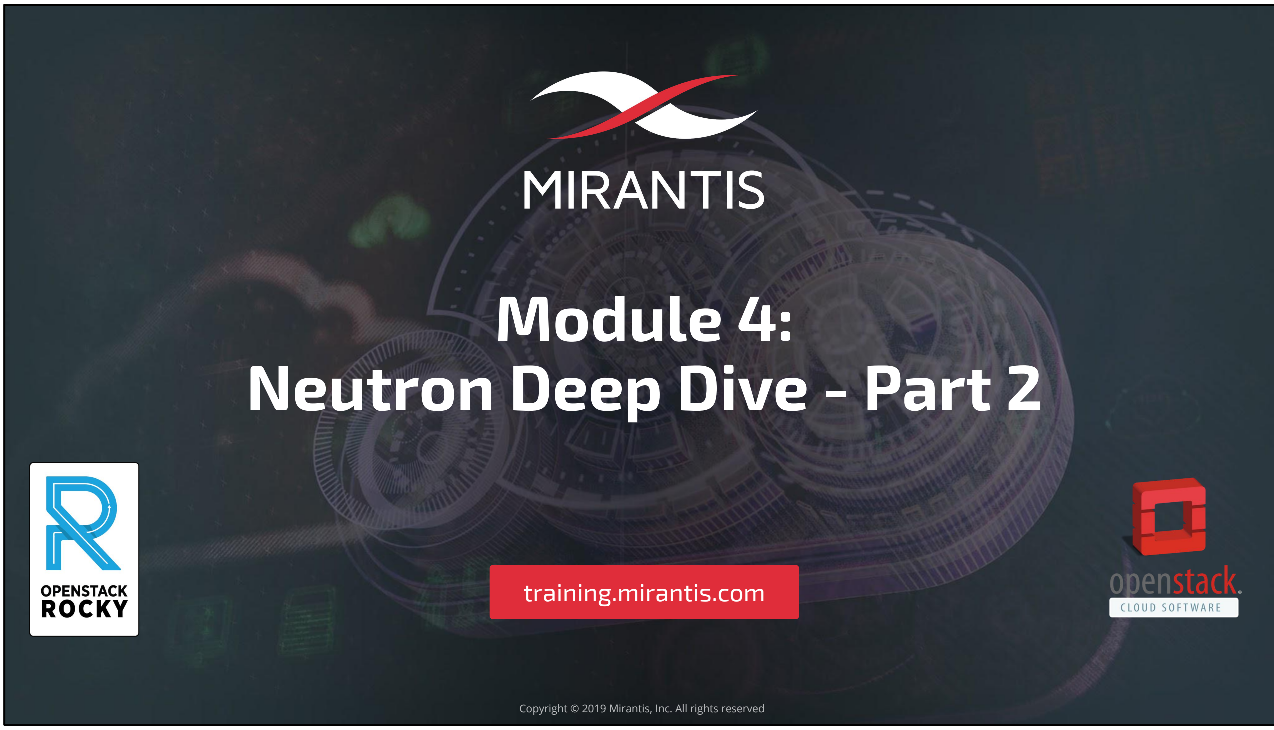


This presentation provides an overview and details related to the Neutron networking service in OpenStack.
Neutron is a core component of OpenStack that virtualizes network components, emulating the physical network. Networks can be created on request from the CLI, REST API, or Dashboard UI.

Neutron provides support to build networks/subnetworks and routers plus advanced network topologies and policies for load balancing, firewalls,or VPN.

Neutron provides the networking objects, including vendor support through the use of plugins (pluggable python classes), that the other OpenStack components use.
There are many network vendors. That leads to many Neutron plugins that can be downloaded from the OpenStack Marketplace,git,or directly from the vendor.

Inbound and outbound traffic flow

Fixed versus floating IP addresses

IPtables: SNAT / DNAT

Packet filtering

Neutron Security Groups

Metadata service

MIRANTIS

It is time to look deeper into what happens *under the covers*.
As you create the network and deploy VMs, IPtable rules are automatically created and updated for SNAT and DNAT, for example.
Neutron security groups allow you to define rule to access deployed VMs. For example, *allow ingress traffic to flow on port 22*. As such, security groups define packet filtering rules.

IPs: Fixed versus floating

- Fixed IPs:
  - Assigned to each instance at boot
  - Private IP ranges (10.0.0.0, 192.168.0.0, etc.)
  - Used for communication between instances and outbound to external networks
  - Inaccessible from external networks
- Floating IPs:
  - Pools of *publicly routable* IPs registered in OpenStack by cloud administrator
  - Allocated and associated to instances by cloud users
  - Provides access to instances from external networks (inbound)
    - Uses DNAT to translate floating IP to fixed IP of instance
  - Supports multiple floating IP pools, leading to different internet service providers (ISPs)

MIRANTIS

In OpenStack, there are two different types of IP addresses: fixed and floating IPs. Fixed IPs are the private addresses given to an instance. These allow for communication between instances, and communication to external networks. Floating IPs are public addresses that allow for external access into instances.

A few more details …

- As virtual machines are deployed:
  - Each VM instance is assigned a static IP address in the *private* network
  - IPtable rules are created for NAT: (stored in ***qrouter*** namespace)
    - SNAT (source NAT) for **outbound** (egress) requests *from* VM
      - Translate source (static) IP address in request to public IP address
    - DNAT (destination NAT) for **inbound** (ingress) requests *to* VM
      - Floating IP addresses are needed to access VMs
      - Translate target IP address in request to static IP address of instance
  - Security Group rules needed to allow access to ports on the VMs, such as port 22 for SSH
    - Creates rules (IPtables or OpenFlow tables) for packet filtering

MIRANTIS

Inbound communication (to a VM) is controlled by:
- **SNAT** (source network address translation)
- Requires a *floating IP address*
- Security group rules

Outbound communication (from a VM) is controlled by:
- **DNAT** (destination network address translation)

SNAT and DNAT are implemented with the *nat table* on the network node, **stored in the qrouter- namespace**.

Since the NAT table is stored in the qrouter namespace, you must issue an **ip netns exec** command to display its rules.

IPtables overview

- A user-space application program that allows a system administrator to configure tables provided by the Linux kernel **firewall** and the **chains and rules** it stores
  - NAT table: For *network address translation*
  - Filter table: For *packet filtering* (used only with Linux Bridge)
- Chains are a list of rules
  - Might be stitched together (for example, daisy-chained) to enforce complex logics
- Rules: Each rule specifies the matching criteria of an IP packet and the action it should take
  - Match: source/destination IP, source/destination port, connection state, etc.
  - Action: ACCEPT, REJECT, DROP, LOG, **DNAT**, **SNAT**, etc

IP tables use the Linux kernel firewall to route packets and translate (SNAT, DNAT) IP addresses.

There are four built-in tables. Each contains several chains

- Filter table (Default IPtable): For *packet filtering*
  - used on compute node when using Linux Bridge
  - The default (*reference implementation*) uses OVS for bridging. In that case, OpenFlow tables are used for packet filtering
- NAT table: For *network address translation* (used by network node)
- Mangle table: For packet manipulation (not used by Neutron)
- Raw table: For configuration exemptions (not used by Neutron)

<u>Example Actions:</u>

- ACCEPT: allows the packet to pass
- REJECT: disallows the packet, returning an error to the requestor
- DROP: disallows the packet, does not send a response to the requestor
- LOG: logs the packet to syslog, continues to the next rule in the chain
- DNAT: Rewrites the destination IP address or port in the packet, continues to the next rule in the chain
- SNAT: Rewrites the source IP address or port in the packet, continues to the next rule in the chain

IPtable and NAT

- Each **router** creates a unique *qrouter-* namespace
- SNAT/DNAT rules are stored in the NAT table for the *qrouter-* namespace
  - **SNAT (Source NAT):** From private network to external network
  - **DNAT (Destination NAT):** From external network to private network
- The following chains are used in NAT table
  - PREROUTING chain: Alters destination IP address before routing (DNAT)
  - POSTROUTING chain: Alters source IP address after routing (SNAT)
  - OUTPUT chain: For locally generated IP packets

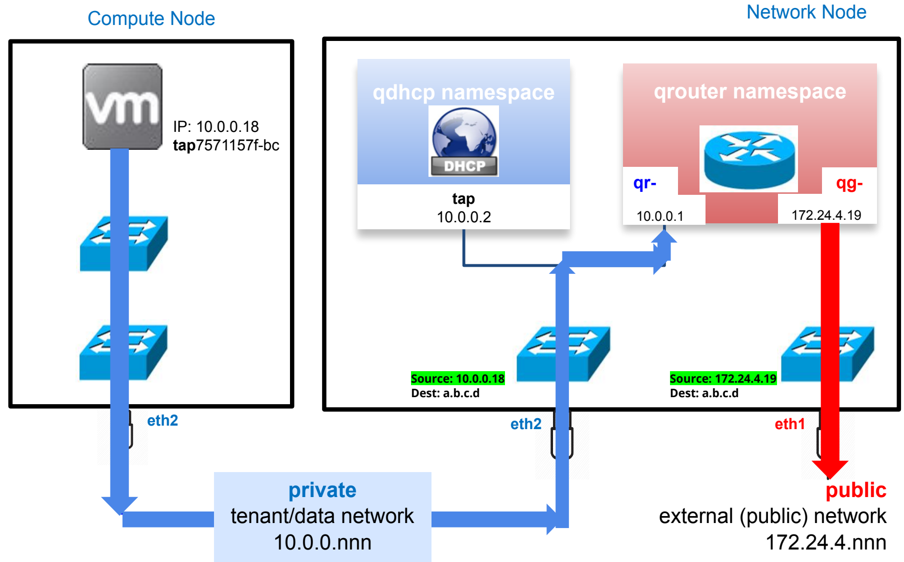
MIRANTIS

SNAT: for outbound packets; modifies the private network IP address (static) used by virtual machines when connecting to the public (external) network.
DNAT: for inbound packets; modifies the public network IP address (floating IP) used by users connecting in to the virtual machine instance.

Since the NAT table is stored in the qrouter namespace, you must issue an **ip netns exec** command to display it.

Example - Outbound flow: SNAT with static IP

- Each instance has a fixed IP in the private network
  - 10.0.0.18
- Assigned by DHCP (dnsmasq)
- No floating IP
- Outbound traffic (from VM) is SNATed to to the gateway IP address (qg-) on the external (public) network
  - 172.24.4.19
  - NAT table resides in ***qrouter*** namespace



MIRANTIS

7

Notice the IP addresses for the ports:

- **qr-** is assigned the IP address of the gateway (the first IP address in the private pool)
- **qg-** is assigned the first IP address in the public pool
- **tap** (DHCP) is assigned the first *available* IP address in the private pool

Each virtual machine instance has at least 1 interface (vif, eth0); it is connected to the private (tenant/data) network. In the example above, its IP address is 10.0.0.18. To access the external (public) network, the 10. address must be translated to a 172. address. Without a floating IP address, the 10.internal address is translated to the address of the **qg- interface** in the **router namespace**. This is known as **source NAT** (SNAT).

dnsmasq is the default DHCP server for the tenant network; assigning fixed IP addresses to the virtual machines. When dnsmasq starts, it binds to the tap interface in the qdhcp namespace.

IPtable example: outbound (SNAT)

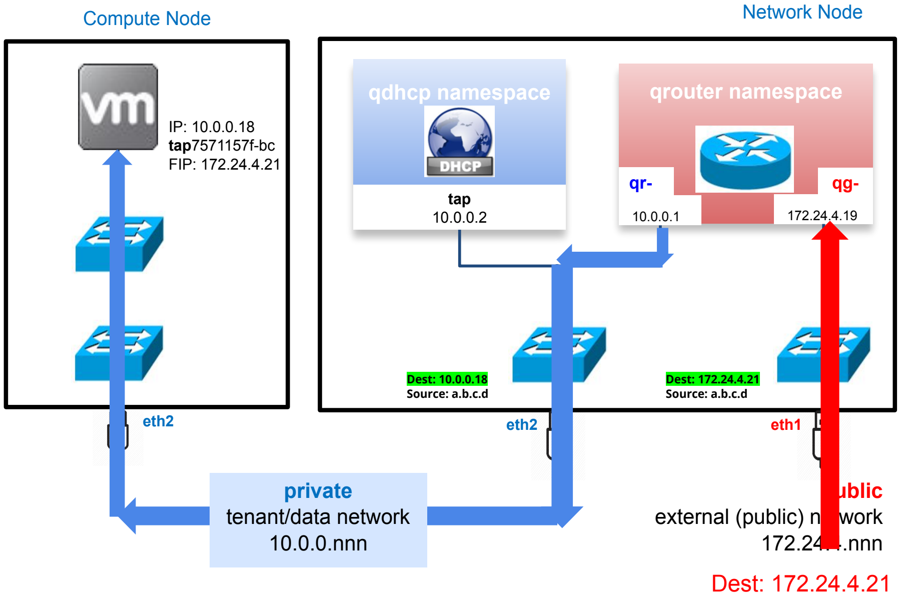
- Without floating IP, all outbound requests are routed to the *qg- interface*
  - Translate IP addr to qg- interface (172.24.4.19)
  - Regardless of instance IP

```
Chain neutron-l3-agent-float-snat


Chain neutron-l3-agent-snat (1 references)
num  target prot opt source               destination
1     neutron-l3-agent-float-snat  all  --  anywhere              anywhere
2     SNAT         all  --  anywhere             anywhere             to:172.24.4.19
3     SNAT         all  --  anywhere             anywhere             mark match !
                                                                      0x2/0xffff ctstate DNAT to:172.24.4.19
```

Example - Inbound flow: DNAT with floating IP

- A floating IP is required for inbound connections
  - 172.24.4.21
- Floating IP is manually associated with VM
  - Not assigned by DHCP
  - Not visible inside VM
  - Can be freed up and reused for another VM
- Floating IP is DNATed to the VM IP
  - 10.0.0.18
  - NAT table resides in ***qrouter*** namespace



Notice the IP addresses for the ports:

- **qr-** is assigned the IP address of the gateway (the first IP address in the private pool)
- **qg-** is assigned the first IP address in the public pool
- **tap** (DHCP) is assigned the first *available* IP address in the private pool

Each VM can optionally have an IP address on the public (external) network. The IP address is assigned manually, after the VM instance is deployed. This is known as a ***floating IP address***. It supports access from the public (external) network inbound to the virtual machine instance. For example, to access the VM instance with an IP address of 10.0.0.18, a floating IP of 172.24.4.21 is assigned. The 172. IP address is translated to the 10. address. This is known as **destination NAT** (DNAT).

This allows users to access the VM instance with the 172.24.4.21 IP address. For example, they would SSH to 172.24.4.21.

In some implementations, such as Amazon Web Services (AWS), floating IPs are called *Elastic IPs*.

IPtable example: inbound (DNAT)

- Translate floating IP addr (172.24.4.21) to instance IP addr (10.0.0.18)

```
Chain neutron-l3-agent-OUTPUT (1 references)
num  target     prot opt source               destination
1    DNAT       all  --  anywhere             172.24.4.21          to:10.0.0.18
```

- Once associated, a new rule is created - floating IP is also used for outbound (SNAT)

```
Chain neutron-l3-agent-float-snat (1 references)
num  target     prot opt source               destination
1    SNAT       all  --  10.0.0.18            anywhere             to:172.24.4.21
```

Neutron Security Groups

Creating *firewall rules* for VM-VM communication, as well as, ingress and egress communication

MIRANTIS

Neutron security groups control traffic between VMs (VM-VM communication), as well as, egress (outbound) and ingress (inbound).

By default, all packets are dropped for ingress (inbound) traffic. For example, if you need to SSH into a VM instance, you need to allow *ingress traffic on port 22*.

Neutron security groups define packet filtering rules to allow connections to specific ports, such as port 22 for SSH..

Neutron security groups

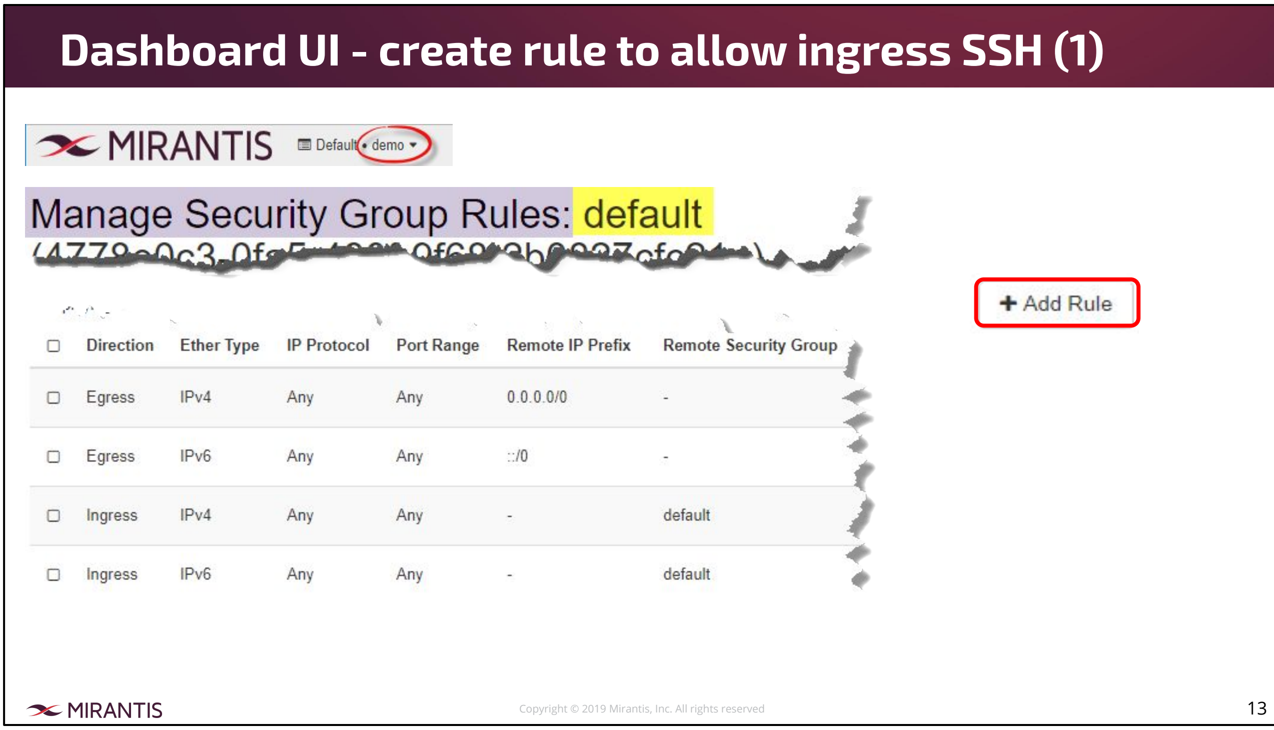
- Sets of *IP filter* rules; applied against VM instance
- Each project has a **default** security group, with rules as follows:
  - All *egress* (outbound) traffic allowed
  - All VM-to-VM communication allowed
    - Within VMs using same security group, by default
  - All *ingress* (inbound) traffic dropped
- Must create security group rules to allow
  - SSH, HTTP, ping, etc.
  - Only traffic that matches security group rules is allowed
    - Otherwise, all traffic dropped
- Supports IPv4 and IPv6

Security groups are sets of IP filter rules that are applied to an instance. Security groups can allow traffic in to a VM instance (ingress) or out of an instance (egress). They are project-specific and project members can edit the default rules for their group as well as add new rule sets.

In essence, security groups act like a virtual firewall for your virtual machine instances, providing **port-level security**.

Suppose you need to open port 22 for SSH connections in to your VM instances. You have 2 choices:

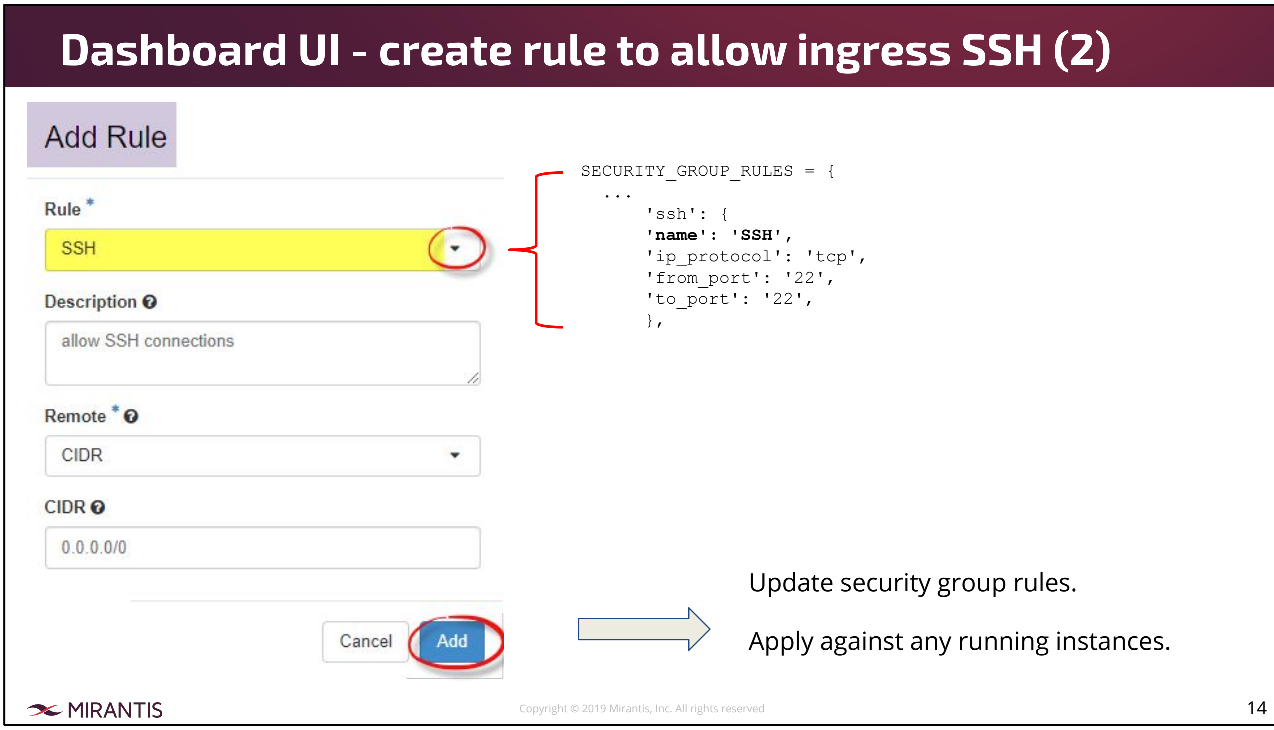
- Update the default security group for your project to include a rule for ingress traffic on port 22
- Create a new security group and include the rule for ingress traffic on port 22



This slide shows an example of the **default** security group for the ***demo project***:

- All egress (outbound) traffic allowed (IPv4 and IPv6)
- All VM-to-VM communication allowed within VMs using same (default) security group. Notice the Remote Security Group column?
- All ingress (inbound) traffic dropped, by default

Click **Add Rule** to create the SSH ingress rule. See the next slide ...



Continued from the previous slide …
After clicking **Add Rule**, the *Add Rule* dialog is displayed.
Neutron provides several commonly used rules, such as SSH, ICMP, DNS, RDP, HTTP, SMTP, etc. The rules are defined in the **local_settings.py** configuration file for the Dashboard. An example of the pre-defined SSH rule is shown on the slide.
You can also define custom rules.

Click **Add** to update the **default** security group for the project. The security group rules create IPtables rules, and are applicable to all instances using the
All VM instances deployed using the default security group can now have ingress SSH connections.

OpenFlow rules example: allow ingress traffic to port 22

```
sudo ovs-ofctl dump-flows br-int | grep tp_dst=22
cookie=0x43754fdaf4d1d490, duration=72914.624s, table=82, n_packets=0,
  n_bytes=0, idle_age=65534, hard_age=65534,
  priority=77,ct_state=+est-rel-rpl,tcp,reg5=0x2b,
  tp_dst=22 actions=output:43

cookie=0x43754fdaf4d1d490, duration=72914.624s, table=82, n_packets=0,
  n_bytes=0, idle_age=65534, hard_age=65534,
  priority=77,ct_state=+new-est,tcp,reg5=0x2b,tp_dst=22
  actions=ct(commit,zone=NXM_NX_REG6[0..15]),output:43,resubmit(,92)
```

Route to OpenFlow *port* 43

Port 43 rule routes packet to tap interface on VM

```
sudo ovs-ofctl dump-ports br-int 43
port "tap7571157f-bc": rx pkts=106, bytes=10589, drop=0, ...
```

MIRANTIS

The flow table in OpenFlow switches plays a critical role in OpenStack networking. The flow table stores the rules (populated by the OpenFlow controllers) for ***controlling and directing the packet flows***.

The **Output action** forwards a packet to a specified OpenFlow port. In this example, packets for *established* and *new* connections on port 22 are routed to OpenFlow port 43. The rule for port 43 forwards packets to the ***tap interface*** of the VM. In this example, the ID of the port on the VM begins with “7571157f-bc.”

You can see the tap interface ID if you issue an **openstack port list** command

Note: These flows are defined in *table 82*. Table 82 accepts established and related connections. OpenFlow supports many tables. A packet might be routed through multiple tables. If the packet is not routed, it is dropped, by default.

To learn more about OpenFlow:
https://overlaid.net/2017/02/15/openflow-basic-concepts-and-theory/

For more details on Neutron and OpenFlow rules:
https://docs.openstack.org/neutron/pike/contributor/internals/openvswitch_firewall.html

Firewall driver (packet filtering): back end choices

- Configured in **ml2_conf.ini**:

```
[securitygroup]
# Driver for security groups firewall in the L2 agent (string value)
firewall_driver = openvswitch | iptables_hybrid | …
```

Firewall_driver option	DNAT / SNAT	Packet filtering
openvswitch	Linux IPtables	OpenFlow tables
iptables_hybrid	Linux IPtables	Linux IPtables

MIRANTIS

Historically, Open vSwitch (OVS) could not interact directly with *iptables* to implement security groups. Thus, the OVS agent and Compute service used a *Linux Bridge* (**qbr-**) between each instance (VM) and the OVS integration bridge (br-int) to implement security groups. The *Linux Bridge* device contains the *iptables packet filtering* rules pertaining to the instance.

Due to the additional components between instances and the physical network infrastructure, you might experience scalability and performance problems.

To alleviate such problems, the OVS agent includes an optional firewall driver that natively implements security groups as flows in OVS rather than Linux bridge and *iptables*, thus increasing scalability and performance.

The native *OVS firewall implementation* requires kernel and user space support for ***conntrack***, thus requiring a minimum versions of Linux kernel 4.3 and Open vSwitch version 2.5 or newer.

Metadata service

Metadata service - overview

- Metadata service allows VM instances to retrieve instance-specific data
  - Public SSH keys, user data, init scripts, and so on
  - Reachable at http://169.254.169.254:80
- Neutron metadata service is a proxy - implemented by **nova-api** at port **8775**

MIRANTIS

How do you get from 169.254.169.254, port 80, to the nova-api on the controller node, port 8775? See the next slide for the details.

Metadata service - flow

1. VM sends metadata request to http://169.254.169.254:80
2. Outbound requests are routed to default gateway on Neutron router (**qg-** port of the **qrouter** namespace)
3. NAT table *redirects* the request from TCP port 80 to port **9697**
   - Neutron metadata proxy listens on port 9697 is In the **qrouter** namespace
4. Neutron metadata proxy relays the request to a **nova-api-meta** WSGI worker, at port **8775**, based on configuration file:

```
## metadata_agent.ini ##
nova_metadata_host = <host_ip_addr>
nova_metadata_port = 8775
```

MIRANTIS 19

The NAT table rule to redirect the requests from port 80 to port 9697:

```
Chain neutron-l3-agent-PREROUTING (1 references)
num   pkts bytes target prot opt in out   source           destination
1     393  23580 REDIRECT     tcp  --  qr-+     *      0.0.0.0/0
169.254.169.254   tcp dpt:80 redir ports 9697
```

Metadata example

Instances can retrieve the **public SSH key** (identified by keypair name when a user requests the new instance) by making a GET request to the metadata service.

For example:

$ curl http://169.254.169.254/2009-04-04/meta-data/public-keys/0/openssh-key

```
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAAAgQDYVEprvtYJXVOBN0XNKVVRNCRX6BlnNbI+US\
LGais1sUWPwtSg7z9K9vhbYAPUZcq8c/s5S9dg5vTHbsiyPCIDOKyeHba4MUJq8Oh5b2i71/3B\
ISpyxTBH/uZDHdslW2a+SrPDCeuMMoss9NFhBdKtDkdG9zyi0ibmCP6yMdEX8Q== Generated\
by Nova
```

MIRANTIS

For more information on the OpenStack metadata service:
https://docs.openstack.org/nova/rocky/user/metadata-service.html

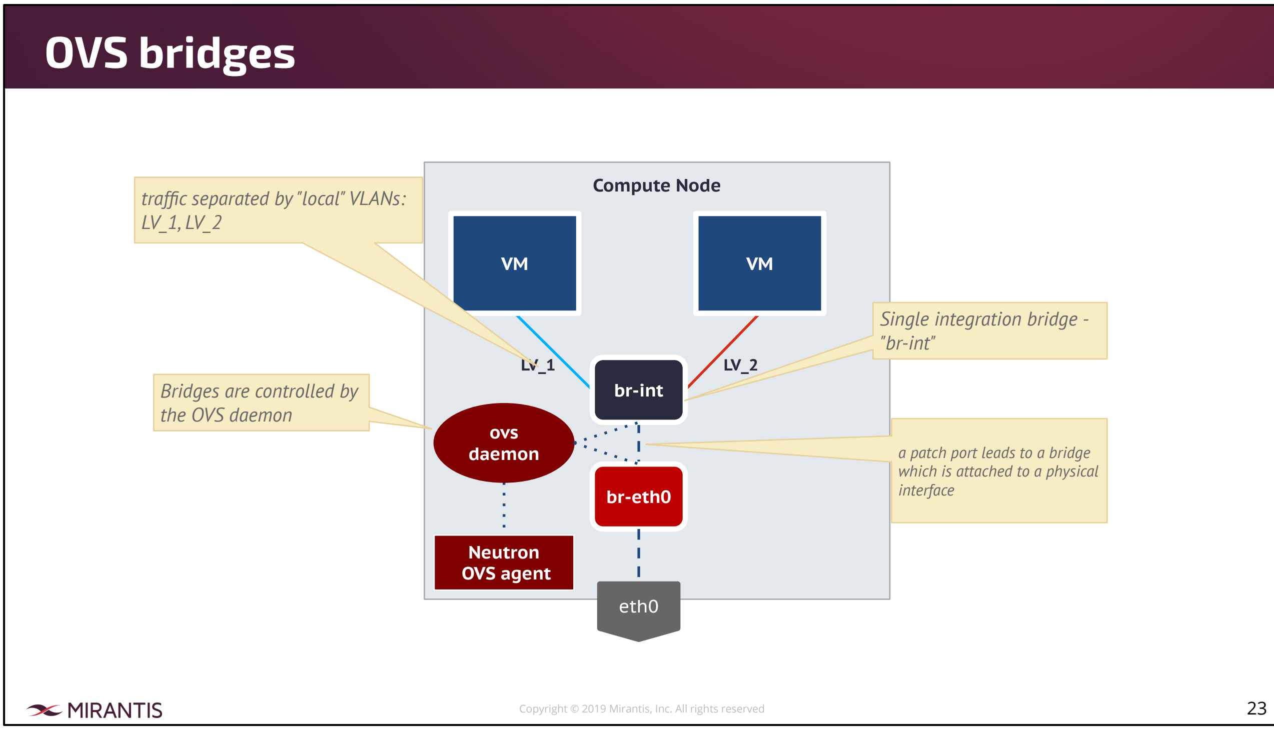
Neutron with Open vSwitch (OVS)

Open vSwitch plugin

- Open vSwitch (OVS) is an industry standard that supports the necessary layer 2 network types
  - FLAT: Networks share one layer 2 domain
  - VLAN: Networks are separated by 802.1Q VLANs
  - TUNNEL: Traffic is carried over GRE/VXLAN with different per-net tunnel IDs
- OVS also includes OpenFlow
- OVS is the *reference implementation* used by the ML2 plugin

With the OVS plugin (*reference implementation*), tenant traffic can be separated in 3 ways.

- FLAT - just like bridging
- VLAN - 802.1Q VLANs
- TUNNEL - tunnel connections between compute nodes with different tunnel_IDs (openvswitch concept based on FC 2890) for each network

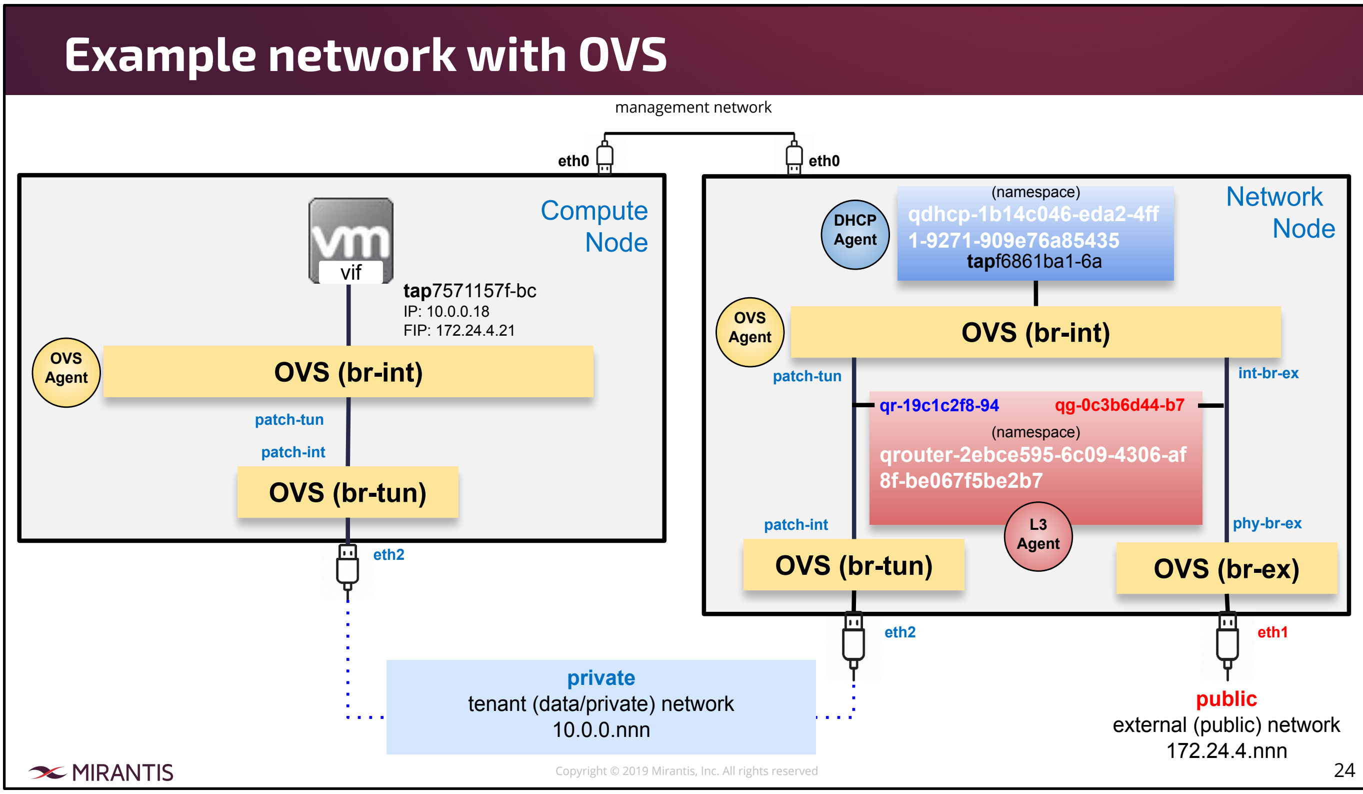


The OVS plugin creates the following bridges(virtual switches):

- **br-int**: the *integration bridge*. All VM instances connect to br-int, including instances from all tenants.
- One or more ethernet bridges. Sometimes, named br-ex or br-vlan. This slide has 1 ethernet bridge named br-eth0 which connects to the eth0 interface on the machine (compute node).

The OVS daemon (ovs-vswitchd) manages the bridges. The Neutron OVS agent interfaces with the OVS daemon.
The OVS bridges are connected by patch ports or v-eth ports.
In this example, packets inbound for 1 of the instances, flows from eth0,across the br-eth0 bridge, to the br-int bridge, to the instance.

Since there is a single integration bridge to potentially serve multiple tenants, there must be a way to separate tenant traffic on the bridge. OVS plugin uses *local vlans* for this. The local vlans exist only inside a given compute node and are not carried outside.(For passing traffic between different compute nodes they are stripped and converted to *global VLAN IDs*, or tunnel IDs, depending on the mode used.



This slide shows an example of the Network and Compute nodes, connected through the br-tun OVS bridge (and eth2 port) and the private network. A VM has been deployed to the Compute node, creating a tap interface to connect the VM to the OVS br-int.
Remember what each agent manages:

- OVS agent: interfaces with the **OVS daemon** (ovs-vswitchd) to manage OVS bridges and v-eth pairs
- L3 agent: manages qrouter namespace, including qr- and qg- interfaces
- DHCP agent: manages the qdhcp namespace, including the tap interface

The qrouter namespace name is derived from: **qrouter-<*router_UUID*>**
The qdhcp namespace name is derived from: : **qdhcp-<*private_network_UUID*>**

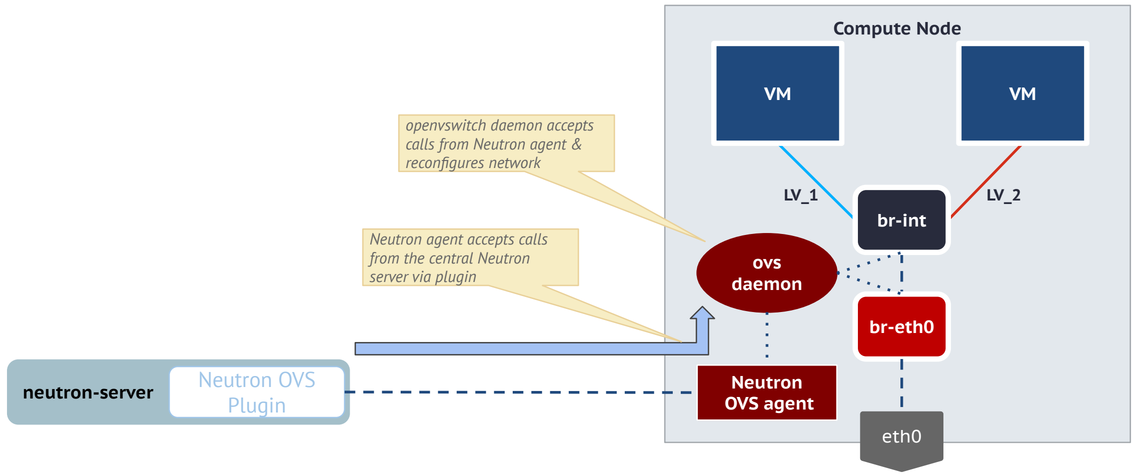
Where did the various UUIDs come from? Issue a **neutron port-list** command to display all of the interfaces (tap, qr-, qg-, and VM):

```
openstack port list (modified)
ID                                     | Fixed IP Addresses
06d6a361-1843-4796-a558-acd7344775ef | ip_address='192.168.0.2'  : DHCP interface  : tap-06d6a361-18 :
lb-mgmt
0c3b6d44-b7bd-4b37-ba7a-ba0e32e7e208 | ip_address='172.24.4.19'  : router gateway  : qg-0c3b6d44-b7  :
router1
19c1c2f8-9443-47c3-a16d-efe69f799ad5 | ip_address='10.0.0.1'    : router interface: qr-19c1c2f8-94  :
router1
2f92c654-391d-45ac-aa60-debf62d967d2 | ip_address='172.24.4.21'  : floating IP     :
7571157f-bcad-4032-8ad1-f0950600b737 | ip_address='10.0.0.18'   : VM instance     : tap-7571157f-bc
a87559e9-77eb-492d-aa55-1c7fcdc67587 | ip_address='192.168.0.18' : LB health mon   : o-hm0
f6861ba1-6a96-4e74-a2c9-e41c6ee5ef8f | ip_address='10.0.0.2'    : DHCP interface  : tap-f6861ba1-6a :
private
```

Notice the IP addresses for the ports:

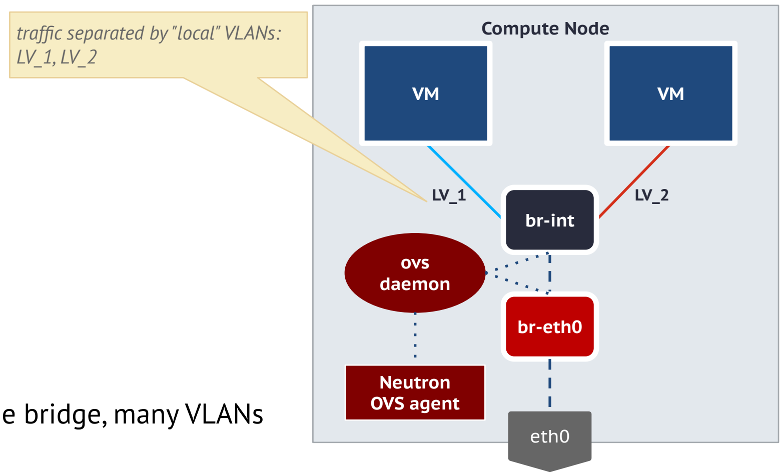
- **qr-** is assigned the IP address of the gateway (the first IP address in the private pool)
- **qg-** is assigned the first IP address in the public pool
- **tap** (DHCP) is assigned the first available IP address in the private pool

OpenvSwitch Plugin: OVS daemon



The bridges are manipulated by neutron-server via ovs_plugin in the following manner:
- we have one neutron-server running somewhere on the cloud controller node
- neutron-server uses ovs neutron plugin to communicate with neutron-agent running on compute-node
- neutron agent talks to ovs-daemon to manipulate openvswitch. ovs-daemon is not a part of Neutron. It is a system daemon which belongs to OVS installation

OpenvSwitch plugin: local VLANs

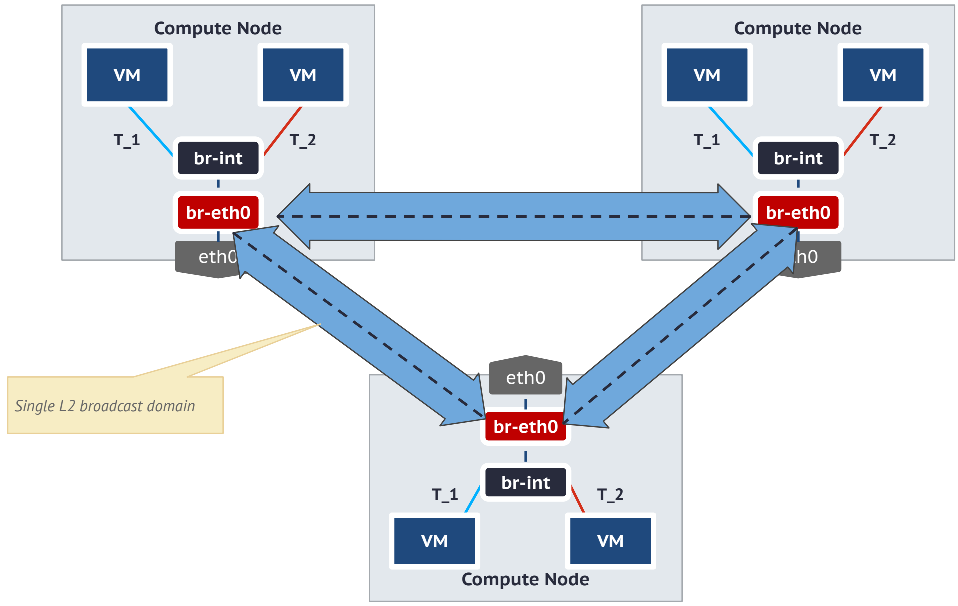


One bridge, many VLANs

Since we have a single bridge to serve potentially multiple tenants, there must be a way to separate tenant traffic on the bridge.
OVS plugin uses "Local vlans" for this.

"Local vlans" exist only inside a given compute node and are not carried outside.(For passing traffic between different compute nodes they are stripped converted to "global VLAN ids, or tunnel IDs, depending on the mode used - we will tell later about it).

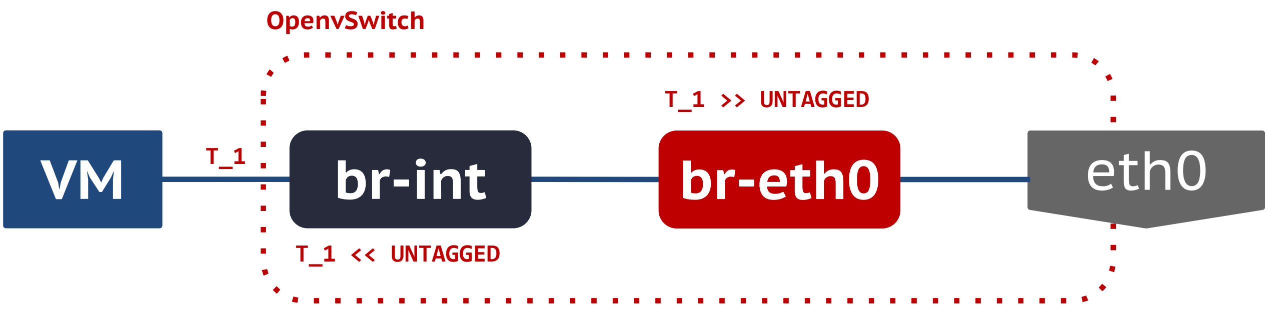
OpenvSwitch Plugin: FLAT Mode



The diagram shows 3 compute nodes, running VMs belonging to 2 tenants.

Traffic for all tenants is exchanged between compute nodes runs in a single layer 2 broadcast domain and is not separated by vlans or anything else.

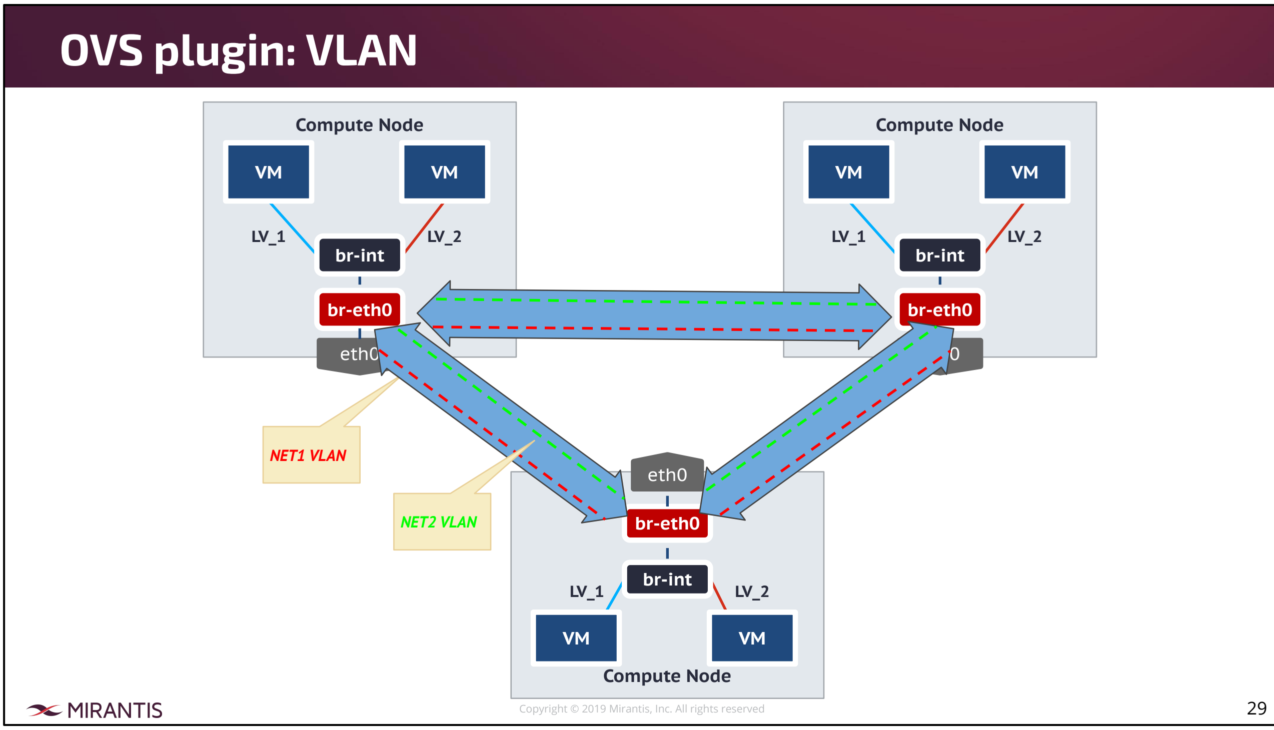
Local vs. Global Traffic IDs: FLAT Mode



Local tag (segmentation ID) stripped before sending down the wire

With neutron one can distinguish between local VIDs and global traffic IDs (my own naming). In neutron code they are also referred to as "segmentation_ID".

For the FLAT scheme, the local VIDs are just stripped by openvswitch and untagged packet is passed down the wire.

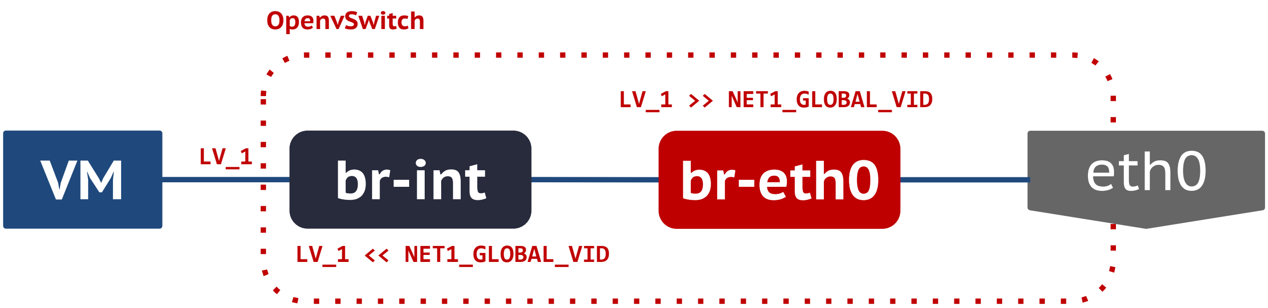


The diagram shows 3 compute nodes, running vm instances belonging to 2 tenants: NET1 and NET2 VLANs.

In VLAN mode traffic is separated by standard 802.1q vlan tags.

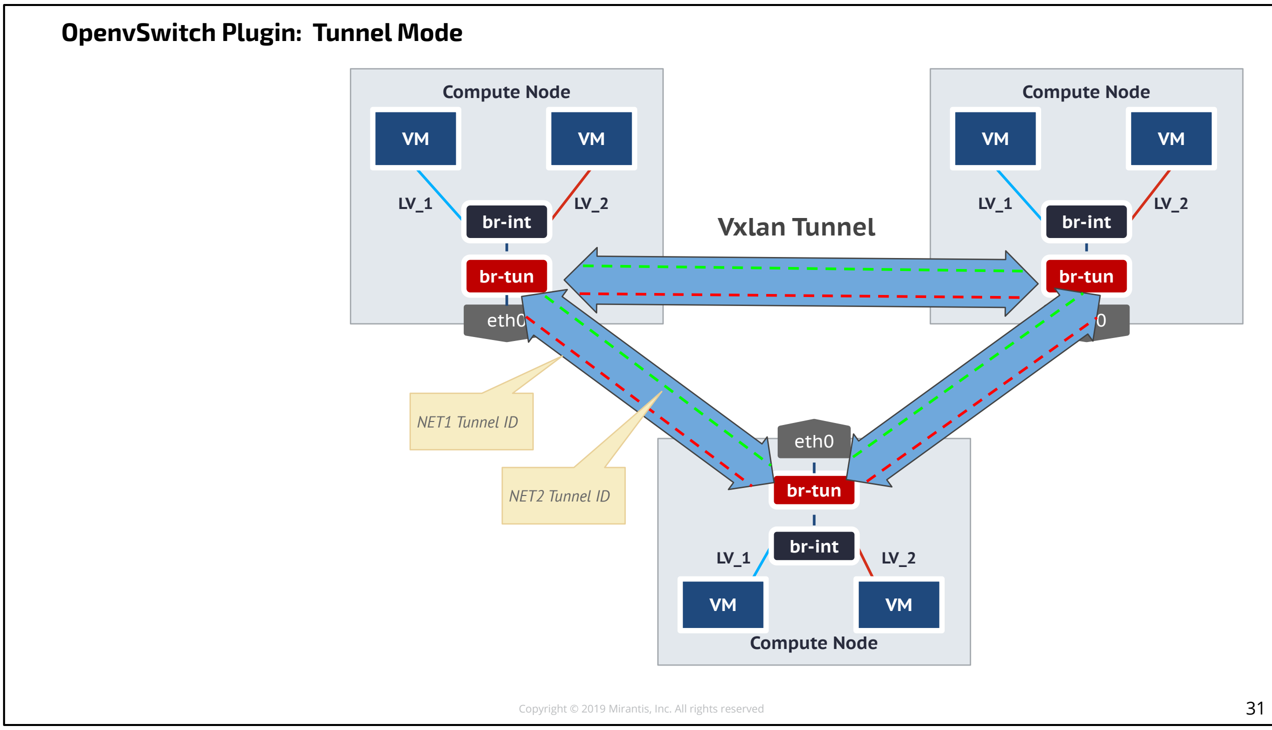
Using VLANs does have a limitation of slightly over 4000 VLAN IDs maximum.

Local vs. Global Traffic IDs: VLAN Mode



Local VLAN tag changed to "global" VLAN tag, also called “segmentation” VLAN ID

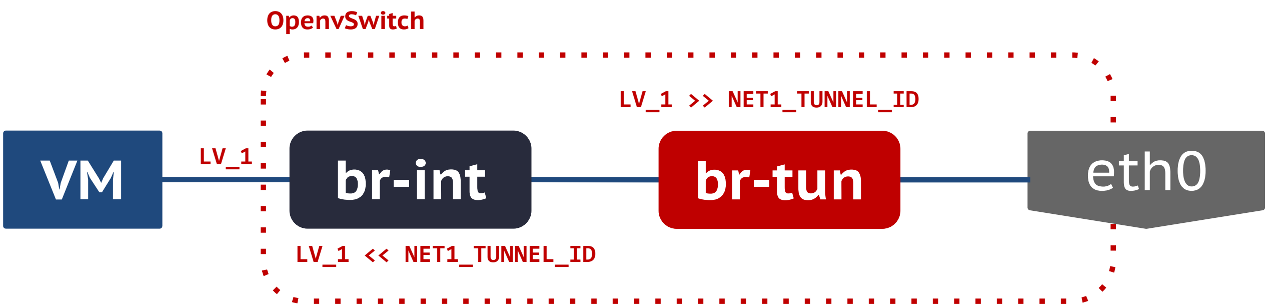
In VLAN mode, local vlan ID tags are converted to "global" vlan ID tags which correspond to those assigned to a given network by the Neutron administrator.



This one works the same like VLAN, with exception that a mesh of GRE tunnels (point-to-point) is created between hypervisor bridge br-tun IFs.

Networks are separated using "tunnel IDs".

Local vs. Global Traffic IDs: Tunnel Mode



Local VLAN tag changed to Vxlan tunnel ID

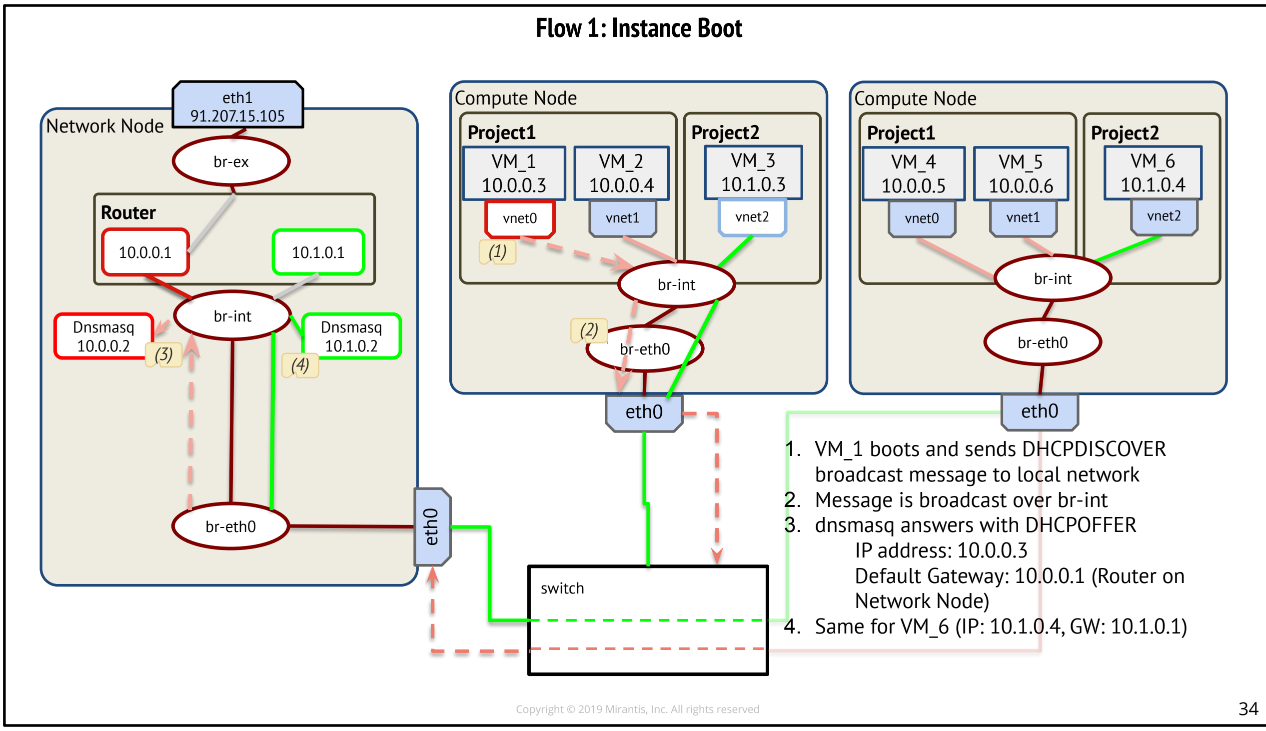
MIRANTIS

Local VIDs are converted to GRE tunnel IDs by openvswitch.

Traffic flow (reference)

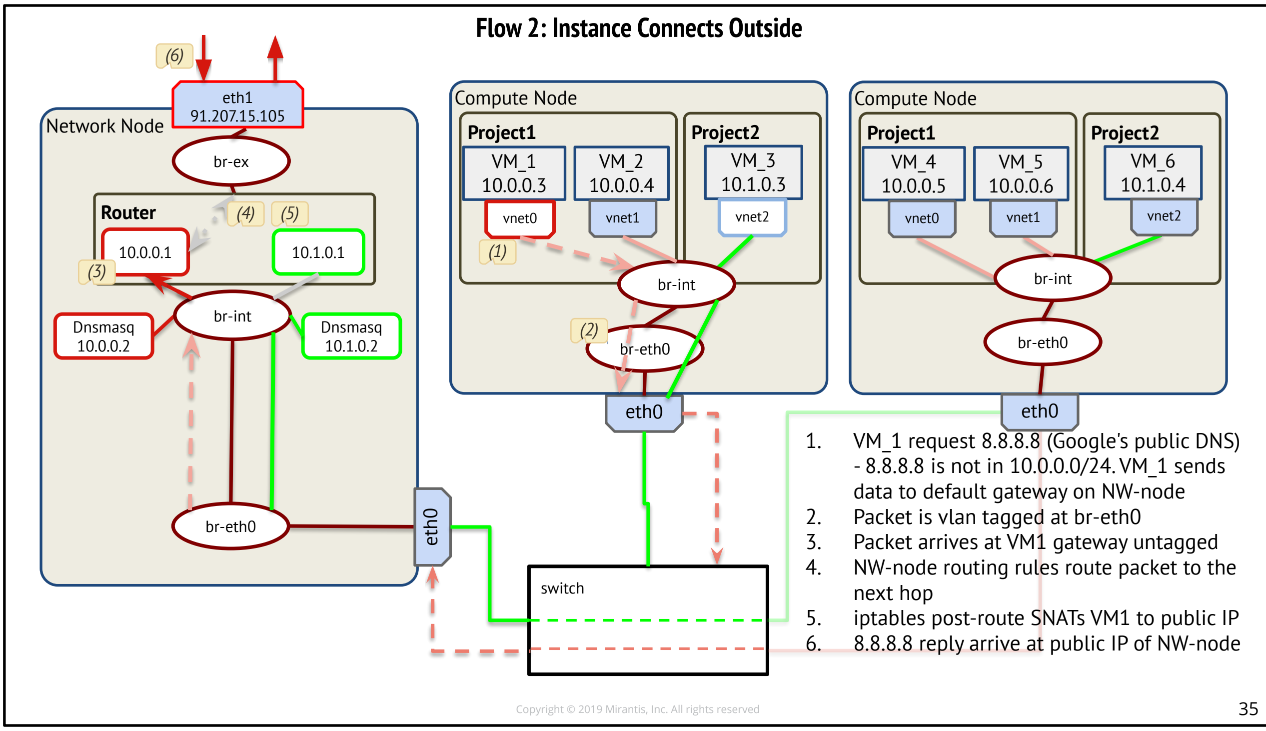
- Scenarios Key Assumptions
  - Single-host networking
  - Vlan Mode
  - Two tenants with separate networks

Here are our assumptions for the following flows. We're using multi-host networking with the VlanManager network manager. Two tenants have been created, each with their own separate network.



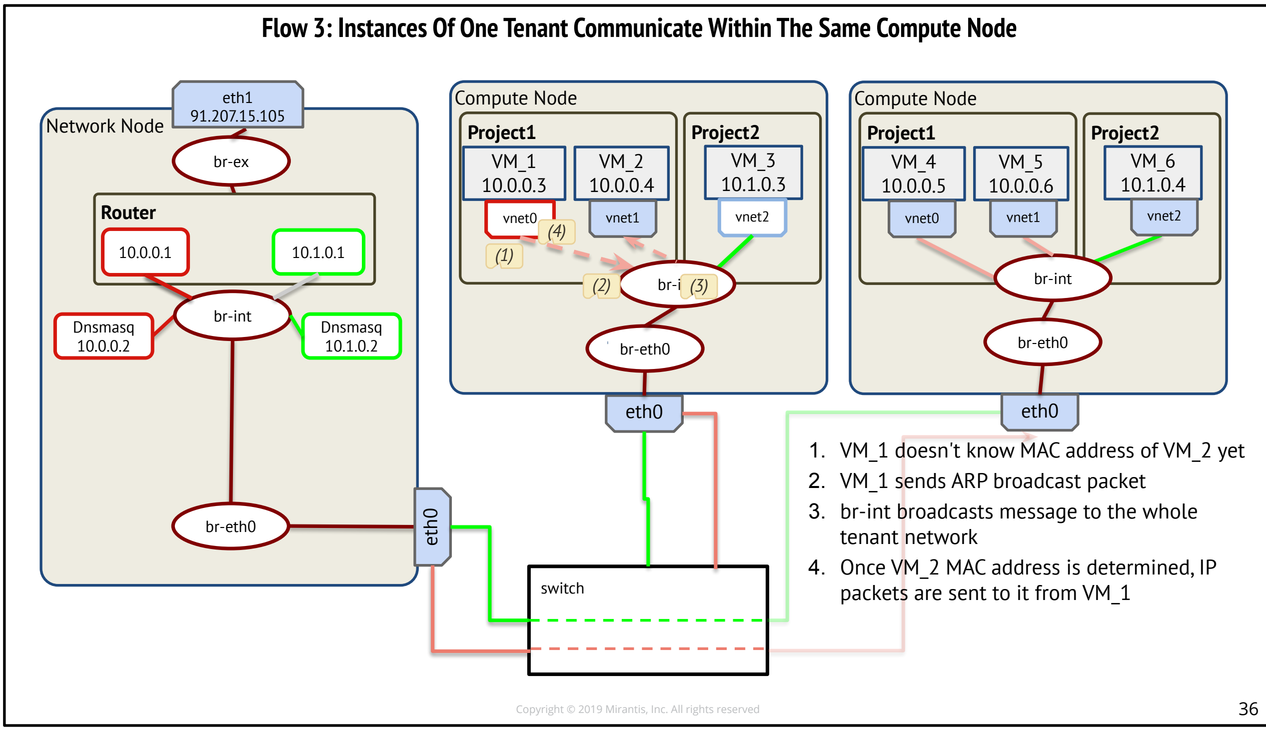
Two instances, in different tenants, on the same compute node, wish to communicate with one another (VM_1 to VM_3)

1. VM_1 sees that the destination of the packet resides on a different network (10.1.0.0), so it sends that packet to it's default gateway (10.0.0.1)
2. The packet arrives at br-int, then sent to network node
3. Network Node sees that the destination of the packet resides on a different network, so it consults it's routing table. There was an entry added to the routing table for the 10.1.0.0 network when VM_3 was launched, indicating that the entry point into the 10.1.0.0 network is through another router interface, so it sends the packet there.
4. The packet arrives at br-int, then sent back to compute node
5. Br-int broadcasts the ARP request to all connected devices. VM_3 responds with it's MAC address and is routed back to VM_1



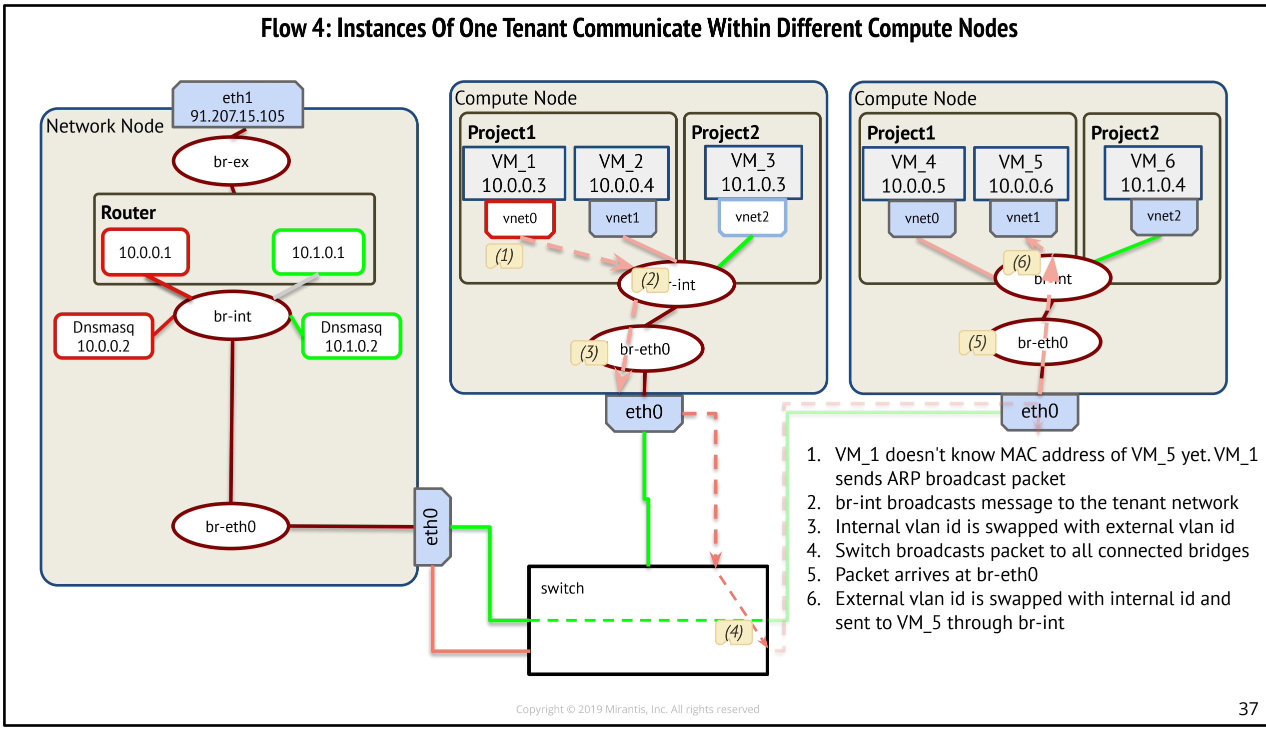
Two instances, in different tenants, on the same compute node, wish to communicate with one another (VM_1 to VM_3)

1. VM_1 sees that the destination of the packet resides on a different network (10.1.0.0), so it sends that packet to it's default gateway (10.0.0.1)
2. The packet arrives at br-int, then sent to network node
3. Network Node sees that the destination of the packet resides on a different network, so it consults it's routing table. There was an entry added to the routing table for the 10.1.0.0 network when VM_3 was launched, indicating that the entry point into the 10.1.0.0 network is through another router interface, so it sends the packet there.
4. The packet arrives at br-int, then sent back to compute node
5. Br-int broadcasts the ARP request to all connected devices. VM_3 responds with it's MAC address and is routed back to VM_1



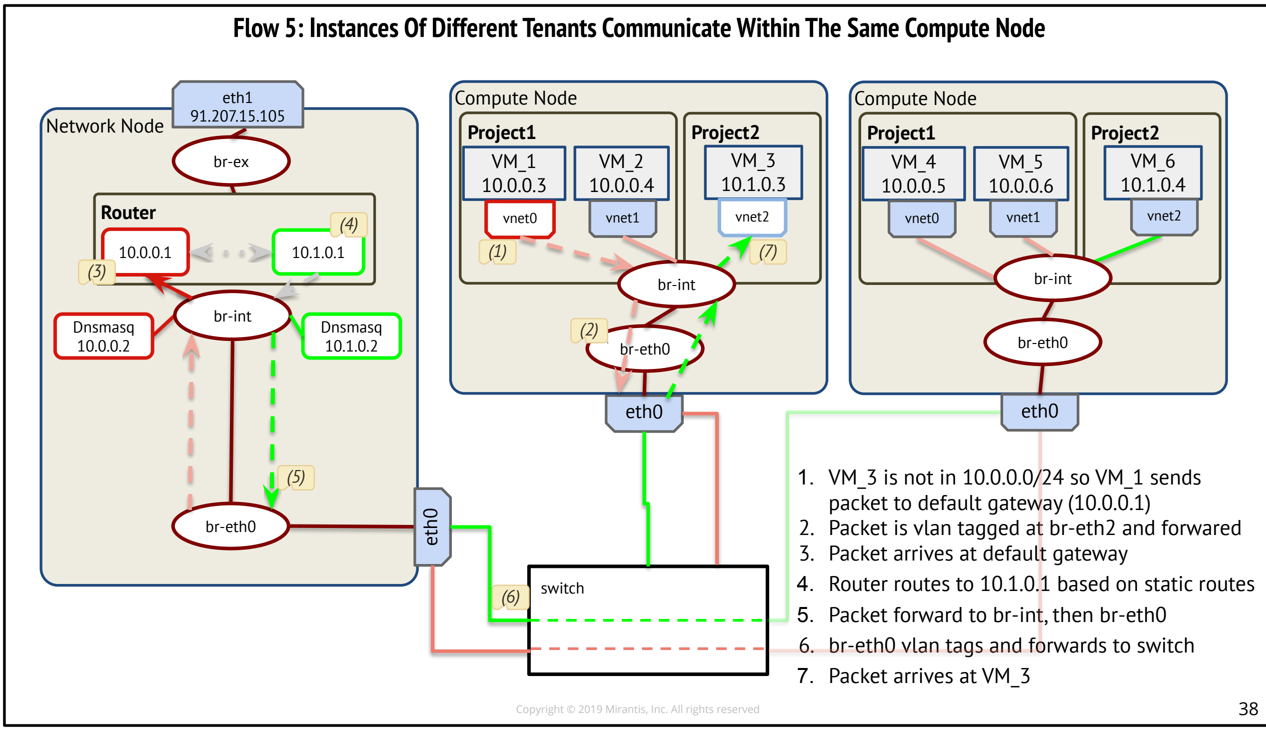
Two instances, in different tenants, on the same compute node, wish to communicate with one another (VM_1 to VM_3)

1. VM_1 sees that the destination of the packet resides on a different network (10.1.0.0), so it sends that packet to it's default gateway (10.0.0.1)
2. The packet arrives at br-int, then sent to network node
3. Network Node sees that the destination of the packet resides on a different network, so it consults it's routing table. There was an entry added to the routing table for the 10.1.0.0 network when VM_3 was launched, indicating that the entry point into the 10.1.0.0 network is through another router interface, so it sends the packet there.
4. The packet arrives at br-int, then sent back to compute node
5. Br-int broadcasts the ARP request to all connected devices. VM_3 responds with it's MAC address and is routed back to VM_1



Two instances, in different tenants, on the same compute node, wish to communicate with one another (VM_1 to VM_3)

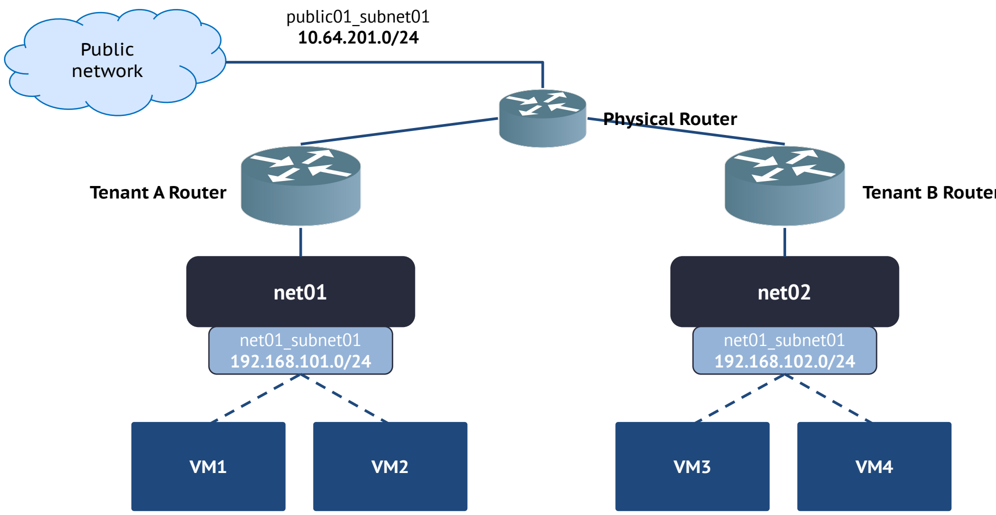
1. VM_1 sees that the destination of the packet resides on a different network (10.1.0.0), so it sends that packet to it's default gateway (10.0.0.1)
2. The packet arrives at br-int, then sent to network node
3. Network Node sees that the destination of the packet resides on a different network, so it consults it's routing table. There was an entry added to the routing table for the 10.1.0.0 network when VM_3 was launched, indicating that the entry point into the 10.1.0.0 network is through another router interface, so it sends the packet there.
4. The packet arrives at br-int, then sent back to compute node
5. Br-int broadcasts the ARP request to all connected devices. VM_3 responds with it's MAC address and is routed back to VM_1



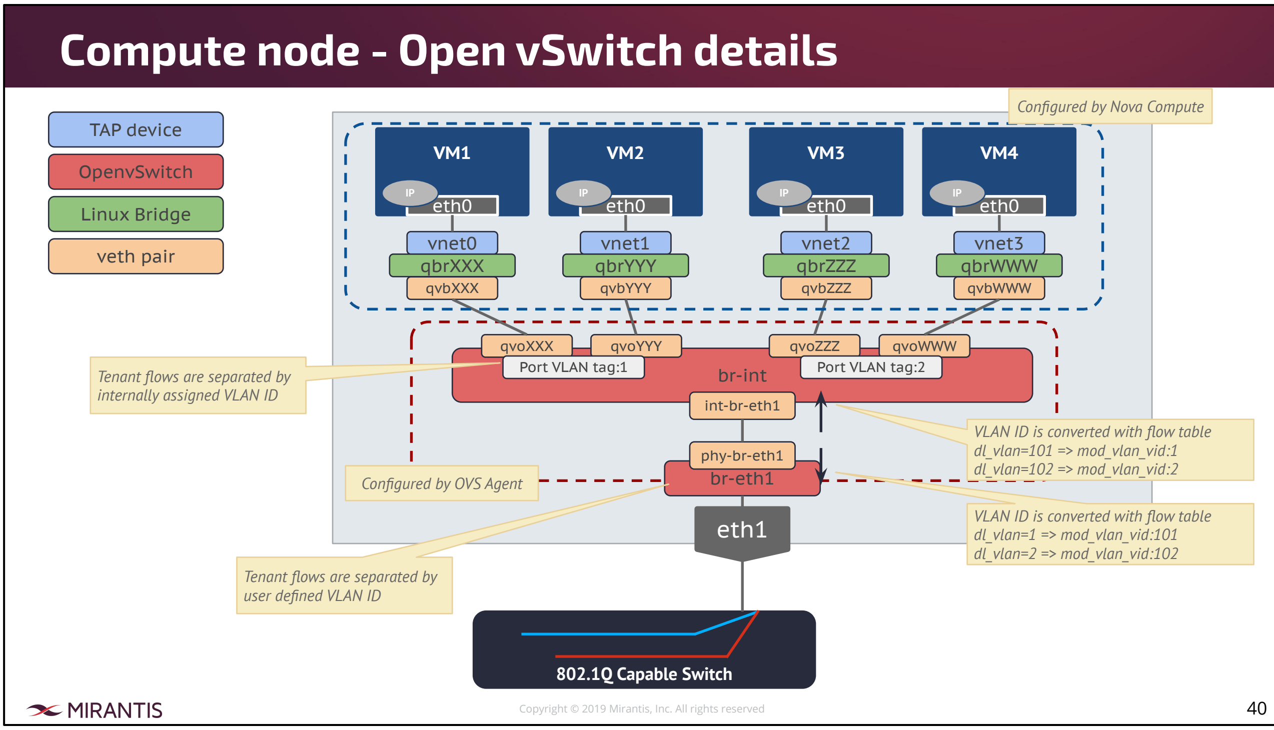
Two instances, in different tenants, on the same compute node, wish to communicate with one another (VM_1 to VM_3)

1. VM_1 sees that the destination of the packet resides on a different network (10.1.0.0), so it sends that packet to it's default gateway (10.0.0.1)
2. The packet arrives at br-int, then sent to network node
3. Network Node sees that the destination of the packet resides on a different network, so it consults it's routing table. There was an entry added to the routing table for the 10.1.0.0 network when VM_3 was launched, indicating that the entry point into the 10.1.0.0 network is through another router interface, so it sends the packet there.
4. The packet arrives at br-int, then sent back to compute node
5. Br-int broadcasts the ARP request to all connected devices. VM_3 responds with it's MAC address and is routed back to VM_1

OpenvSwitch – Under The Hood



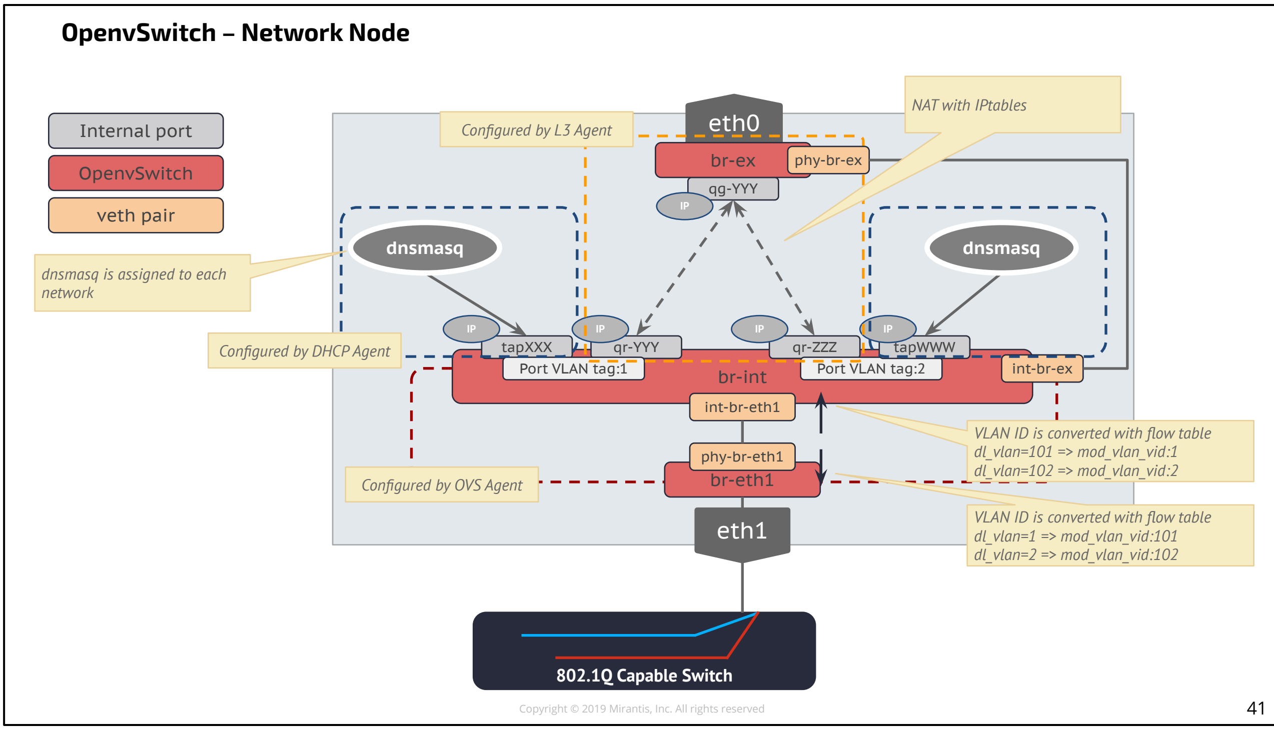
MIRANTIS



Neutron delegates the implementation detail and actual layout of the networking to its plugin. Let's look at how the Linux Bridge plugin manifests this logical network layout.
This is what is created by the Linux Bridge agent on a compute node.

In this diagram, we see three different types of virtual networking devices.

- TAP devices are created by virtualization platforms such as KVM and Xen on the host systems. A frame being sent from ethX in a virtual machine is received by a tap device on the host system. They typically show up as vnetX devices.
- A VLAN device gets associated with a VLAN ID, and takes care of tagging traffic to a VLAN as it leaves the system, and untags traffic as it is sent into the system. VLAN devices get attached to the physical NIC that communication is occurring on.
- A Linux bridge (QBR) acts as a simple software switch. Anything that comes into the bridge is sent to a different device that has been connected to the Bridge. Each VM has a unique Linux Bridge created for it to connect to the br-int.



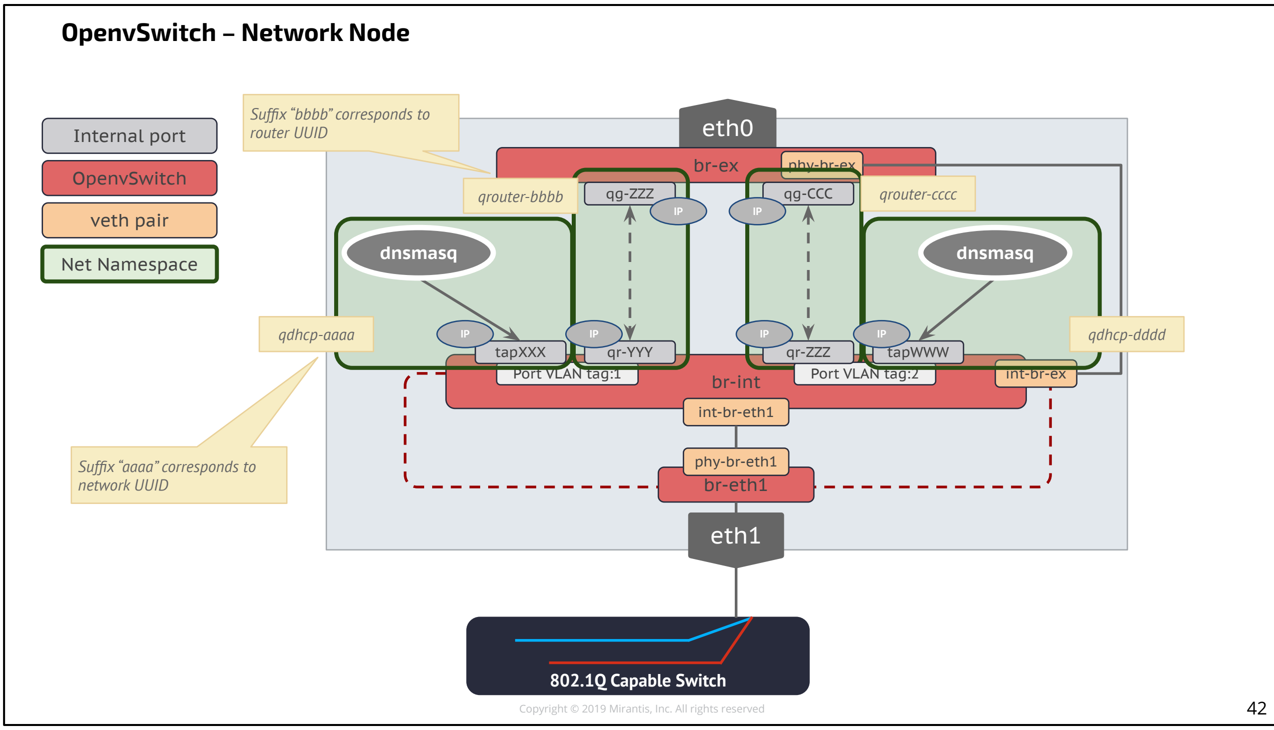
Neutron delegates the implementation detail and actual layout of the networking to its plugin. Let's look at how the Linux Bridge plugin manifests this logical network layout.
This is what is created by the Linux Bridge agent on a compute node.

In this diagram, we see three different types of virtual networking devices.

TAP devices are created by virtualization platforms such as KVM and Xen on the host systems. A frame being sent from ethX in a virtual machine is received by a tap device on the host system. They typically show up as vnetX devices.

A VLAN device gets associated with a VLAN ID, and takes care of tagging traffic to a VLAN as it leaves the system, and untags traffic as it is sent into the system. VLAN devices get attached to the physical NIC that communication is occurring on.

A Linux bridge acts as a simple software switch. Anything that comes into the bridge is sent to a different device that has been connected to the bridge.



Neutron delegates the implementation detail and actual layout of the networking to its plugin. Let's look at how the Linux Bridge plugin manifests this logical network layout.
This is what is created by the Linux Bridge agent on a compute node.

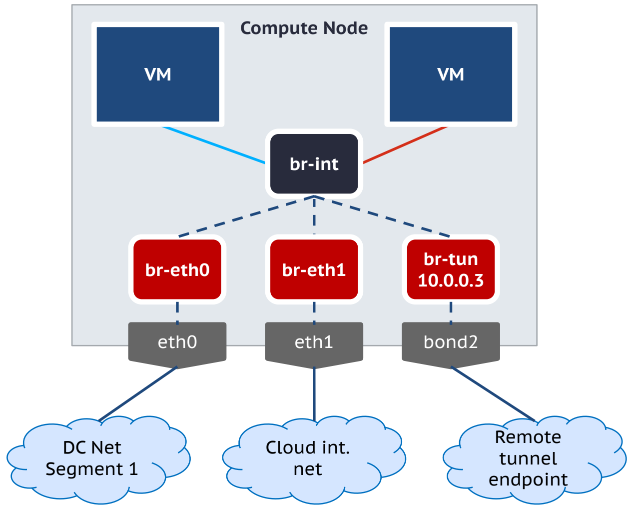
In this diagram, we see three different types of virtual networking devices.

TAP devices are created by virtualization platforms such as KVM and Xen on the host systems. A frame being sent from ethX in a virtual machine is received by a tap device on the host system. They typically show up as vnetX devices.

A VLAN device gets associated with a VLAN ID, and takes care of tagging traffic to a VLAN as it leaves the system, and untags traffic as it is sent into the system. VLAN devices get attached to the physical NIC that communication is occurring on.

A Linux bridge acts as a simple software switch. Anything that comes into the bridge is sent to a different device that has been connected to the bridge.

Integration With Multiple Datacenter Networks

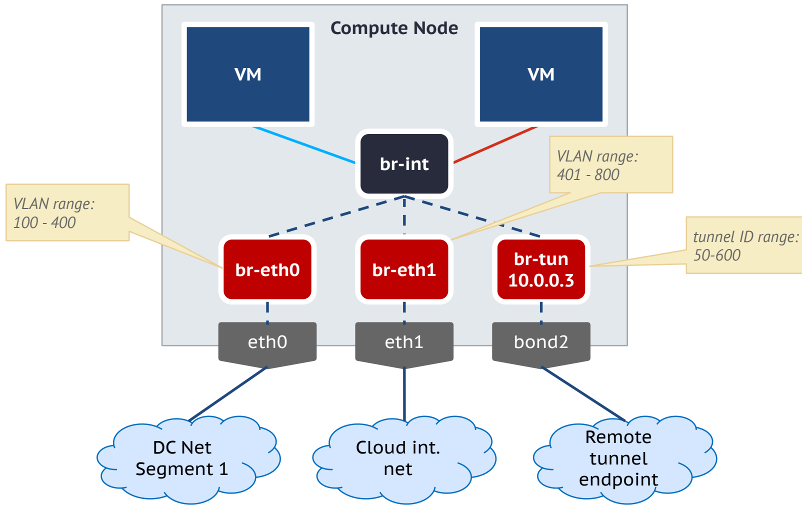


Dedicated per-NIC bridge

MIRANTIS

This requirement is satisfied by capability of neutron to use multiple bridges leading to different physical devices.

Integration With Multiple Datacenter Networks



VLAN ranges are mapped to per-NIC bridges

How to deal with multiple bridges:

If VLANs are in use, then ranges of them can be mapped to different bridges

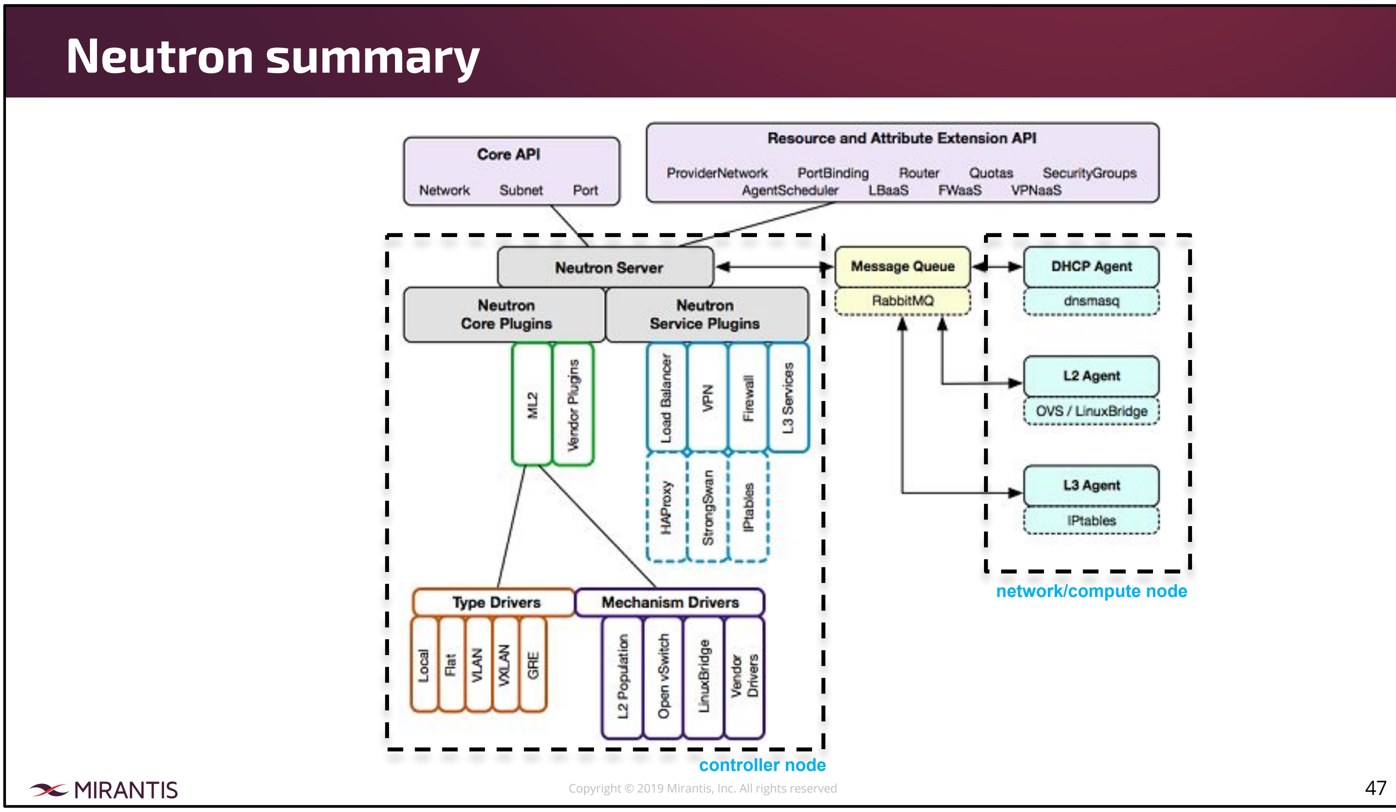
In case of Gre we can specify tunnel ranges.

Summary

Summary

You should now be able to:

- Understand Neutron concepts and plugin architecture
  - Modular Layer (ML2) *type* and *mechanism* drivers
  - Agents: IPAM, L3, DHCP
- Understand Open vSwitch network implementation
- Explain what network namespaces are and why they are important
- Describe Octavia LBaaS v2
- Use the CLI to implement a load balancer solution
- Explain what a floating IP is and how to allocate/associate
- Describe NAT and packet filtering, including how Neutron security groups apply



This slide summarizes the Neutron lecture.

- Neutron-server services requests for :
  - Core API services: Network, subnet, port
    - Type drivers discussed:
      - Local
      - Flat
      - VLAN
      - VXLAN
      - GRE (Generic Routing Encapsulation)
    - Mechanism drivers discussed:
      - Linux Bridge
      - Open vSwitch
      - Vendor plugins
  - Service plugins: Router, Load balancer, firewall
- DHCP agent uses dnsmasq
- L3 agent uses IP tables for SNAT and DNAT

Lab exercises

Lab 8: External Network Connectivity to VM

Lab 9: Neutron Under the Hood

MIRANTIS

Reference

Optional Material

The slides in this section are provided as reference material. They will not be presented by the instructor.

DVR Traffic Flows

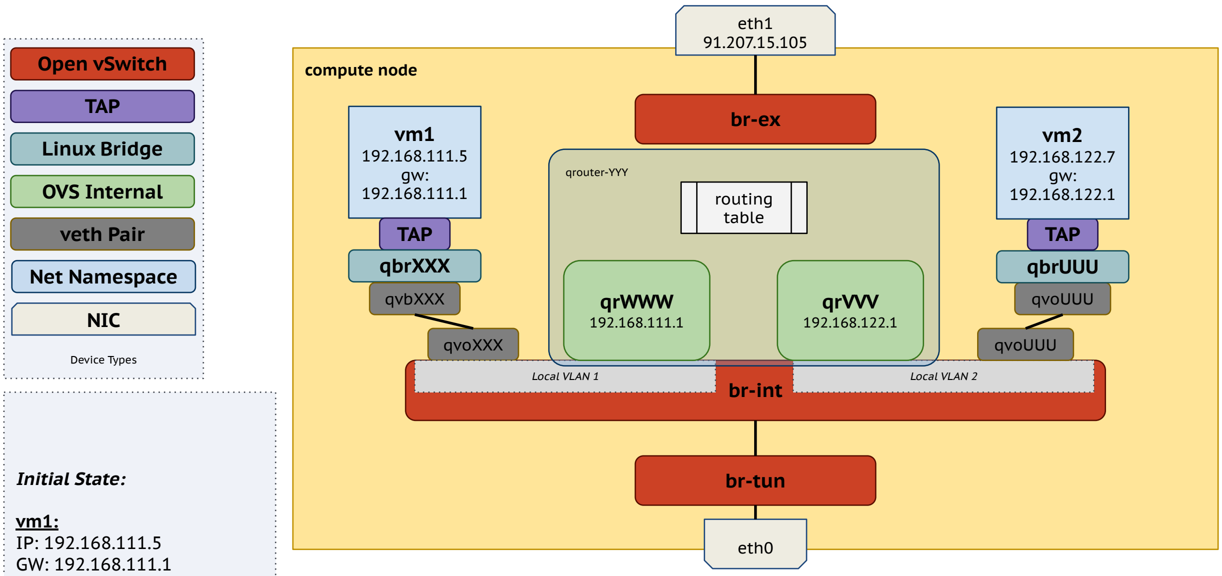
- Key Assumptions:
  - Neutron DVR
  - ML2 Plugin with Open vSwitch and L2 Population mechanisms
  - VXLAN isolation
  - One project with two networks

Flow 1

Instance without Floating IP connects outside

http://docs.openstack.org/mitaka/networking-guide/scenario-dvr-ovs.html

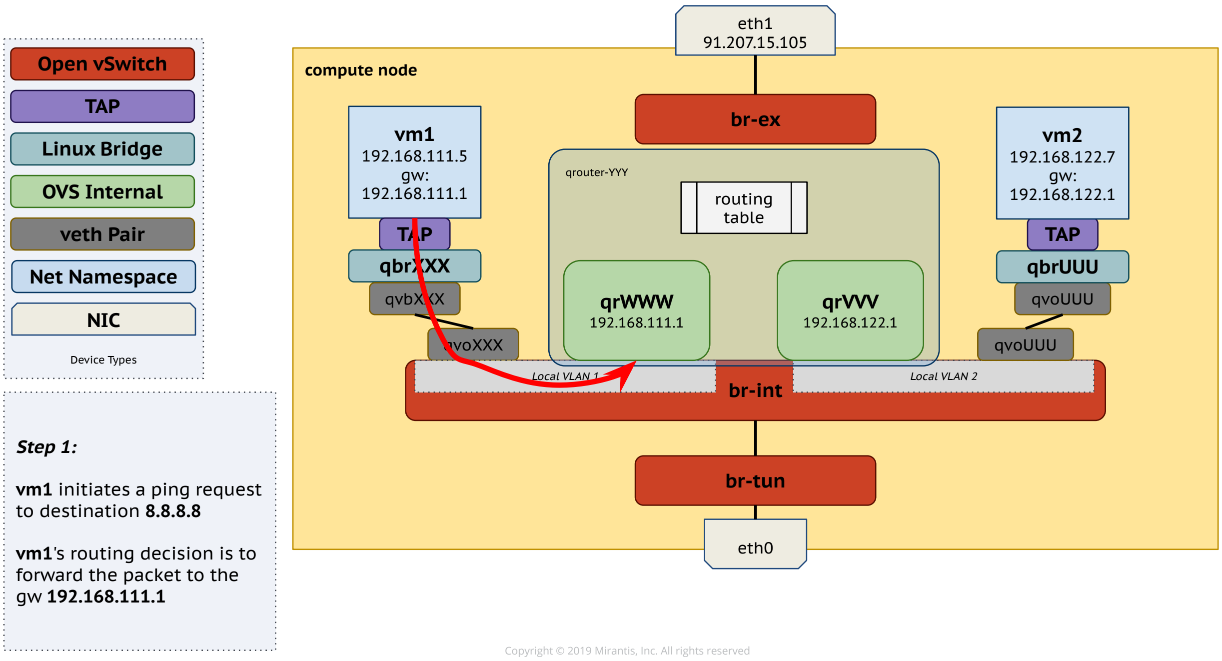
Flow 1: DVR Instance North - South SNAT Traffic - Compute Node



Initial State:

vm1:
IP: 192.168.111.5
GW: 192.168.111.1

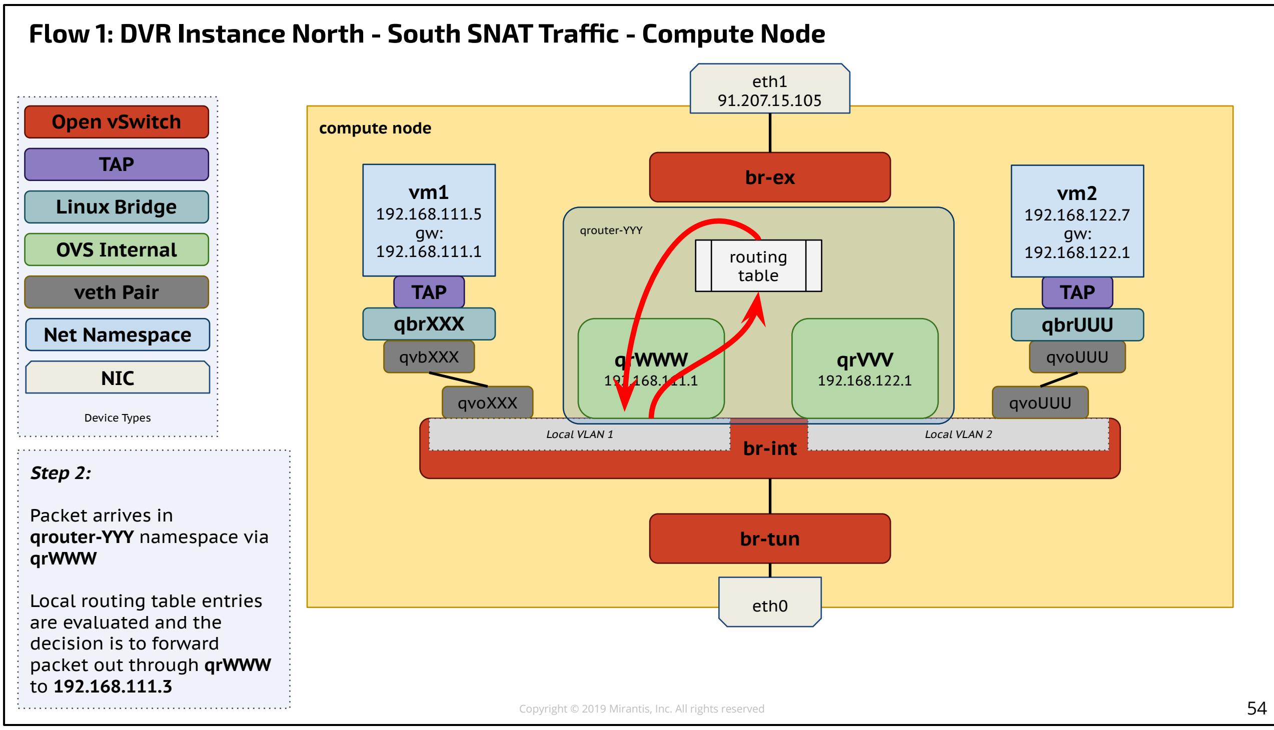
Flow 1: DVR Instance North - South SNAT Traffic - Compute Node



Step 1:

vm1 initiates a ping request to destination **8.8.8.8**

vm1's routing decision is to forward the packet to the gw **192.168.111.1**



There are several routing tables that exist within the router namespace on a compute node. In a topology such as what we have here, 3 tables will exist.

1 table is used for east-west communication.
1 table is used for SNAT communication from 192.168.111.0/24
1 table is used for SNAT communication from 192.168.122.0/24

The way a specific routing table is used is through a rule.

Sample output:

```
root@compute1 # ip netns exec qrouter-YYY bash
root@compute1 # ip rule list
0:          from all lookup local
32766:      from all lookup main
32767:      from all lookup default
3232263937: from 192.168.111.1/24 lookup 3232263937
3232266753: from 192.168.122.1/24 lookup 3232263937
```

Rules are evaluated in order until a match is found. First the main table is evaluated:

```
root@compute1 # ip route show table main
192.168.111.0/24 dev qr-WWW …
192.168.122.0/24 dev qr-VVV ....
```

So, nothing there since our destination here is 8.8.8.8

Next, the default table is evaluated:

```
root@compute1 # ip route show table default
<no output>
```

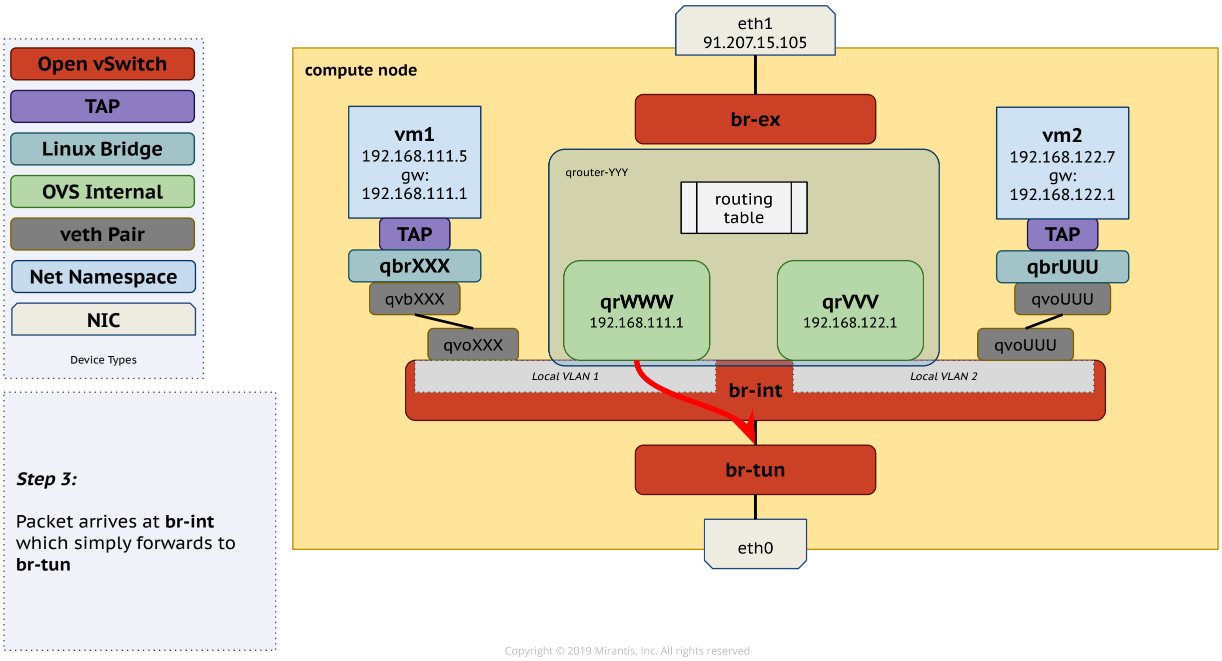
Well, ok then.

Next, the `3232263937` table is evaluated:

```
root@compute1 # ip route show table 3232263937
default via 192.168.111.3 dev qr-WWW
```

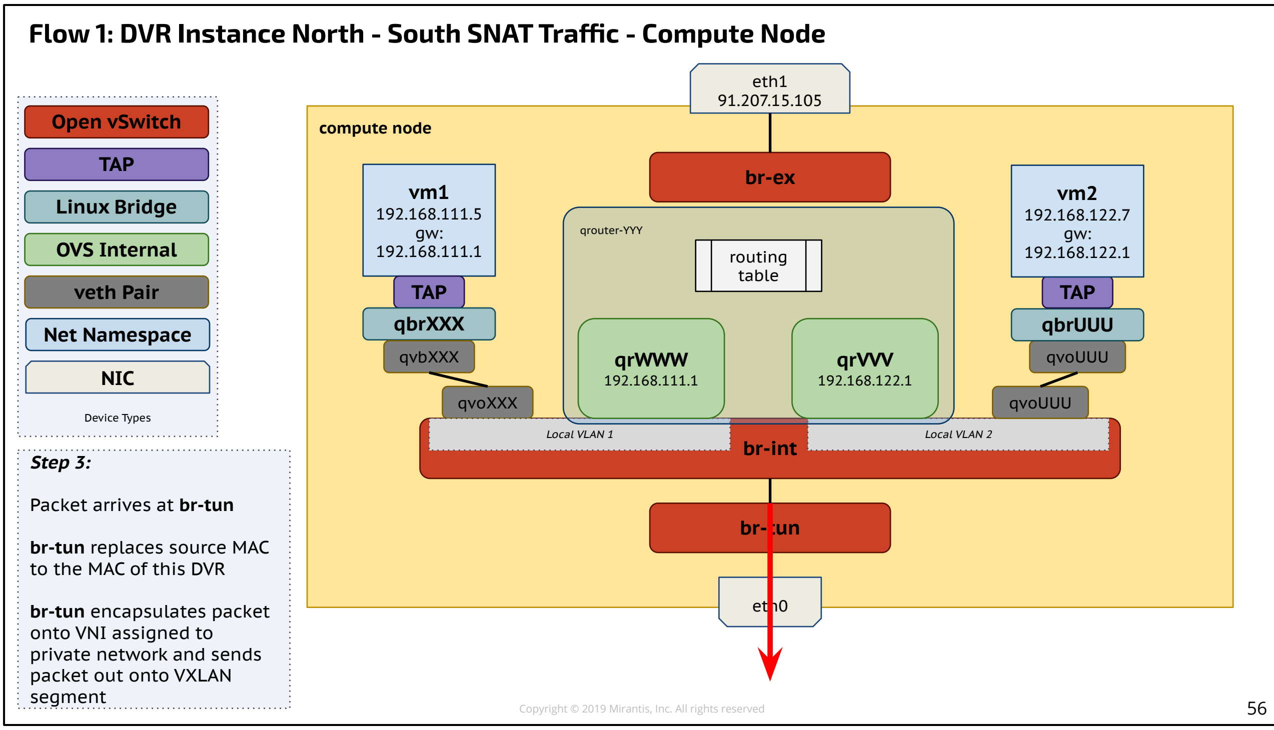
So where does 192.168.111.3 live? On the network node, inside a snat-XYZ namespace.

Flow 1: DVR Instance North - South SNAT Traffic - Compute Node



Step 3:

Packet arrives at **br-int** which simply forwards to **br-tun**



I have not yet found a way to determine what MAC address is associated with a DVR node aside from looking in neutron's DB.

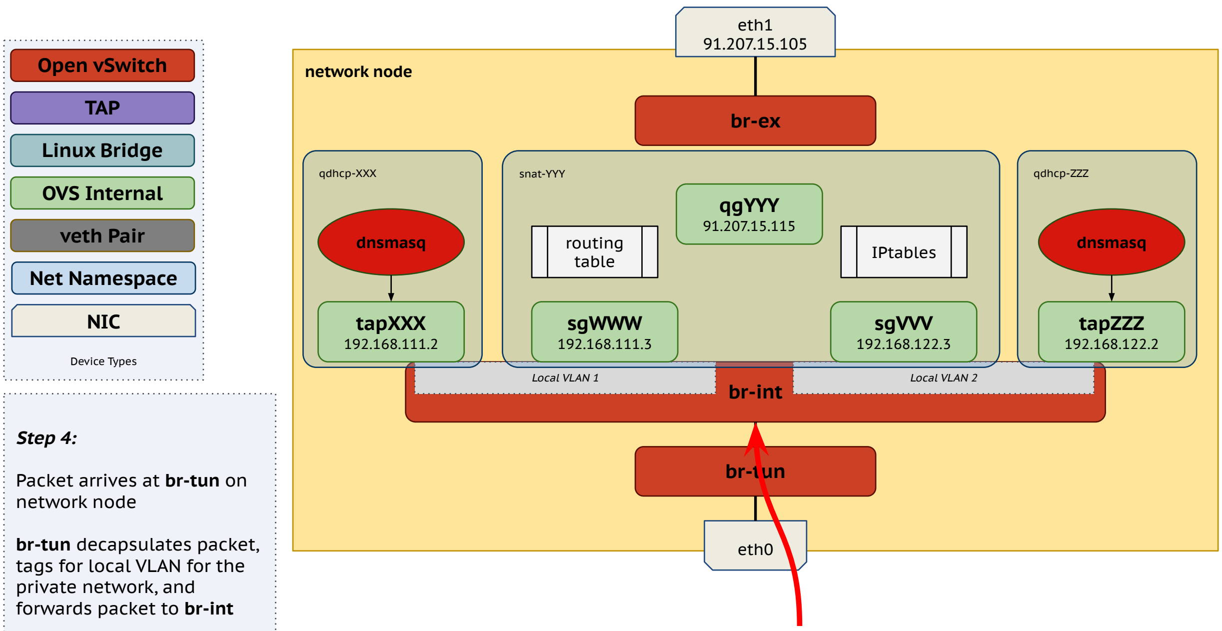
```
select * from neutron.dvr_host_macs;
+-----------------------------------+-------------------------------
---+
| host                              | mac_address
      |
+-----------------------------------+-------------------------------
---+
| compute1                          | fa:16:3f:c7:eb:e5
      |
...
+-----------------------------------+-------------------------------
---+
```

There is a flow rule on br-tun that will perform the source MAC replacement. Snippet from that flow rule below:

```
ovs-ofctl dump-flows br-tun | grep fa:16:3f:c7:eb:e5

...  dl_vlan=1,dl_src=<qrWWW_mac> actions=mod_dl_src:fa:16:3f:c7:eb:e5
```

Flow 1: DVR Instance North - South SNAT Traffic - Network Node

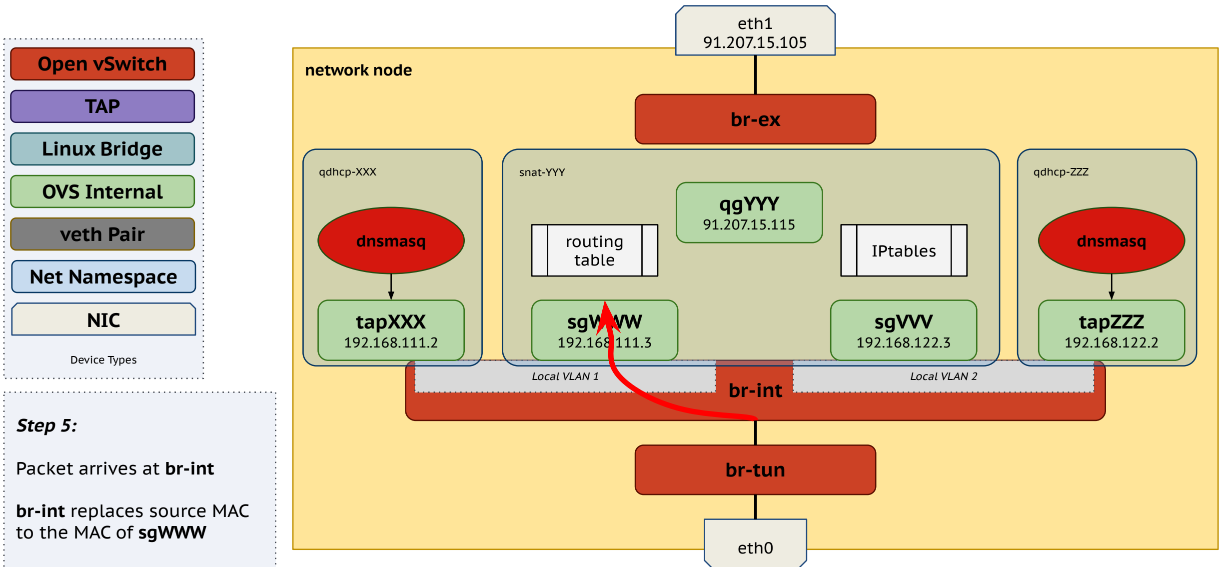


Step 4:

Packet arrives at **br-tun** on network node

br-tun decapsulates packet, tags for local VLAN for the private network, and forwards packet to **br-int**

Flow 1: DVR Instance North - South SNAT Traffic - Network Node



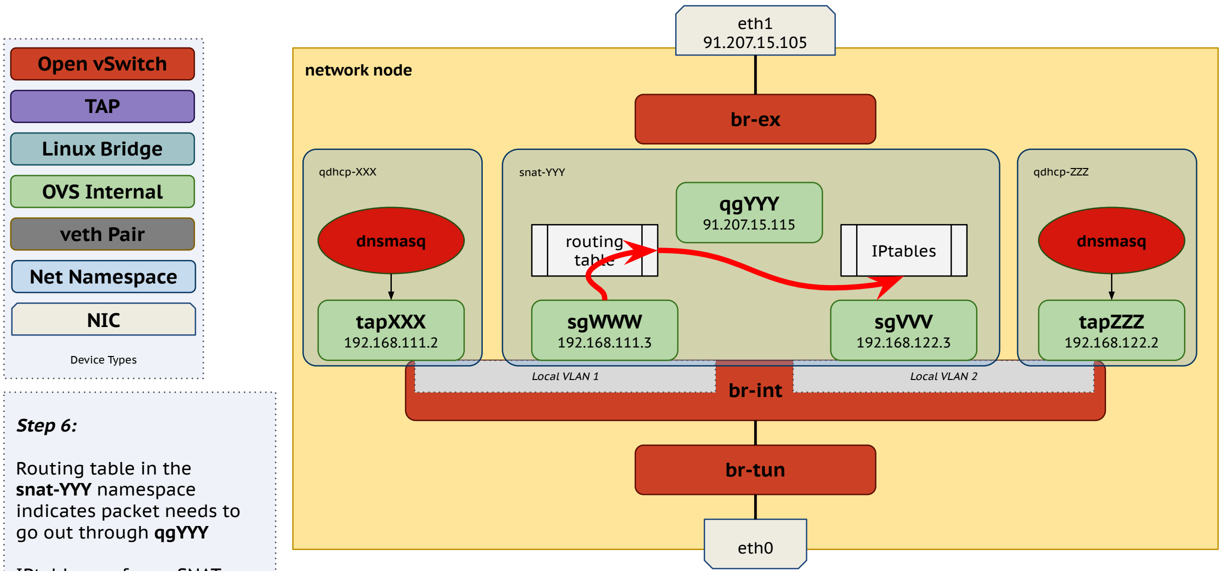
Step 5:

Packet arrives at **br-int**

br-int replaces source MAC to the MAC of **sgWWW**

br-int forwards the packet to **sgWWW** in the **snat-YYY** namespace

Flow 1: DVR Instance North - South SNAT Traffic - Network Node

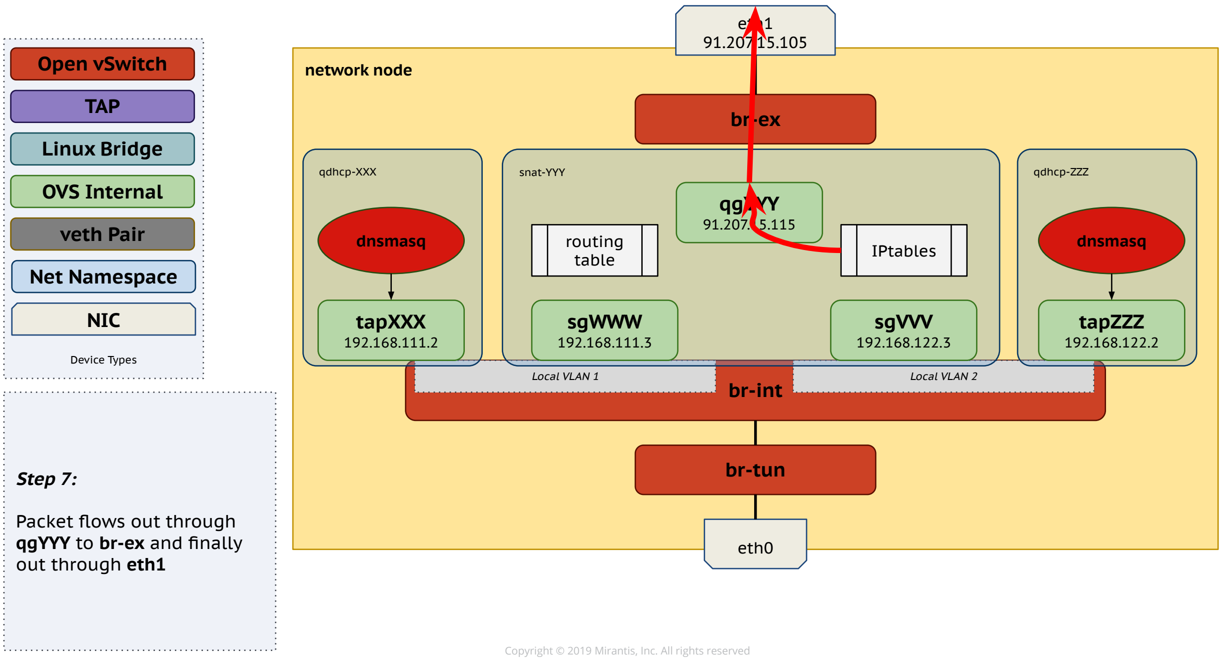


Step 6:

Routing table in the **snat-YYY** namespace indicates packet needs to go out through **qgYYY**

IPtables performs SNAT modifying source IP to 91.207.15.115

Flow 1: DVR Instance North - South SNAT Traffic - Network Node



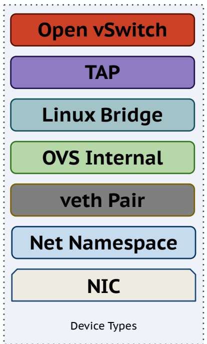
Step 7:

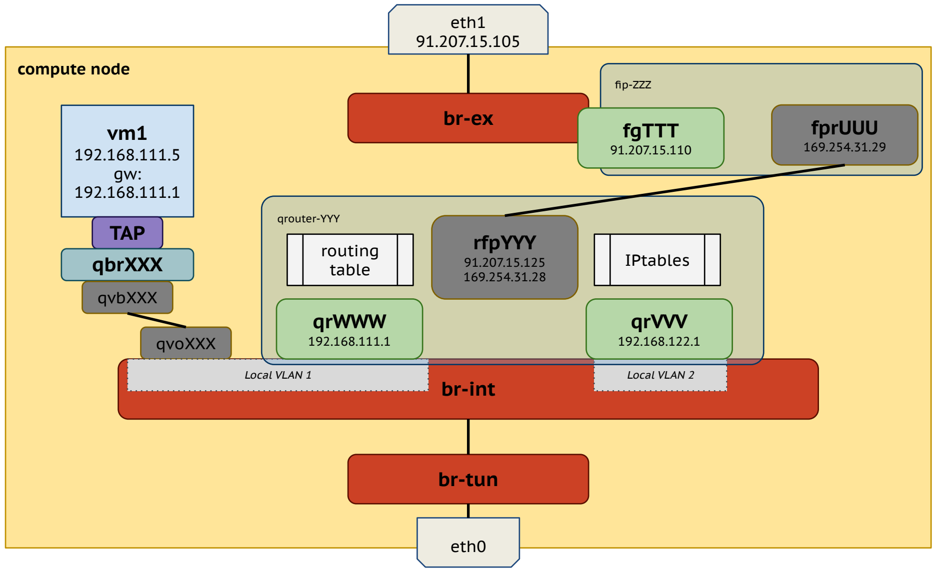
Packet flows out through **qgYYY** to **br-ex** and finally out through **eth1**

Flow 2

Instance with Floating IP connects outside

Flow 2: DVR Instance North - South DNAT Traffic - Compute Node

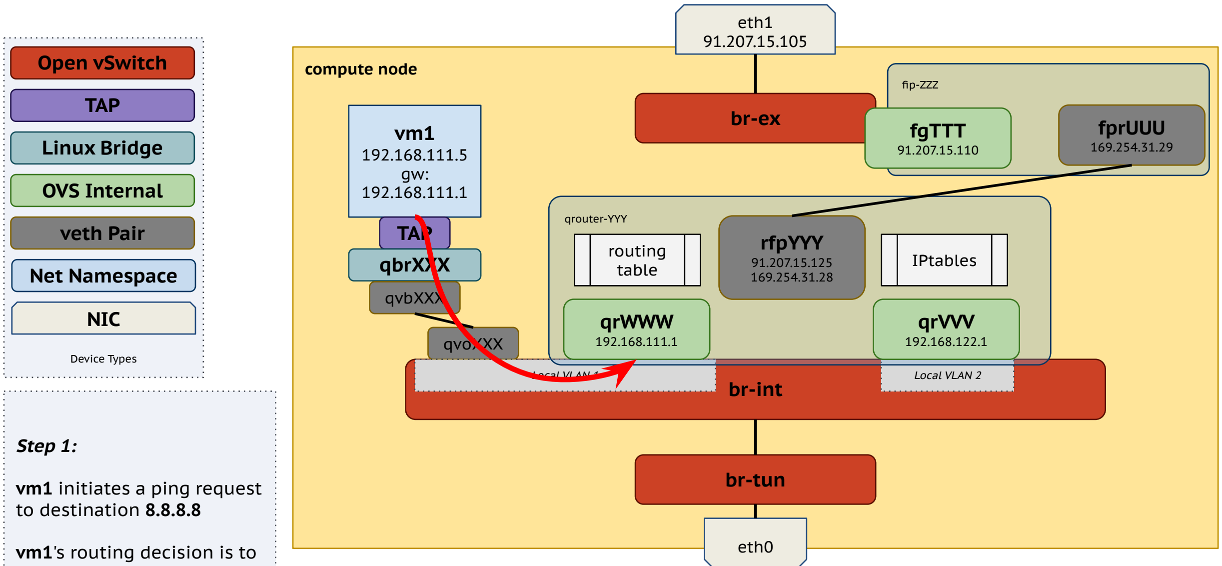




Initial State:

<u>**vm1:**</u>
IP: 192.168.111.5
GW: 192.168.111.1
FIP: 91.207.15.125

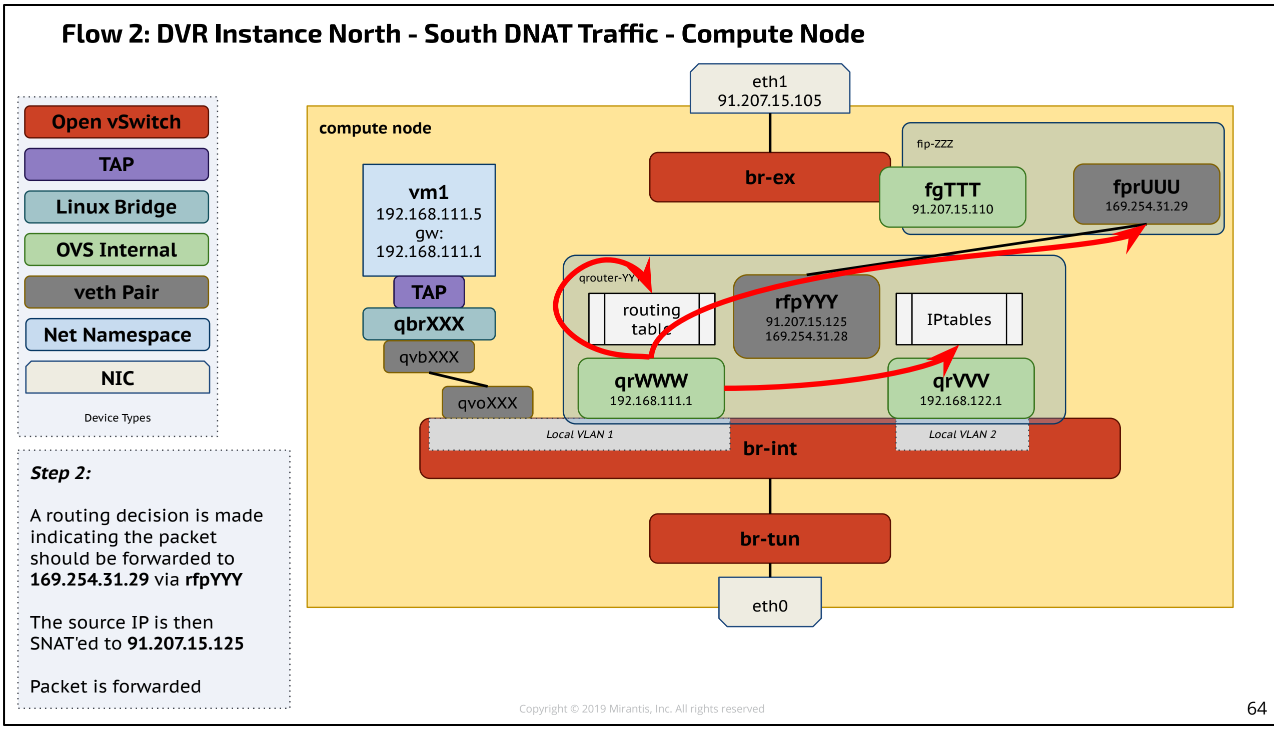
Flow 2: DVR Instance North - South DNAT Traffic - Compute Node



Step 1:

vm1 initiates a ping request to destination **8.8.8.8**

vm1's routing decision is to forward the packet to its gw **192.168.111.1**



There are two primary configuration details done within the qrouter namespace to allow for this.

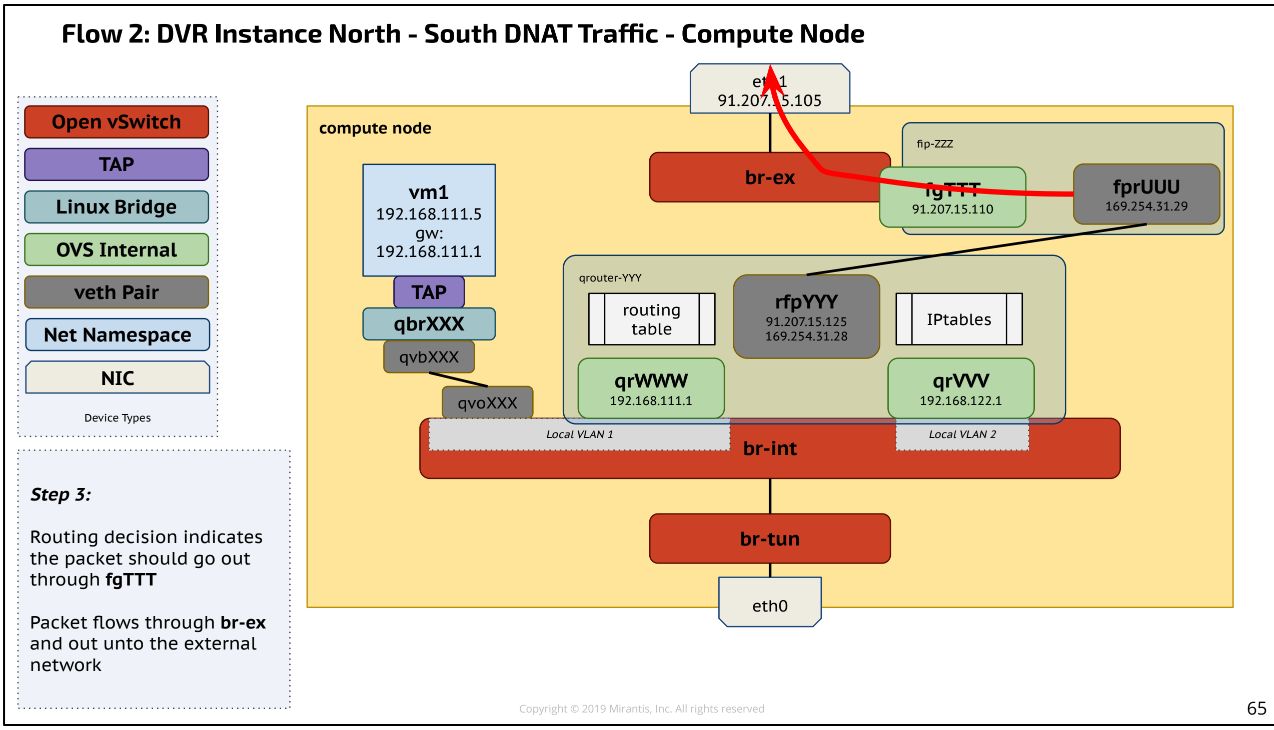
1. A routing table that routes all traffic coming from the fixed IP into the fip namespace.

```
root@compute1 # ip rule list
32768: from 192.168.111.5 lookup 16

root@compute1 # ip route show table 16
default via 169.254.31.29 dev rfp-YYY
```

2. A rule in IPtables to perform SNAT to modify source IP from fixed to floating

```
Chain neutron-l3-agent-float-snat
...
target      source              destination         
SNAT        192.168.111.5       0.0.0.0/0         to:
91.207.15.125
```



There is one primary configuration detail done within the fip namespace to allow for this.

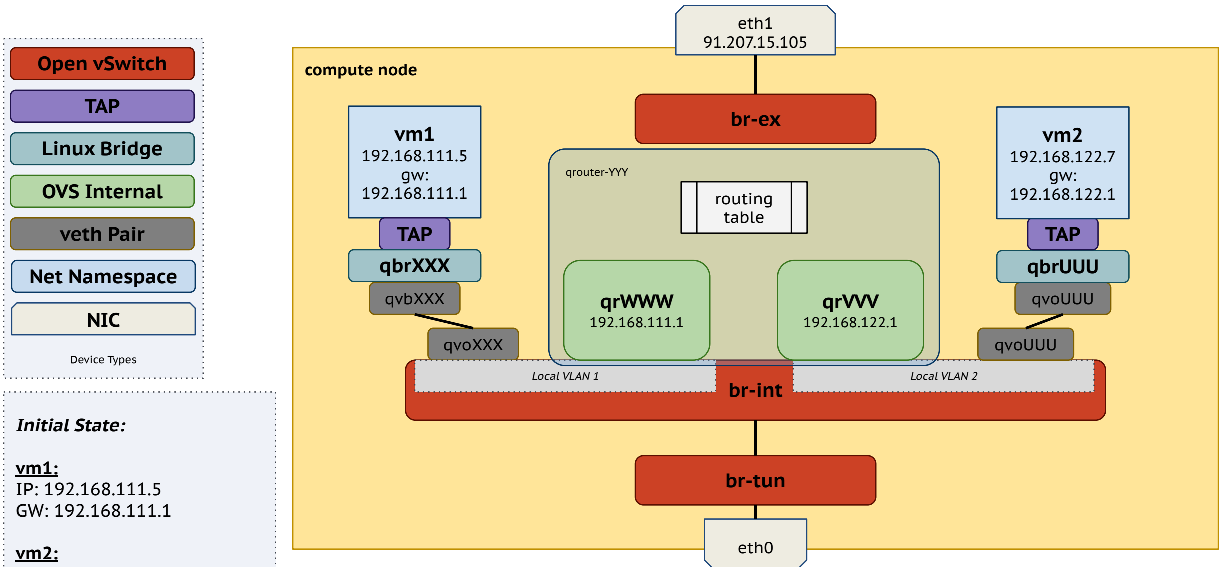
1. Default route in the fip namespace

```
root@compute1 # ip route
default via 91.207.15.1 dev fgTTT
```

Flow 3

Two instances on same compute node

Flow 3: DVR Instance East - West Traffic - Compute Node

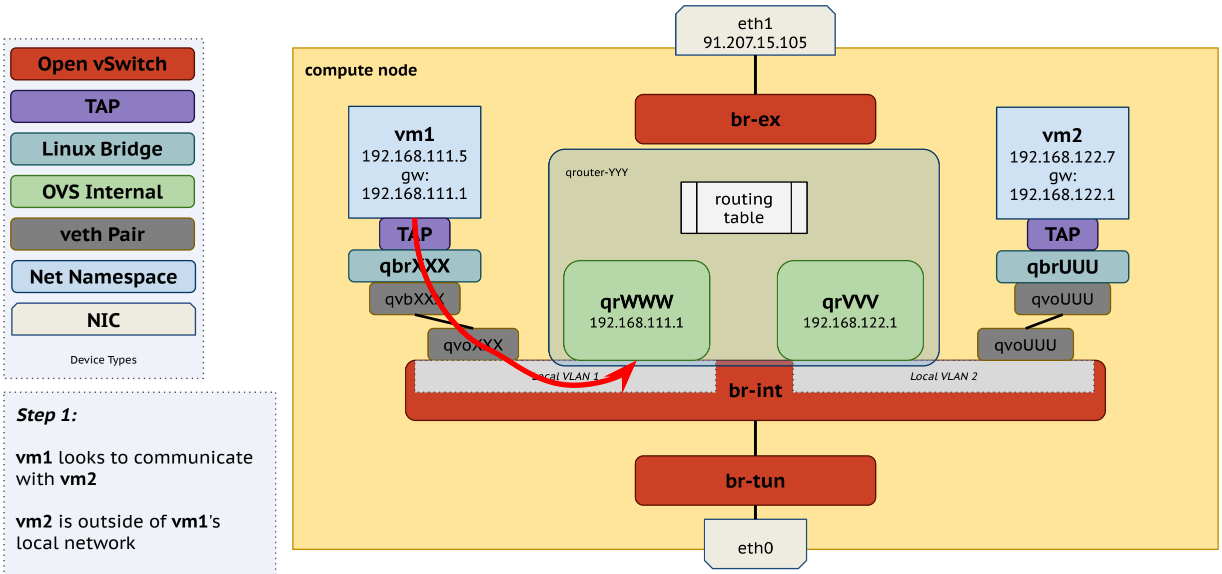


Initial State:

vm1:
IP: 192.168.111.5
GW: 192.168.111.1

vm2:
IP: 192.168.122.7
GW: 192.168.122.1

Flow 3: DVR Instance East - West Traffic - Compute Node



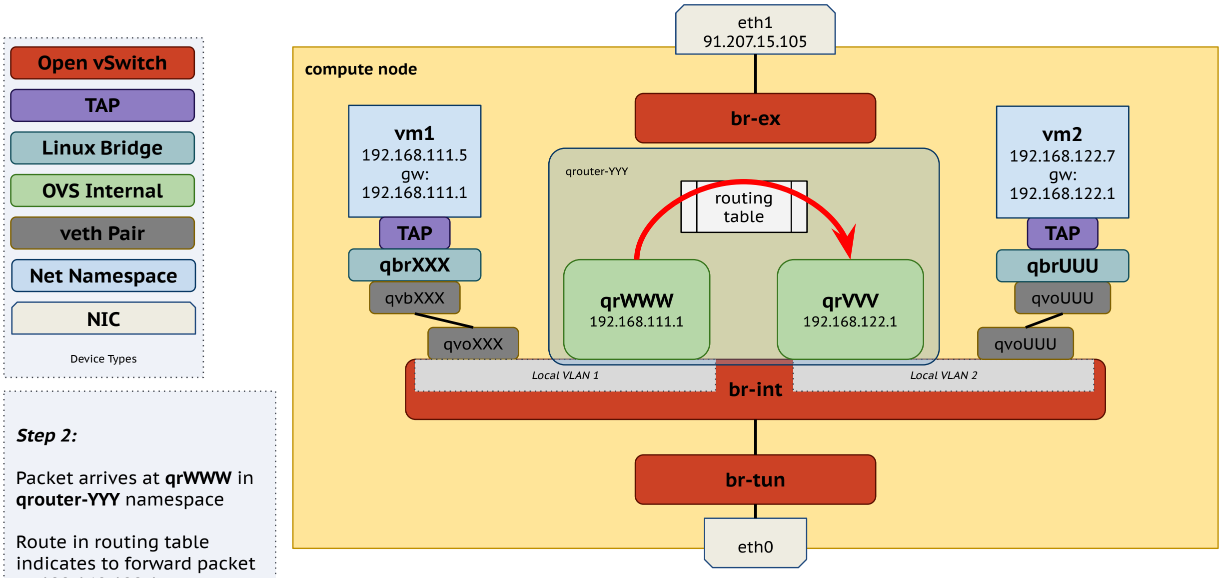
Step 1:

vm1 looks to communicate with **vm2**

vm2 is outside of **vm1**'s local network

vm1 forwards packet to gw **192.168.111.1**

Flow 3: DVR Instance East - West Traffic - Compute Node

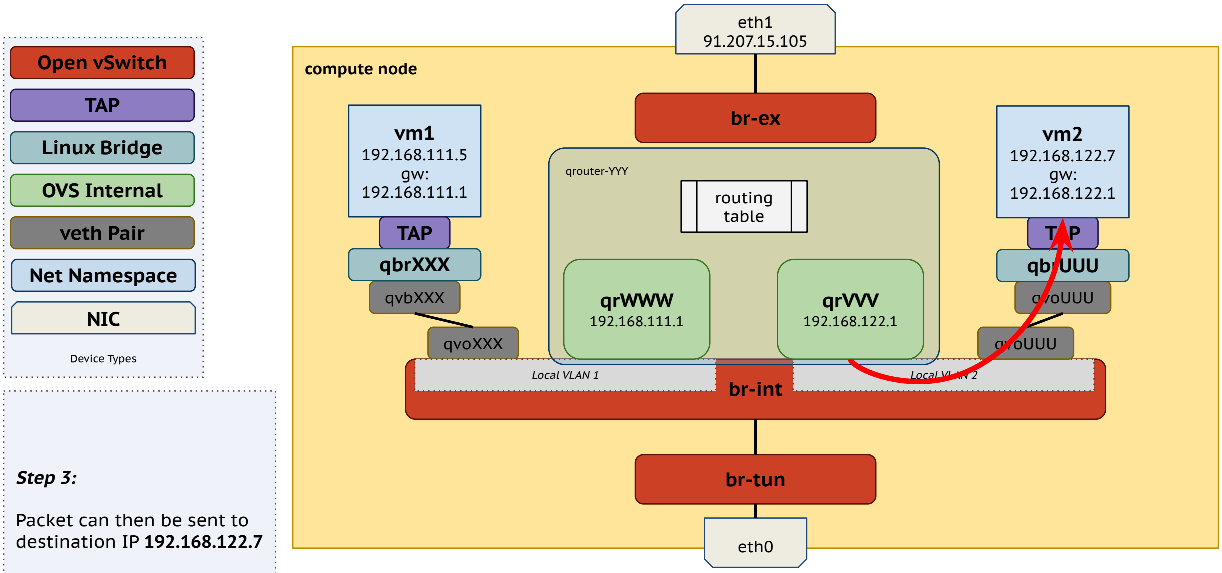


Step 2:

Packet arrives at **qrWWW** in **qrouter-YYY** namespace

Route in routing table indicates to forward packet to **192.168.122.1**

Flow 3: DVR Instance East - West Traffic - Compute Node



Step 3:

Packet can then be sent to destination IP **192.168.122.7**

Flow 4

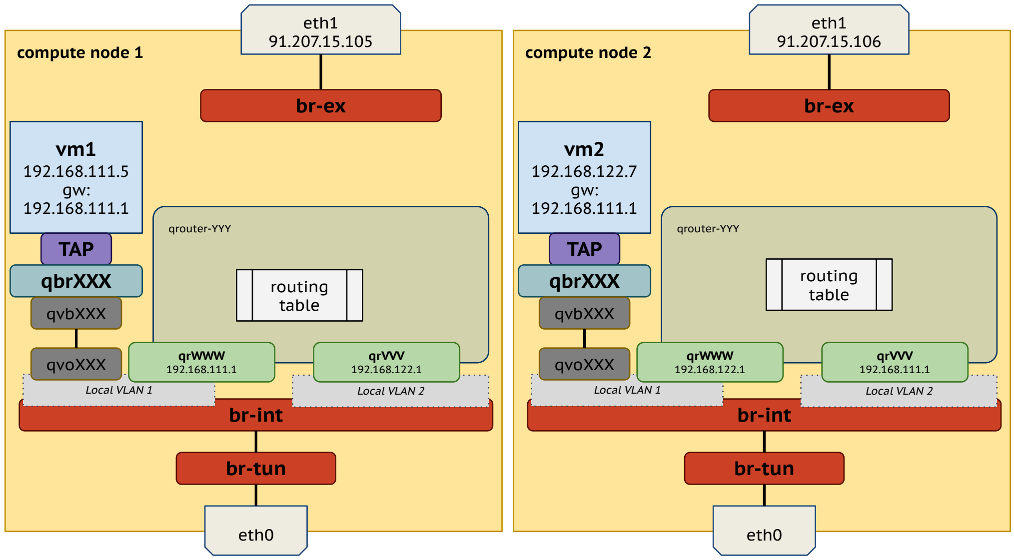
Two instances on different compute nodes

Flow 4: DVR Instance East - West Traffic - Compute Node

Initial State:

vm1:
IP: 192.168.111.5
GW: 192.168.111.1

vm2:
IP: 192.168.122.7
GW: 192.168.122.1



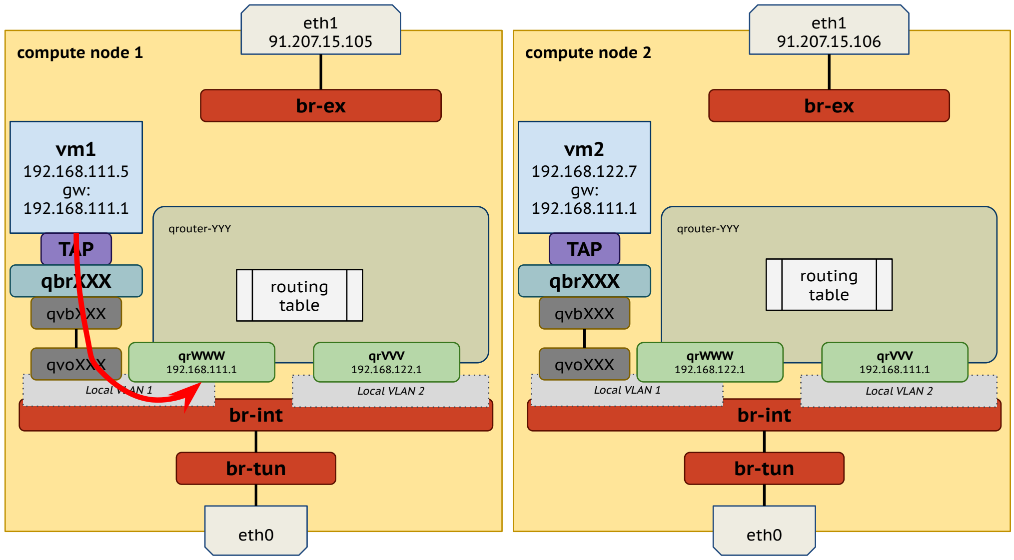
Flow 4: DVR Instance East - West Traffic - Compute Node

Step 1:

vm1 looks to communicate with **vm2**

vm2 is outside of **vm1**'s local network

vm1 forwards packet to gw **192.168.111.1**

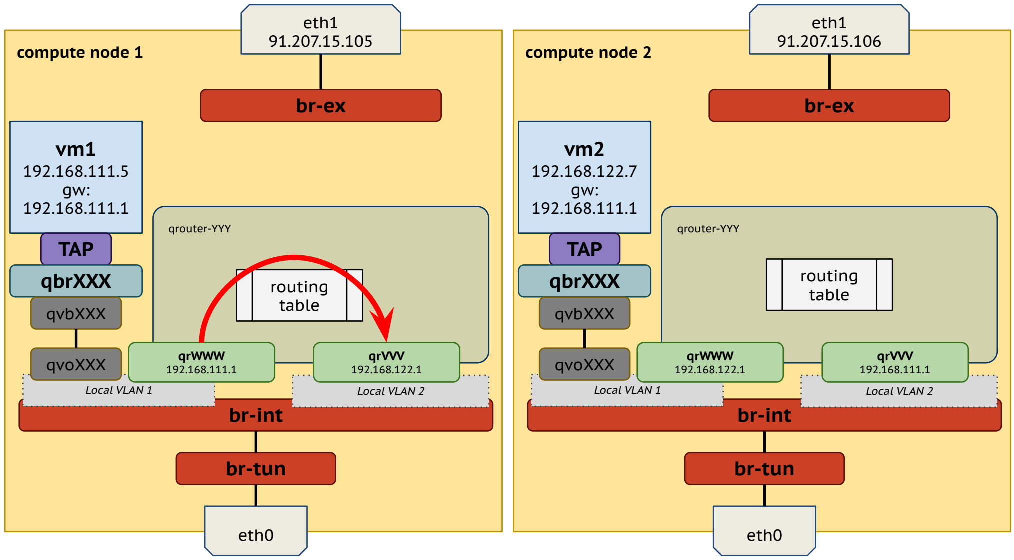


Flow 4: DVR Instance East - West Traffic - Compute Node

Step 2:

Packet arrives at **qrWWW** in **qrouter-YYY** namespace

Route in routing table indicates to forward packet to **192.168.122.1**



Flow 4: DVR Instance East - West Traffic - Compute Node

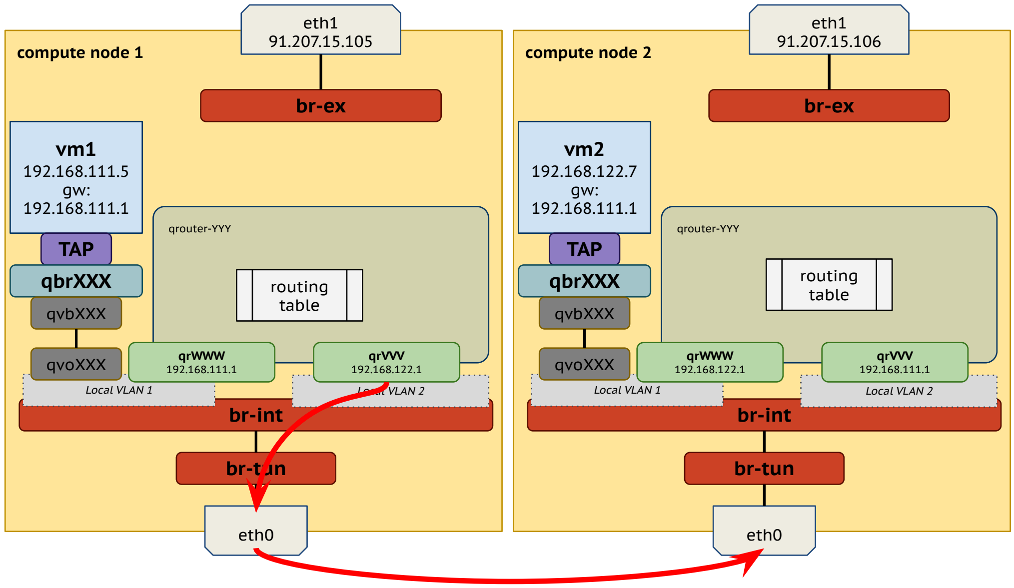
Step 3:

qrVVV forwards packet to **br-int**

br-int forwards packet to **br-tun**

br-tun replaces source MAC to **qrouter-YYY** MAC

br-tun encapsulates packet onto VNI assigned to private network and sends packet out onto VXLAN segment to **compute node 2**



Flow 4: DVR Instance East - West Traffic - Compute Node

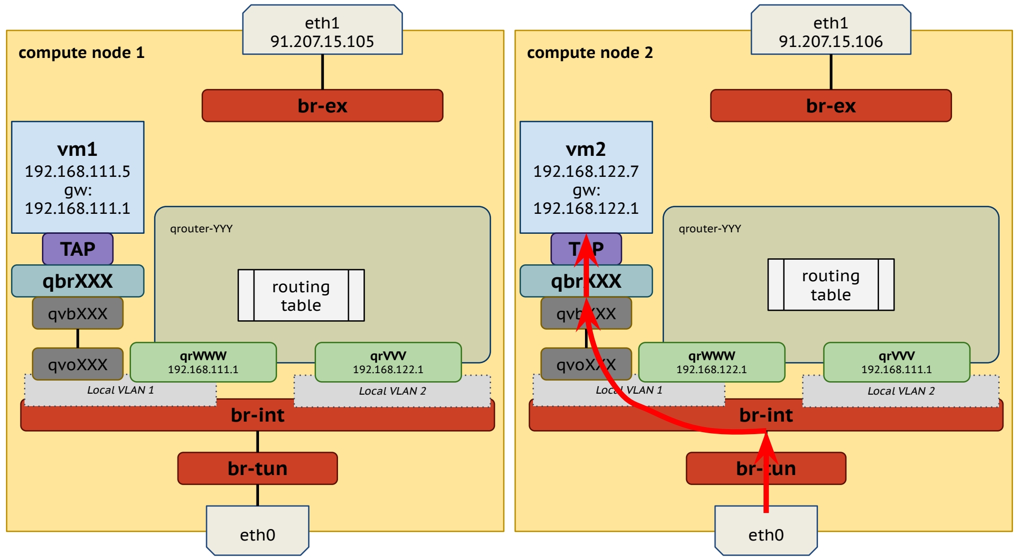
Step 4:

br-tun decapsulates packet, tags for local VLAN for the private network, and forwards packet to **br-int**

br-int replaces source MAC with MAC of **qrWWW**

br-int forwards packet to **qbrXXX**

qbrXXX forwards packet to **vm2**

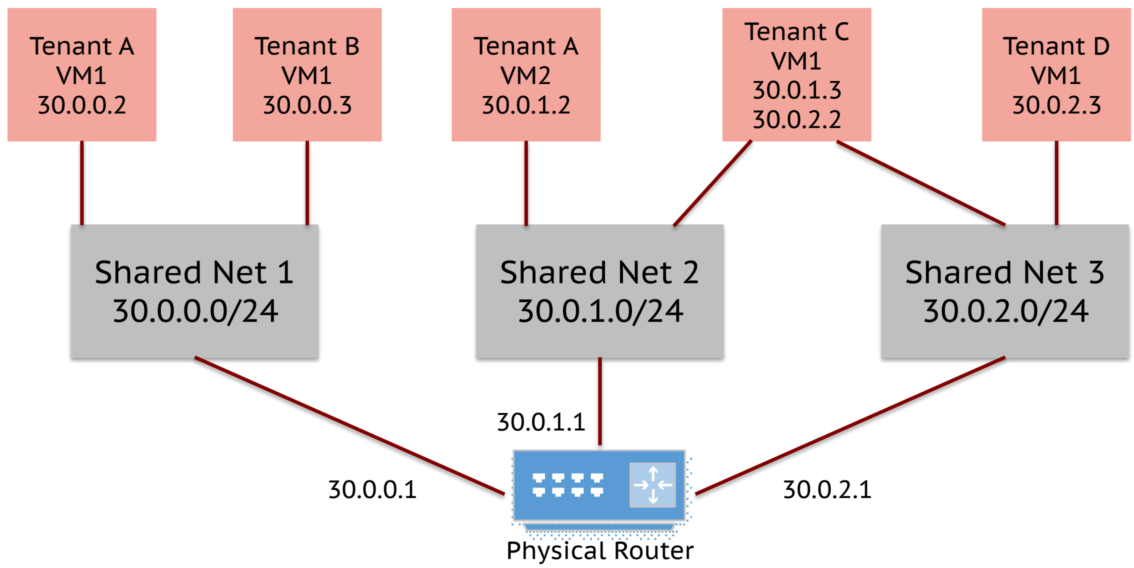


Neutron Network Topologies

Neutron: Single Flat Network

- Model 1:
  - Single NIC
  - Receives a fixed IP address from the subnet(s) associated
  - Floating IPs are not supported

Neutron: Single Flat Network



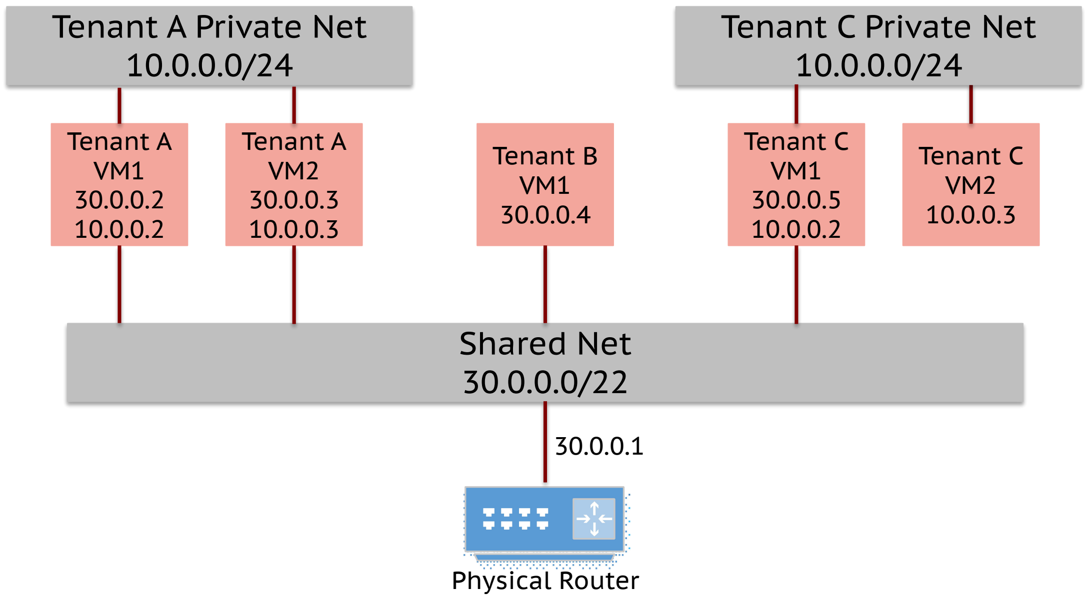
Neutron: Multiple Flat Networks

- Model 2:
  - Same as Model 1 but clients can choose which network to plug into.

Neutron: Mixed Flat & Private Networks

- Model 3:
  - Enables multi-tier topologies.
  - VMs can use multiple NICs.
  - A VM can act as a gateway providing routing, NAT, or LB.

Neutron: Mixed Flat & Private Networks



Neutron: Per-Tenant Router With Private Networks

- Model 4:
  - Overlapping IP address.
  - Separate Networks.
  - Abilities for tenant to create their own routers.

Neutron: Per-Tenant Router With Private Networks