0.0.1:5000/v2.0  
username = demo  
password = admin  
tenant_name = demo
```

操作命令

help

显示帮助信息。

list

列出本地的 OpenvSwitch 网桥，例如

```
EasyOVS> list
s1
Port: s1-eth2 s1 s1-eth1
Interface: s1-eth2 s1 s1-eth1
Controller:ptcp:6634 tcp:127.0.0.1:6633
Fail_Mode: secure

s2
Port: s2 s2-eth3 s2-eth2 s2-eth1
Interface: s2 s2-eth3 s2-eth2 s2-eth1
Controller:tcp:127.0.0.1:6633 ptcp:6635
Fail_Mode: secure

s3
Port: s3-eth1 s3-eth3 s3-eth2 s3
Interface: s3-eth1 s3-eth3 s3-eth2 s3
Controller:ptcp:6636 tcp:127.0.0.1:6633
Fail_Mode: secure
```

show

```
EasyOVS> [bridge|default] show
```

显示某个网桥上的端口信息，例如

```
EasyOVS> br-int show
br-int
Intf          Port      Vlan   Type      vmiP
vmMAC
int-br-eth0    15
qvo260209fa-72 11      1       192.168.0.4
               fa:16:3e:0f:17:04
qvo583c7038-d3 2       1       192.168.0.2
               fa:16:3e:9c:dc:3a
qvo8bf9cba2-3f 9       1       192.168.0.5
               fa:16:3e:a2:2f:0e
qvod4de9fe0-6d 8       2       10.0.0.2
               fa:16:3e:38:2b:2e
br-int         LOCAL     internal
```

dump

```
EasyOVS> [bridge|default] dump
```

显示网桥上绑定的流表规则，例如

```
EasyOVS> s1 dump
ID TAB PKT      PRI      MATCH
                  ACT
0 0 0           2400    dl_dst=ff:ff:ff:ff:ff:ff
                  CONTROLLER:65535
1 0 0           2400    arp
                  CONTROLLER:65535
2 0 0           2400    dl_type=0x88cc
                  CONTROLLER:65535
3 0 0           2400    ip, nw_proto=2
                  CONTROLLER:65535
4 0 0           801     ip
                  CONTROLLER:65535
5 0 2           800
```

addflow

```
EasyOVS> [bridge|default] addflow [match] actions=[action]
```

添加一条流到网桥，例如

```
EasyOVS> br-int addflow priority=3 ip actions=OUTPUT:1
```

delflow

```
EasyOVS> [bridge|default] delflow id1 id2...
```

从网桥删除流，其中 `id` 信息可以从 `dump` 的结果中拿到。

set

```
EasyOVS> bridge set
```

指定默认网桥，同时进入网桥操作模式，指定后进行操作可以忽略网桥信息。

```
EasyOVS> set br-int
Set the default bridge to br-int.
EasyOVS: br-int>
```

exit

```
EasyOVS> exit
```

退出网桥模式，或者退出程序。

get

```
EasyOVS> get
```

在网桥模式下，获取当前的网桥名称。

```
EasyOVS: br-int> get
Current default bridge is br-int
```

ipt

```
EasyOVS> ipt vm_ip1, vm_ip2...
```

给定虚拟机 IP 地址，显示与它相关的 `iptables` 规则。需要启用 OpenStack 支持。

```
EasyOVS> ipt 192.168.0.2 192.168.0.4
## IP = 192.168.0.2, port = qvo583c7038-d ##
      PKTS      SOURCE          DESTINATION      PROT      OTHER
#IN:
      672      all            all            all      state RELATED,
ESTABLISHED
      0      all            all            tcp      tcp dpt:22
      0      all            all            icmp
      0      192.168.0.4    all            all
      3      192.168.0.5    all            all
      8      10.0.0.2       all            all
  85784      192.168.0.3   all            udp      udp spt:67 dpt
:68
#OUT:
```

```

      196K    all          all          udp    udp spt:68 dpt
:67
      86155    all          all          all    state RELATED,
ESTABLISHED
      1241    all          all          all
#SRC_FILTER:
      59163    192.168.0.2    all          all    MAC FA:16:3E:9
C:DC:3A
## IP = 192.168.0.4, port = qvo260209fa-7 ##
      PKTS    SOURCE        DESTINATION   PROT  OTHER
#IN:
      73    all          all          all    state RELATED,
ESTABLISHED
      0    all          all          tcp    tcp dpt:22
      0    all          all          icmp
      0    192.168.0.2    all          all
      0    192.168.0.5    all          all
      0    10.0.0.2      all          all
      11331   192.168.0.3    all          udp    udp spt:67 dpt
:68
#OUT:
      30034   all          all          udp    udp spt:68 dpt
:67
      11377   all          all          all    state RELATED,
ESTABLISHED
      12    all          all          all
#SRC_FILTER:
      9859    192.168.0.4    all          all    MAC FA:16:3E:0
F:17:04

```

query

```
EasyOVS> query port_ip, port_id...
```

给定某个的端口的 IP 地址，或者部分端口 id 信息，显示该端口相关的完整信息。
需要启用 OpenStack 支持。

```
EasyOVS> query 10.0.0.2,c4493802
## port_id = f47c62b0-dbd2-4faa-9cdd-8abc886ce08f
status: ACTIVE
name:
allowed_address_pairs: []
admin_state_up: True
network_id: ea3928dc-b1fd-4a1a-940e-82b8c55214e6
tenant_id: 3a55e7b5f5504649a2dfde7147383d02
extra_dhcp_opts: []
binding:vnic_type: normal
device_owner: compute:az_compute
mac_address: fa:16:3e:52:7a:f2
fixed_ips: [{u'subnet_id': u'94bf94c0-6568-4520-aee3-d12b5e47212
8', u'ip_address': u'10.0.0.2'}]
id: f47c62b0-dbd2-4faa-9cdd-8abc886ce08f
security_groups: [u'7c2b801b-4590-4a1f-9837-1cce7f6d1d0']
device_id: c3522974-8a08-481c-87b5-fe3822f5c89c
## port_id = c4493802-4344-42bd-87a6-1b783f88609a
status: ACTIVE
name:
allowed_address_pairs: []
admin_state_up: True
network_id: ea3928dc-b1fd-4a1a-940e-82b8c55214e6
tenant_id: 3a55e7b5f5504649a2dfde7147383d02
extra_dhcp_opts: []
binding:vnic_type: normal
device_owner: compute:az_compute
mac_address: fa:16:3e:94:84:90
fixed_ips: [{u'subnet_id': u'94bf94c0-6568-4520-aee3-d12b5e47212
8', u'ip_address': u'10.0.0.4'}]
id: c4493802-4344-42bd-87a6-1b783f88609a
security_groups: [u'7c2b801b-4590-4a1f-9837-1cce7f6d1d0']
device_id: 9365c842-9228-44a6-88ad-33d7389cda5f
```

sh

```
EasyOVS> sh cmd
```

执行系统命令。

```
EasyOVS> sh ls -l
total 48
drwxr-xr-x. 2 root root 4096 Apr  1 14:34 bin
drwxr-xr-x. 5 root root 4096 Apr  1 14:56 build
drwxr-xr-x. 2 root root 4096 Apr  1 14:56 dist
drwxr-xr-x. 2 root root 4096 Apr  1 14:09 doc
drwxr-xr-x. 4 root root 4096 Apr  1 14:56 easyovs
-rw-r--r--. 1 root root  660 Apr  1 14:56 easyovs.1
drwxr-xr-x. 2 root root 4096 Apr  1 14:56 easyovs.egg-info
-rw-r--r--. 1 root root 2214 Apr  1 14:53 INSTALL
-rw-r--r--. 1 root root 1194 Apr  1 14:53 Makefile
-rw-r--r--. 1 root root 3836 Apr  1 14:53 README.md
-rw-r--r--. 1 root root 1177 Apr  1 14:53 setup.py
drwxr-xr-x. 2 root root 4096 Apr  1 14:09 util
```

quit

输入 `^d` 或者 `quit` 命令来退出程序。

参数

-h

显示帮助信息。

```
$ easyovs -h
Usage: easyovs [options]
(type easyovs -h for details)
```

The easyovs utility creates operation CLI from the command line.
It can run
given commands, invoke the EasyOVS CLI, and run tests.

Options:

-h, --help	show this help message and exit
-c, --clean	clean and exit
-m CMD, --cmd=CMD	Run customized commands for tests.
-v VERBOSITY, --verbosity=VERBOSITY	info warning critical error debug output
--version	

-C

进行环境清理。

-m

不进入 CLI，直接执行给定的命令，显示结果。例如

```
$ sudo easyovs -m "show br-int"
Intf          Port      Vlan     Type      vmIP
vmMAC
qvo47c62b0-db    2          1          10.0.0.2
fa:16:3e:52:7a:f2
qvoc4493802-43    3          1          10.0.0.4
fa:16:3e:94:84:90
br-int          LOCAL      internal
patch-tun        6          patch
```

例如

```
$ sudo easyovs -m 'br-int dump'
ID TAB PKT      PRI    MATCH
          ACT
0   0   30       1      *
                  NORMAL
1   23  0        0      *
                  drop
```

-v

设置输出信息的日志级别，包括 debug , info , warn , error 等，方便进行调试。

--version

显示版本信息。

参考

- http://openstack.redhat.com/Networking_in_too_much_detail
- <http://web.archive.org/web/20150215214007/http://masimum.inf.um.es/firm/2013/12/26/the-journey-of-a-packet-within-an-openstack-cloud>
- <http://packetpushers.net/openstack-quantum-network-implementation-in-linux/>
- <http://blog.scottlowe.org/2013/09/04/introducing-linux-network-namespaces/>
- <http://assafmuller.wordpress.com/2013/10/14/gre-tunnels-in-openstack-neutron/>
- <http://lwn.net/Articles/580893/>

附：安装配置

参考 <https://github.com/yeasy/openstack-tool>。

DevStack 安装 OpenStack 多节点 (Juno+Neutron+ML2+VXLAN)

目前安装 OpenStack 常见的方案有 Redhat 的 RDO 和社区的 DevStack。

当然，也可以手动安装，可以参考：github.com/ChaimaGhribi/OpenStack-Juno-Installation/blob/master/OpenStack-Juno-Installation.rst

其中，RDO 功能比较强大，运行也稳定，可以在一个节点上通过一个 answer 文件直接部署多个节点，搭建一套 OpenStack 环境。但是可惜，在 Ubuntu 上还不支持。

DevStack 支持 Ubuntu、Fedora 等环境，需要在每个节点上单独执行，适合进行实验。目前常见的教程一般都是讲解 DevStack 单节点安装。本文讲解最新的 Juno 版本在多节点上的安装过程。

网络环境

两台机器，分为控制节点（同时也作为网络节点）和计算节点。

控制节点

eth0: 9.186.100.77/24 作为管理网络（同时也是公共网络）。 eth1:
10.0.100.77/24 作为内部网络接口。

计算节点

eth0: 9.186.100.88/24 作为管理网络（同时也是公共网络）。 eth1:
10.0.100.88/24 作为内部网络接口。

配置 **stack** 用户

创建 **stack** 用户

```
sudo groupadd stack  
sudo useradd -g stack -s /bin/bash -d /opt/stack -m stack
```

添加 **stack** 用户权限。

```
sudo echo "stack ALL=(ALL) NOPASSWD: ALL" >> /etc/sudoers
```

切换到 **stack** 用户

```
sudo su - stack
```

下载代码

下载 **devstack** 代码，并切换到 **stable/juno** 分支。

```
sudo apt-get install git -y  
git clone https://git.openstack.org/openstack-dev/devstack -b stable/juno
```

编写运行配置文件

在 **devstack** 根目录下，编写 **local.conf**。

控制节点的 **local.conf**

```
[[local|localrc]]  
  
MULTI_HOST=true  
HOST_IP=9.186.100.77 # management network  
  
FIXED_RANGE=10.0.0.0/24 #tenant private network  
NETWORK_GATEWAY=10.0.0.1  
#FIXED_NETWORK_SIZE=4096  
  
PUBLIC_INTERFACE=eth0 #public network  
FLOATING_RANGE=9.186.100.128/25  
PUBLIC_NETWORK_GATEWAY=9.186.100.1  
  
TUNNEL_ENDPOINT_IP=10.0.100.77 # data network endpoint for vxlan  
  
LOGFILE=/opt/stack/logs/stack.sh.log  
  
# Credentials  
ADMIN_PASSWORD=admin  
MYSQL_PASSWORD=secret  
RABBIT_PASSWORD=secret  
SERVICE_PASSWORD=secret  
SERVICE_TOKEN=abcdefghijklmnopqrstuvwxyz  
  
# enable neutron-ml2-vxlan  
disable_service n-net  
enable_service q-svc,q-agt,q-dhcp,q-l3,q-meta,q-metering,q-lbaas  
,q-fwaas,q-vpn,neutron,tempест,heat  
  
#OFFLINE=True  
  
LOG_COLOR=False  
LOGDIR=$DEST/logs  
SCREEN_LOGDIR=$LOGDIR/screen
```

计算节点的 local.conf

```
[[local|localrc]]  
MULTI_HOST=true
```

```
HOST_IP=9.186.100.88 # management network

FIXED_RANGE=10.0.0.0/24 # tenant private network
NETWORK_GATEWAY=10.0.0.1
#FIXED_NETWORK_SIZE=4096

TUNNEL_ENDPOINT_IP=10.0.100.88 # data network endpoint for vxlan

# Credentials
ADMIN_PASSWORD=admin
MYSQL_PASSWORD=secret
RABBIT_PASSWORD=secret
SERVICE_PASSWORD=secret
SERVICE_TOKEN=abcdefghijklmnopqrstuvwxyz

# Service information
SERVICE_HOST=9.186.100.77
MYSQL_HOST=$SERVICE_HOST
RABBIT_HOST=$SERVICE_HOST
GLANCE_HOSTPORT=$SERVICE_HOST:9292
Q_HOST=$SERVICE_HOST
KEYSTONE_AUTH_HOST=$SERVICE_HOST
KEYSTONE_SERVICE_HOST=$SERVICE_HOST

CEILOMETER_BACKEND=mongodb
DATABASE_TYPE=mysql

ENABLED_SERVICES=n-cpu, q-agt, neutron

# vnc config
NOVA_VNC_ENABLED=True
NOVNCPROXY_URL="http://$SERVICE_HOST:6080/vnc_auto.html"
VNCSERVER_LISTEN=$HOST_IP
VNCSERVER_PROXYCLIENT_ADDRESS=$VNCSERVER_LISTEN

#OFFLINE=True

LOG_COLOR=False
LOGDIR=$DEST/logs
```

```
SCREEN_LOGDIR=$LOGDIR/screen  
#LOGFILE=/opt/stack/logs/stack.sh.log
```

执行配置

执行命令。

```
./stack.sh
```

会输出各项操作的结果。日志会写到 `stack.sh.log` 文件。成功后最后会打印出相关信息。

控制节点上

```
Horizon is now available at http://9.186.100.77/  
Keystone is serving at http://9.186.100.77:5000/v2.0/  
Examples on using novaclient command line is in exercise.sh  
The default users are: admin and demo  
The password: admin  
This is your host ip: 9.186.100.77
```

执行成功后，在控制节点上，将物理网卡 `eth0` 绑定到 `br-ex` 网桥上。

```
sudo ifconfig eth0 0.0.0.0; sudo ovs-vsctl add-port br-ex eth0;  
sudo ifconfig br-ex 9.186.100.77/24; sudo route add default gw 9  
.186.100.1
```

其它事项

停止服务

停止 OpenStack 的各个服务（DevStack 默认每个服务都在一个 `screen` 中以进程方式运行，可以通过 `screen -x stack` 进入查看）。

```
./unstack.sh
```

清除安装。

```
./clean.sh
```

手动清除

有时候有些文件可能清除不干净，手动执行

```
sudo rm -rf /etc/libvirt/qemu/inst*
sudo virsh list | grep inst | awk '{print $1}' | xargs -n1 virsh
destroy
```

依赖版本

安装过程中可能会报某些包版本不满足的问题，手动安装后。

例如报 `six` 包版本过低。

```
sudo pip install six --upgrade
```

screen 操作

`ctrl+a+o` 显示 session 列表。`ctrl+a+d` 挂起到后台。