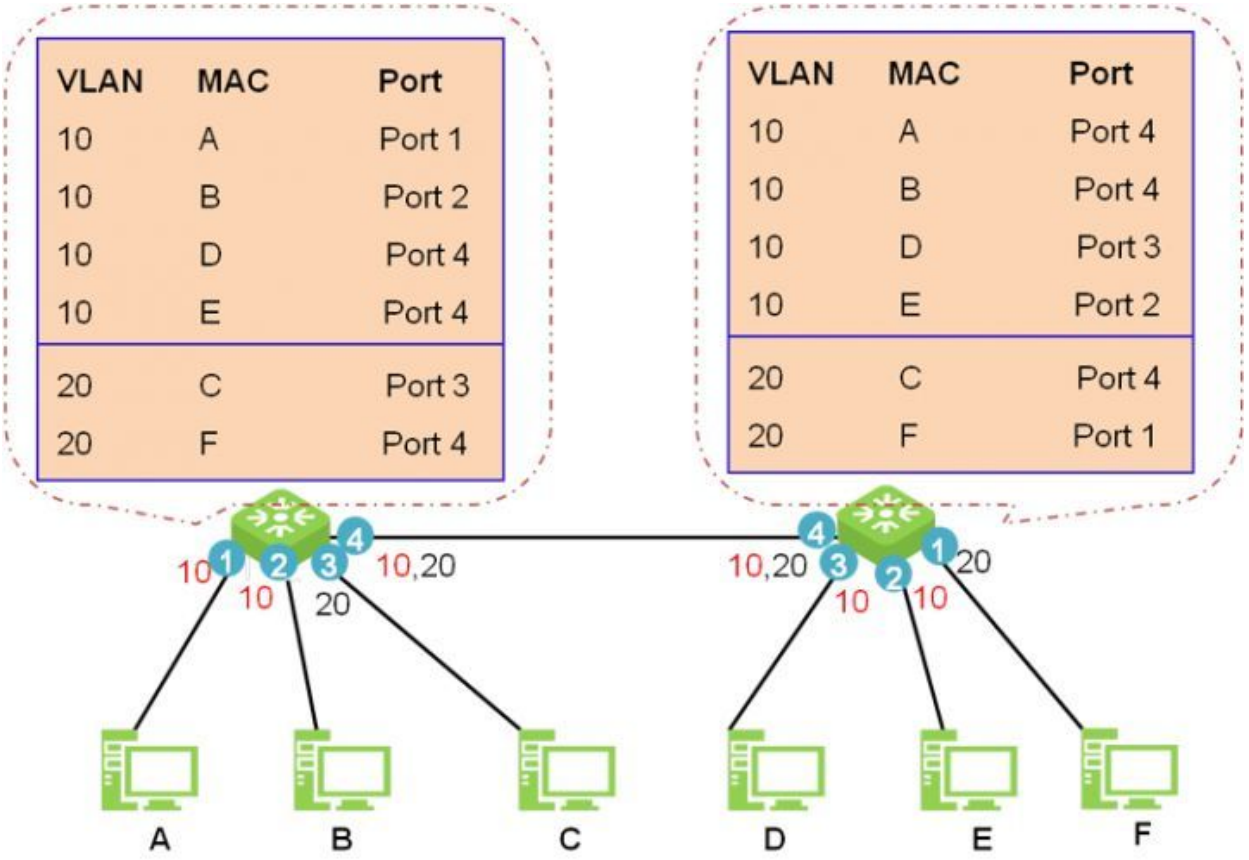
VXLAN in OpenStack Neutron

传统二层网络工作方式

—

传统二层网络通过交换机内的MAC地址表实现转发。如下图所示。



比如A要发送数据给E。因为A与左边的交换机直连， A先将以太网数据帧发给左边的交换机。左边的交换机收到数据帧之后，查找自身的MAC地址表，从自己的端口4发出，发到了右边的交换机。右边的交换机收到数据帧之后，也是通过查找自身的MAC地址表，从自己的端口2发出。因为端口2与E直连，所以以太网数据帧发到了E。

可以看出MAC地址表是交换机完成转发的核心。这里的MAC地址表是一种控制层信息，因为它决定了网络数据帧的转发。传统网络里面，MAC地址表是通过数据层学习获得的。所有经过交换机的以太网数据帧，交换机会读取源MAC地址，并且记录这个数据帧来自哪个交换机端口。这样交换机就知道MAC地址和

交换机端口的对应关系，下次要转发的时候可以根据MAC地址找到对应的交换机端口，进而完成转发。这些对应关系就是交换机的MAC地址表。

既然MAC地址表是学习生成的，那它就不能保证掌握了所有的MAC地址信息。可能还没学到，可能学到了又老化删除了。对于MAC地址表里没有的信息，交换机没办法按照常规的方式完成转发。这个时候，交换机会将数据帧发往所有相关的端口（Trunk口和同一个VLAN下的所有Access端口），这样总能发到目的主机。如果目的主机返回了数据，还是会通过交换机来做转发。交换机收到这个返回的数据帧，就能学习到缺失的信息。这个过程就是常说的flood-learn，发送到所有相关的端口就是flood，从返回的数据学习就是learn。通过flood-learn，交换机学习到缺失的信息，下一次转发同一个目的MAC地址的数据帧时，就不需要再flood。

可以看出，传统的二层网络，并不需要一个独立的“控制层”，只是通过数据层的学习，就能获取相应的转发信息（MAC地址表）。这样的设计，设备相对独立，真正做到插上线就走。

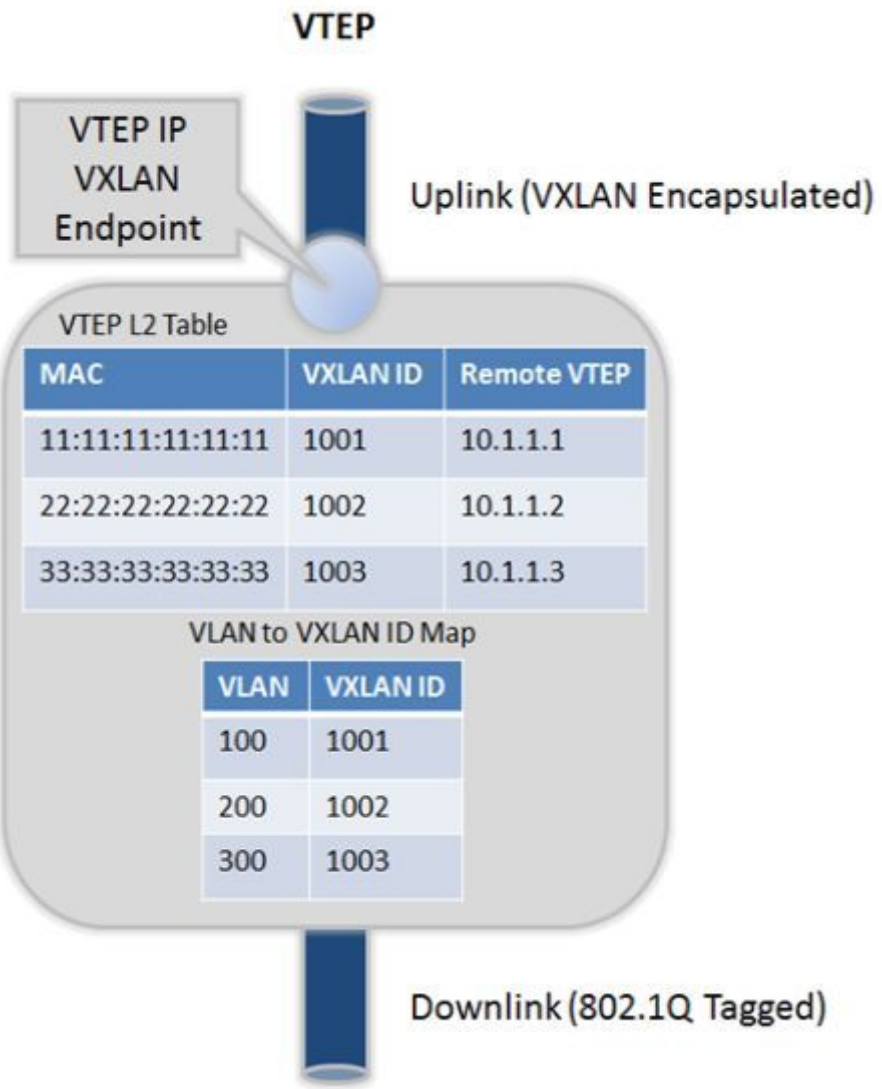
传统VXLAN网络的工作方式

—

所谓的传统的VXLAN网络，是指RFC7348[1]定义的VXLAN的工作方式。

VXLAN将以太网数据帧封装在UDP里面，进而在三层网络传输。VXLAN数据的封装和解封装发生在VTEP（VXLAN Tunnel EndPoint）。VXLAN可以看成是建立在VTEP之间的L2VPN。

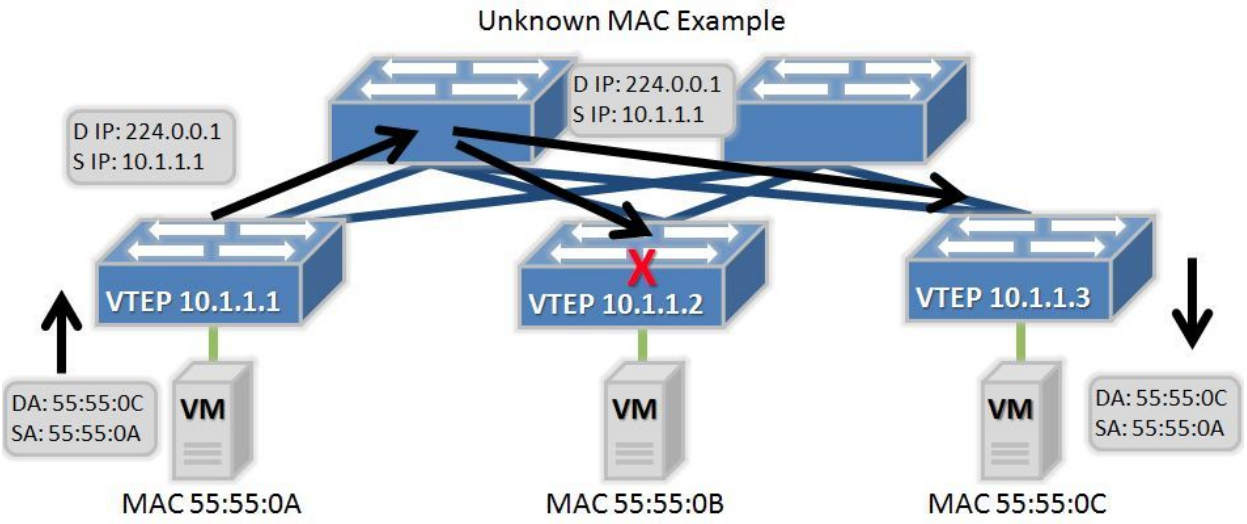
VTEP与传统交换机类似，也是基于MAC地址表工作。VTEP的示意图如下，在VTEP里面，可以认为存在两个表：一个是VLAN和VXLAN的对应关系表；另一个是MAC地址表，里面包含了MAC 地址，VXLAN ID和远端VTEP IP地址的对应关系。VTEP向下连接多个主机，为了区分多个租户网络，不同的租户网络会用VLAN Tag做区分。VTEP收到下连主机的网络数据帧时，会先根据VLAN，查第一个表获得对应的VXLAN ID，之后根据VXLAN ID和目的MAC地址，查MAC地址表获取远端VTEP的IP地址。最后，VTEP会剥离VLAN Tag，按照VXLAN格式封装数据帧，发往远端的VTEP。



VTEP里VLAN和VXLAN的对应关系表是固定的。MAC地址表是VTEP的核心。这里的MAC地址表也是一种控制层信息，因为它决定了网络数据帧的转发。与传统的二层网络类似，传统的VXLAN网络也是通过数据层的学习来更新MAC地址表。VTEP收到的所有的VXLAN数据，VTEP会记录内层报文的源MAC地址，VXLAN ID和远端VTEP的IP地址，进而更新自己的MAC地址表。在这里，与传统的二层网络相比，VLAN ID变成了VXLAN ID，交换机端口变成了远端VTEP IP地址，不过最核心的东西没有变。

既然与传统二层网络类似，那也同样面临MAC地址表信息不全的问题。VTEP如果不能在自己的MAC地址表里面找到对应目的MAC地址的记录，也会有个flood-learn的过程。VTEP会将数据帧封装成VXLAN数据，发送给所有相关的远端VTEP，这样总能发到目的主机。一旦远端VTEP返回数据，那当前VTEP就能学习到缺失的MAC地址信息。

这里有个新的问题，传统二层网络里面，可以根据VLAN来识别flood的范围。可是在VXLAN网络里面，根据什么来识别flood的范围？VXLAN ID是不行的，这个只在VTEP能识别，出了VTEP就没人认识了。传统的VXLAN网络借助了IP组播来识别flood的范围。对于每一个VXLAN ID对应的网络，所有关联的VTEP都预先配置在一个组播组里面。VTEP需要flood的时候，只需要将VXLAN数据的外层IP地址设置成组播地址，这样，所有关联的VTEP都能收到flood。具体的过程如下图所示，224.0.0.1就是一个组播地址。



可以看出，传统的VXLAN网络与传统的二层网络还是很像的。最主要的是，传统的VXLAN网络，也不需要一个特定的“控制层”，只通过数据层的学习，就能获取相应的转发信息（MAC地址表）。这样的设计，使得VXLAN不必依赖某个特定的控制层，兼容性更好，便于早期的协议推广。

OpenStack Neutron

—

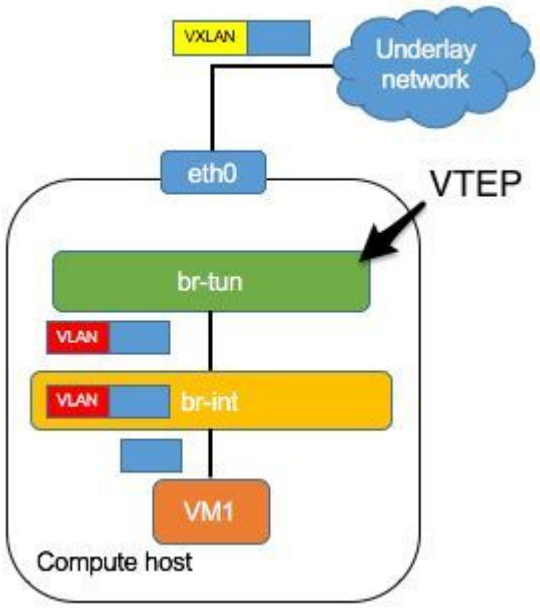
OpenStack Neutron作为OpenStack的网络项目，负责在整个OpenStack环境里面编排和管理虚拟网络环境。OpenStack Neutron支持多种底层，其中应用最广的是基于OpenVSwitch的实现。基于OpenVSwitch的Neutron，支持VXLAN网络，但是在实现上与传统的VXLAN网络有点不一样。

基于软件和OpenFlow的VTEP

—

VTEP是VXLAN网络的核心，Neutron通过软件（OpenVSwitch）和OpenFlow实现VTEP。下图是OpenStack在计算节点上的虚拟交换机的连接示意图。br-int

和br-tun都是OpenVSwitch虚拟交换机。



br-int负责连接虚拟机，并且给不同租户的网络数据打上不同的VLAN Tag，从而实现同一个宿主机上不同租户网络之间的隔离。带VLAN Tag的数据帧被送到br-tun。VTEP在br-tun上实现。前面说过VTEP里面有两张表，一张是VLAN和VXLAN的对应关系表，一张是MAC地址表。在OpenStack Neutron，这两张表通过br-tun上的OpenFlow流表实现。对于VLAN和VXLAN的对应关系，在入方向上是在br-tun的table4实现的。

```
$ sudo ovs-ofctl dump-flows br-tun table=4
table=4, priority=1,tun_id=0x3 actions=mod_vlan_vid:4,resubmit(,9)
table=4, priority=1,tun_id=0x1c actions=mod_vlan_vid:5,resubmit(,9)
```

这里直接将VXLAN ID 3与本地VLAN 4对应，VXLAN ID 28（0x1C）与本地VLAN 5对应。

MAC地址表，默认情况下，仍然是通过数据层的学习，来进行更新。学习发生在table10。

```
$ sudo ovs-ofctl dump-flows br-tun table=10
table=10, priority=1
actions=learn(table=20,hard_timeout=300,priority=1,cookie=0xbdfa8adb40ce98c5,NXM_
>NXM_OF_VLAN_TCI[],load:NXM_NX_TUN_ID[]->NXM_NX_TUN_ID[],output:OXM_OF_IN_PORT[])
```

这条流表乍一看很复杂，但是如果理解了传统网络里面的自学习过程，理解它并不难。br-tun作为VTEP，收到VXLAN数据，解析之后，下一步肯定是要发给br-int，进而送到虚拟机。这条流表在数据帧发往br-int之前（output:”patch-int”），先在table 20里面学习生成一条OpenFlow流表（learn(table=20）。这条流表有

固定的hard_time，priority，cookie，并且匹配的VLAN ID来自于当前被学习的数据帧（NXM_OF_VLAN_TCI[0..11]），匹配的目的MAC地址来自于当前被学习的数据帧的源MAC地址（NXM_OF_ETH_DST[]=NXM_OF_ETH_SRC[]）。匹配完之后，对应的动作有：剥离VLAN（load:0->NXM_OF_VLAN_TCI[]）；设置VXLAN ID为当前数据帧的VXLAN ID（load:NXM_NX_TUN_ID[]->NXM_NX_TUN_ID[]）；从当前被学习数据帧的入端口送出（output:OXM_OF_IN_PORT[]）。

为什么数据帧里面既有VLAN ID，又有VXLAN ID？因为VTEP（br-tun）收到VXLAN数据解封装之后，VXLAN ID作为数据帧的元数据，与数据帧一起送到了OpenFlow流表处理。至于VLAN ID，是在table 4根据VXLAN ID对应过来的。

我们来看一下在table 20，通过学习生成的流表。

```
$ sudo ovs-ofctl dump-flows br-tun table=20
table=20, hard_timeout=300,
priority=1,vlan_tci=0x0005/0x0fff,dl_dst=fa:16:3e:10:51:07 actions=load:0->NXM_OF_VLAN_TCI[],load:0x1c->NXM_NX_TUN_ID[],output:"vxlan-c0a81f27"
```

得到的结果与刚刚介绍的学习的过程一模一样，匹配VLAN ID，目的MAC地址，之后剥离VLAN，设置VXLAN ID，再从某一个端口送出。这个端口，就是之前学习的数据帧的入端口。这条流表有个参数是hard_timeout，这个是流表的老化时间。这个参数表明300秒以后，这条流表会被自动删除。

br-tun的不同端口上配置了不同的远端VTEP IP地址，从br-tun的某个端口送出，实际就是送到了这个端口对应的远端VTEP。例如下面的输出里面，vxlan-c0a81f27对应的远端VTEP的IP地址就是192.168.31.39。

```
$ sudo ovs-vsctl show
… ...
Port "vxlan-c0a81f27"
            Interface "vxlan-c0a81f27"
                type: vxlan
                options: {df_default="true", in_key=flow, local_ip="192.168.31.110", out_key=flow, remote_ip="192.168.31.39"}
```

OpenStack Neutron用软件和OpenFlow实现了VTEP，进而实现VXLAN网络。VTEP的MAC地址表，VLAN/VXLAN对应关系表，都以OpenFlow的形式，存在于br-tun上。不过默认情况下，主要的控制层信息—MAC地址表，还是通过数据层的学习获得。

基于源拷贝的BUM

—

前面介绍的传统VXLAN网络，是通过underlay网络的IP组播实现flood-learn的。其实不只flood-learn，BUM（Broadcast，Unknown Unicast，Multicast）都是通过IP组播实现的。IP组播用起来不太方便，因为一是需要硬件支持，二是需要额外的配置。

Arista作为VXLAN协议的制定者之一，提出采用源拷贝的方式来实现VXLAN的BUM，从而摆脱对IP组播的依赖[2]。具体来说，在原先需要做BUM的场合，源VTEP将数据帧在源端复制多份，再一一发送给相应的所有远端VTEP。这样既达到了BUM的效果，又不需要特殊的underlay网络的支持。OpenStack Neutron也采用了这种方式实现BUM。

首先在br-tun的table 2，识别单播（Unicast），广播（Broadcast）和组播（Multicast）数据帧。

```
$ sudo ovs-ofctl dump-flows br-tun table=2
table=2, priority=0,dl_dst=00:00:00:00:00:00/01:00:00:00:00:00
actions=resubmit(,20)
table=2, priority=0,dl_dst=01:00:00:00:00:00/01:00:00:00:00:00
actions=resubmit(,22)
```

因为广播的目的MAC地址是FF:FF:FF:FF:FF:FF（所有bit都是1），组播的目的MAC地址第一个字节的最低bit必然是1（IEEE 802.1D）[3]，单播的目的MAC地址第一个字节的最低bit必然是0，所以单播被送到table 20，广播和组播被送到送到table22。先来看table20，刚刚说过，这是VTEP的MAC地址表所在的table。VTEP在做转发的时候，如果在MAC地址表里找不到目的MAC地址对应的信息，那么这个单播包对于VTEP来说就是未知单播（Unknown Unicast），接下来就要泛洪（flood）。在table20，有一条优先级最低的OpenFlow流表，这条流表什么也不做，只是将数据转到table22。因为优先级最低，如果走到这条流表，那说明table 20里面没有其他的流表能匹配目的MAC地址，当前的数据帧是一个未知单播。

```
$ sudo ovs-ofctl dump-flows br-tun table=20
table=20, priority=0 actions=resubmit(,22)
```

就这样，虽然路径不太一样，但是BUM数据帧，最后都走到了table 22。在table22，会做源拷贝，将BUM送到所有相关的远端VTEP。

```
$ sudo ovs-ofctl dump-flows br-tun table=22
table=22, priority=1,dl_vlan=4 actions=strip_vlan,load:0x3->NXM_NX_TUN_ID[],output:"vxlan-c0a81f27",output:"vxlan-cae3027f"
table=22, priority=1,dl_vlan=5 actions=strip_vlan,load:0x1c->NXM_NX_TUN_ID[],output:"vxlan-c0a81f27",output:"vxlan-e98a9e17"
```

所以，OpenStack Neutron区别于传统的VXLAN网络，没有基于IP组播来实现VXLAN的BUM，而是基于源拷贝的方式来实现BUM。但是这里还有一个问题，table 22里的OpenFlow流表怎么知道相关的远端VTEP是哪些？这里就需要Neutron去计算、更新并下发相应的VTEP信息。直观的说，当有新的VTEP加入或者离开VXLAN网络，需要Neutron去更新table22里的流表。这里的OpenFlow流表也是控制层信息，因为它决定了数据帧的转发。但是这里使用了一个独立于数据层的组件，也就是OpenStack Neutron来更新控制信息。这里已经有点控制层，数据层分离的意思了。

L2 Population

—

传统的网络下，都是通过数据层学习，来更新作为“控制层信息”的MAC地址表。这种方式本身没有问题，但是如果网络规模变大，相应的flood也会变多。如果把网络通信比喻成两个人说话，那flood就是大喊着说话，让所有人都听见。试想一下，如果flood变多，那整个网络环境就会像是菜市场一样，正常的谈话必然会受到影响。针对这个问题，OpenStack Neutron提供了L2 Population。当打开L2 Population的时候，OpenStack Neutron会根据自己掌握的信息，将所有可能的MAC地址，预先下发到VTEP（br-tun）的MAC地址表（OpenFlow table 20）里面。这样，VTEP在转发数据帧的时候，总是能查到MAC地址对应的信息，也就不用泛洪（flood）了。

打开L2 Population之后，br-tun上的table 20就会新增下面的OpenFlow流表。这个流表与前面学习到的流表，虽然看起来略有不同，但是实际效果是一模一样的。匹配VLAN ID，目的MAC地址，之后剥离VLAN，设置VXLAN ID，再从某一个端口送出。

```
$ sudo ovs-ofctl dump-flows br-tun table=20
table=20, priority=2,dl_vlan=5,dl_dst=fa:16:3e:10:51:07
actions=strip_vlan,load:0x1c->NXM_NX_TUN_ID[],output:"vxlan-c0a81f27"
```

唯一的区别是，这条流表没有老化时间，会一直存在。如果对应的MAC地址失效了，例如虚机被删除了，需要Neutron会去删除这条流表。现在，作为控制层信息的MAC地址表不再依赖数据层，完全由OpenStack Neutron来控制。

L2 Population可以抑制未知单播的泛洪。与L2 Population相关的一个功能是ARP responder。后者只有在前者打开的前提下才能工作。ARP responder有时也叫做ARP proxy，它在本地代答ARP查询，从而抑制ARP广播。因为与VXLAN关系不大，就不多说了。

打开L2 Population的OpenStack Neutron与传统的VXLAN网络已经很不一样了。打开L2 Population之后，数据层只需要传递数据。控制层信息由OpenStack Neutron下发。但是这种方式就一定是完美的吗？传统方式下，通过数据层学习来更新MAC地址表，每个VTEP只需要维护自己需要通信的MAC地址信息，甚至一段时间不用的MAC地址信息，会自动老化删除。这样，每个VTEP，只需要维护一个相对较小的MAC地址表。MAC地址表都是存在内存里面，所以，这种情况下，对内存的消耗较小。而采用L2 Population，每个VTEP会被预先下发所有可能的目的MAC地址信息，虽然免去了flood-learn，但是VTEP的MAC地址表一直是最大的状态，相应的对内存的消耗也要更大。这两种方式各有利弊，没有说哪个一定优于另一个，正因为如此，OpenStack Neutron才保留了两种方式，将选择权交给用户。

[1] https://tools.ietf.org/html/rfc7348#page-8

[2] https://www.arista.com/en/company/news/press-release/21-company/press-release/1016-pr-20141022

[3] http://standards.ieee.org/develop/regauth/tut/macgrp.pdf