Software Defined Networking & OpenStack

SAGE @ GUUG HH / Martin Gerhard Loschwitz

October 10, 2013

Version 1.0


hastexo!

COURSE OUTLINE

Software Defined Networking - an Introduction

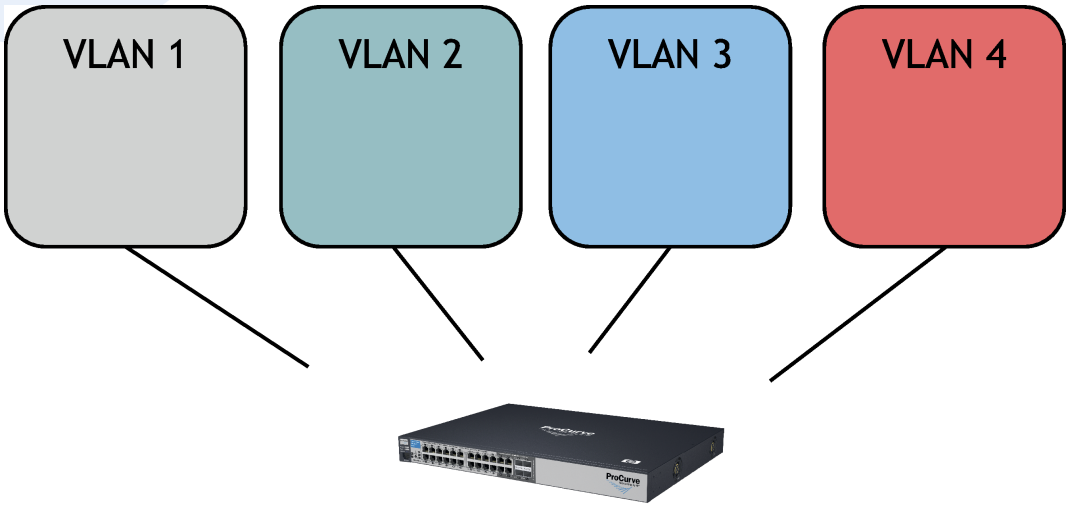
- So what is SDN, and why do we need it?

From Wikipedia: Software defined networking (SDN) is an emerging architecture for computer networking. SDN separates the control plane from the data plane in network switches and routers. Under SDN, the control plane is implemented in software in servers separate from the network equipment and the data plane is implemented in commodity network equipment

SDN PRIMER

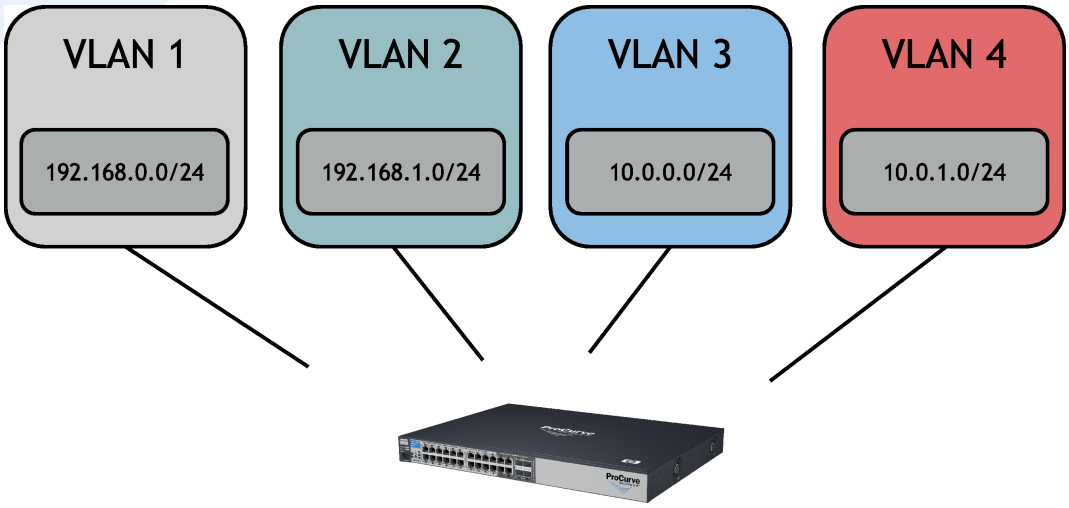
To understand what SDN really is, let's take a look at how networking usually works in DCs and how it used to work in OpenStack-based clouds up to now. First, here's the very traditional model: There's a switch that takes care of connecting individual machines within a cabinet

SDN PRIMER (CONT.)



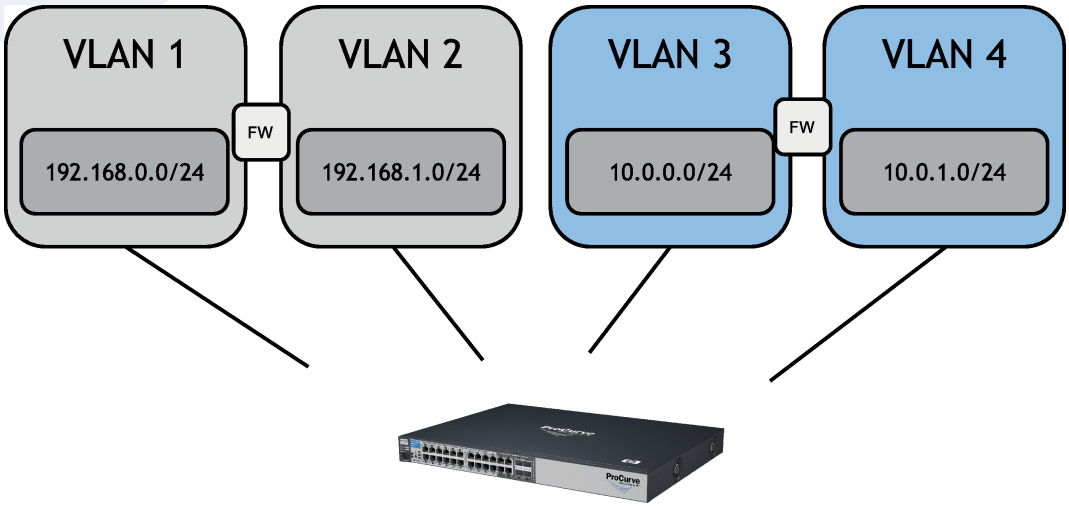
That switch has VLANs configured for individual customers (1 VLAN per customer)

SDN PRIMER (CONT.)



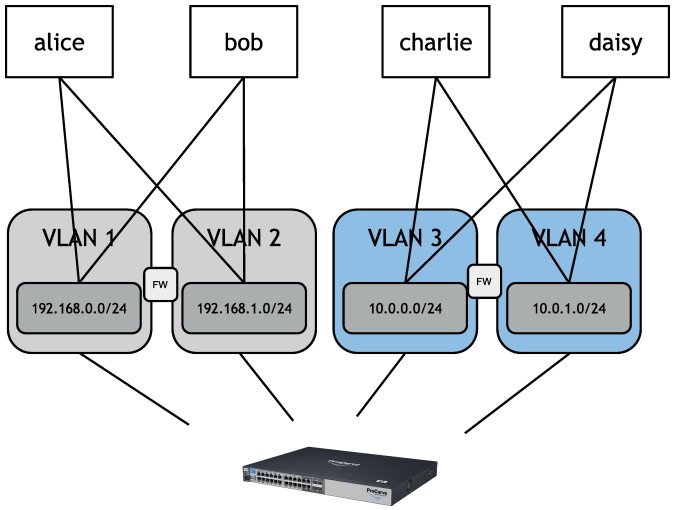
And within these VLANs, we have Subnets, typically one per VLAN

SDN PRIMER (CONT.)



What's also possible is that we have several VLANs owned by the same customer And at some certain level within the network, these different VLANs will be routed into one network space, featuring access control this would typically happen on firewalls

SDN PRIMER (CONT.)



In fact, these networks are where our actual machines plug into This is a simplified view, in fact, every server would need to have a port in every VLAN for this to work properly Typically, we would have some customer servers that belong to customer A, so these plug into the first two VLANs and there might other servers that belong to another user and thus plug into the other VLAN

CONVENTIONAL NETWORKING

- Static Network Design
- No need to scale-out
- Predictable Traffic patterns

Based on this, it's safe to make the following statement: Conventional networking structures come along with some defined assumptions on the state of the network and the state of the computing platform, such as

Static Network Designs. This means that the overall design of the network is planned before and the actual network present in the DC resembles the plans; for instance, there are switches, every server is connected to a switch, networks are constructed in a tree-like manner This also includes the assignment of specific machines to specific purposes; in conventional networks, every single server as such is known and itself an organizational unit within the network infrastructure

Conventional networks don't scale out well because that usually means placing hardware every time scale-out happens. VLANs are a problem with scale-out because you're limited to 4096 (+- a few) vlans per physical network space. So working with network technologies such as VLANs quickly becomes a tedious thing in scale-out scenarios

The traffic patterns are generally pre-determined before implementation. Since scale-out happens with hardware servers and switches, it's easier to pre-define what the usage will be.

CLOUD NETWORKING

- Network as a moving target
- Need to Scale-Out
- Unpredictable traffic patterns

How do these assumptions work together in the cloud computing and virtualization context? Remember, a main-purpose of cloud computing is the ability to create scale-out solutions at ease!

Cloud computing networks will typically not be static at all; they will rather need to be very flexible and depending on how many people use the cloud, there will be several different VMs on several hosts belonging to different users - a static network design just won't do here

Since cloud computing environments are scale-out solutions, they must come with built-in scale-out abilities. Of course, this includes the network: conventional, static network configurations make scale-out hard, causing a problem for cloud solutions

Cloud setups will not typically cause predictable traffic patterns; depending on how many users need a cloud installations resources, there might be a whole lot of traffic or a relatively small amount of traffic at a point in time only. Changing the network infrastructure or the network configuration according to that in conventional setups would simply be impossible.

SOFTWARE DEFINED NETWORKING

- SDN helps to deal with a Cloud's networking requirements
- The Magic Word here is: Decoupling!

This leads to Software Defined Networking. What does Software Defined Networking, or SDN, as it is generally referred to, actually mean?

The most important term in the SDN context is decoupling. What is meant here is decoupling the actual network hardware from the software that is running it; in SDN-based setups, the whole network intelligence is implemented by software which itself is deeply integrated into the computing environment Software functionality that might have been taken care of previously by the software running on the network hardware will now be provided by software that itself is a service running in the network

SDN BASICS

Let's jump back to the switch for a second. Within a switch, you will typically have different parts responsible for doing different things. A switch needs to do two things: send and receive data coordinate data it receives and it sends out

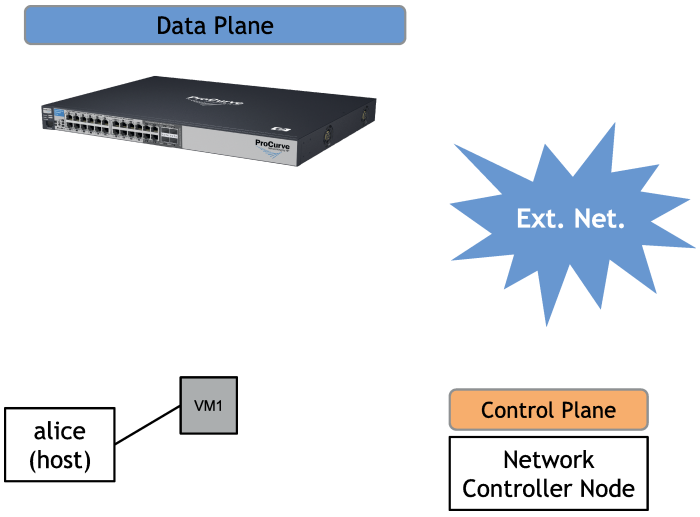
SDN BASICS (CONT.)

Data Plane

Control Plane

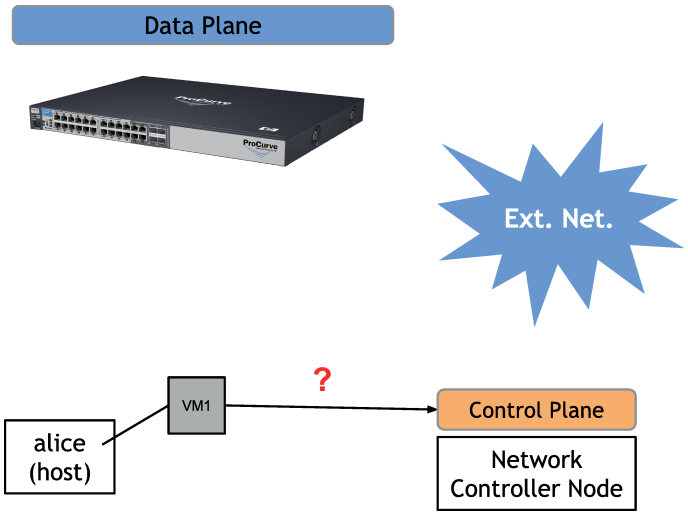
This principle is resembled by two different components present in every switch: the control plane the data plane The data plane receives and sends out packets and acts as a cache The control plane includes the actual network functionality; it is responsible to forward packets to specific ports and to implement VLANs So talking about decoupling, what SDN solutions will typically do is decoupling the data plane from the control plane, i.e. turning what you see here right now into something like this:

SDN BASICS (CONT.)



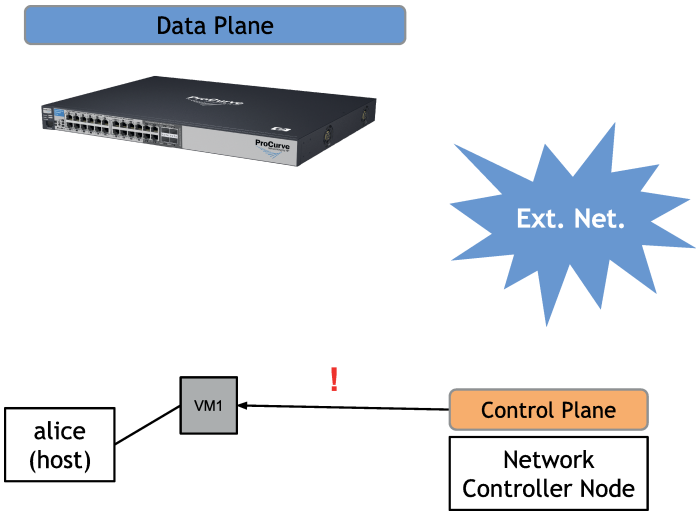
Here, it's fairly obvious that the control plane runs as a separate software on the network controller node while the actual data plane remains on the switch. How do servers and VMs interact with this?

SDN BASICS (CONT.)



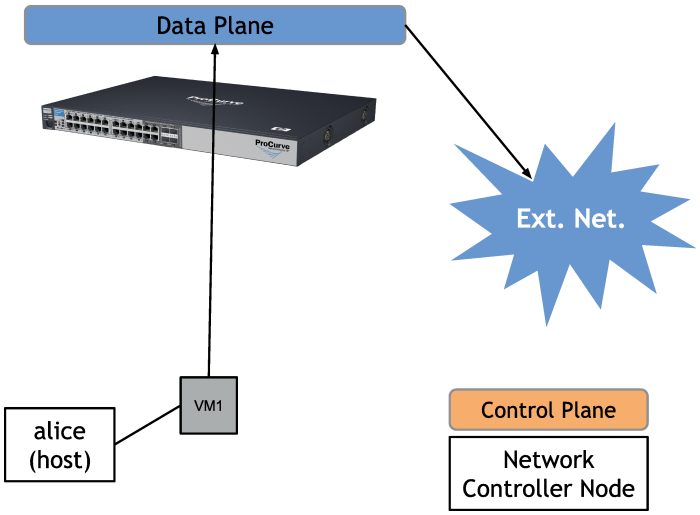
Imagine, we have a VM booting up, requiring network information. It will get the required information from the control plane running on the network controller node and act accordingly

SDN BASICS (CONT.)



The control plane will reply with the required information and supply the VM with all necessary details

SDN BASICS (CONT.)



Finally, the VM will route its packages through the switch, but the switch, in this example, will be nothing but a simple piece of iron without any managing functionality enabled. Everything that may be configured network-wise within the installation can be centrally controlled by influencing the network control plane running on the Network Controller node.

SUMMARY

- SDN for clouds: yay!
- Welcome alternative to convenient networking
- SDN is for networks what virtualization is for computing
- Large scale cloud environments require SDN

As a summary, it is safe to assume that Software-Defined Networking is a welcome alternative to convenient networking architectures SDN is for networks what virtualization is for computing hardware: decoupling hardware and software Large-scale cloud environments require SDN to manage network functionality in a meaningful way

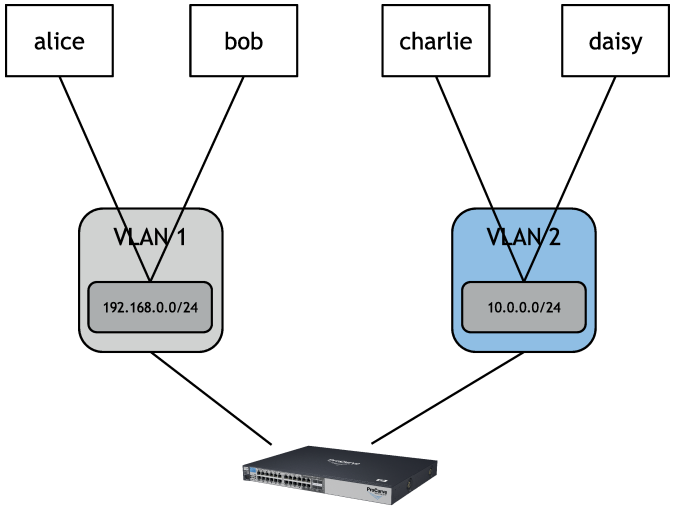
Network Namespaces in Linux

NAMESPACES PRIMER

- In cloud environments, several different users will be using the same host for their VMs
- This requires for networking technology that allows to separate them from each other
- This separation must happen in a safe manner
- Let's take a look at conventional networking approaches first

To understand what Network Namespaces are, first, let's take a look at conventional networking structures and how network interfaces are integrated into this. This might be a rehearsal in some aspects and many of you might already know portions of this -- yet, it's important to draw the complete picture.

CONVENTIONAL NETWORKING



We have seen this slide before - we have a switch, we have servers, these belong to different customers and our customers have their own VLANs. Alice and bob will always ever only talk to each other, and so will charlie and daisy

On Alice, Bob, Charlie and Daisy, the network interfaces which connect to the Switch will typically have one IP address; for special purposes such as failover setups, they might have multiple IP addresses in the same network

There may be virtual machines on alice, bob, charlie and daisy -- either these will use a separate private network which is bridged into the host's network of they will directly bridge into the host network's NIC and use IP addresses within the same IP range

As we have seen previously, static setups such as this don't work very well in the virtualization and cloud computing context

we will typically not have VLANs configured on the switch level

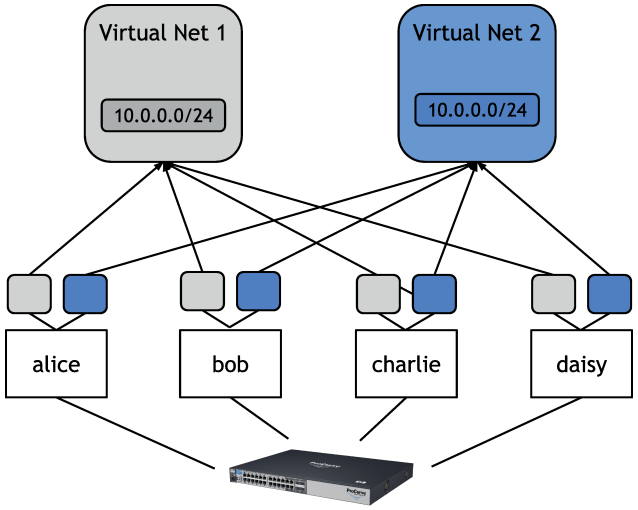
customers will typically not have their own virtualization hosts; they will run their VMs together with others on a generally available pool of hypervisor nodes

VIRTUALIZED NETWORKS

- Conventional networking is very static
- What requirements do virtualized networks have to live up to?

Running several virtual machines from different customers on the same host comes with some requirements. In fact, a virtualized network environment might rather look like this:

VIRTUALIZED NETWORKING



We have virtual networks (customer-specific), we have computing hosts which are not physically split into different network segments (such as VLANs) and we have our VMs running within the virtual networks, all on the same network layer.

VIRTUALIZED NETWORK REQUIREMENTS

- True separation of IP networks
- True separation of data
- NICs shared between customers

Let's take a closer look at the requirements virtualized networks like this one have to live up to:

customer A might want to use 192.168.0.0 for his internal inter-VM network, and customer B might want to use exactly the same network. In a network structure where hardware is decoupled from network functionality (SDN), this must be possible

customer A will not want customer B to be able to see the network traffic of customer A (this might even be a strict requirement with regards to the security of networking)

imagine one hypervisor node running the VMs of 5 different customers. To fulfil the requirements mentioned before, one could follow an approach where every customer gets his own NIC in the hypervisor nodes. This, however, doesn't scale, is tedious and complicated and may become expensive. In fact, all customers running VMs on a hypervisor node need to be able to pipe their data through exactly the same NIC.

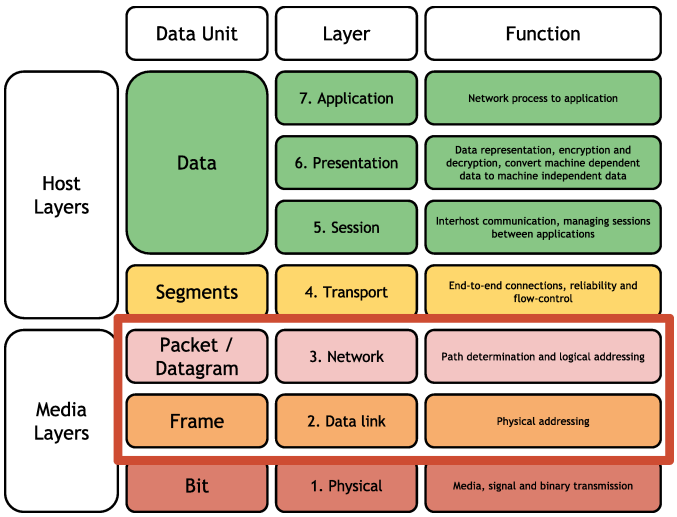
So how can we combine all these factors?

NETWORK NAMESPACES

- The answer is easy: Network Namespaces!
- Work like "chroot for networks"
- Work on a per-process base
- How do they fit into the OSI Layer model?

The answer to this question is easy: Network namespaces! Now what is this, when was it introduced and how does it actually work? Network namespaces may well be explained as "chroot for networks" First of all, Network Namespaces are a container-based virtualization technology present in the Linux kernel network stack In fact, it acts on a per-process base. Processes are assigned a "network namespace" that is specific to them; the namespace may have own network devices (i.e. virtual interfaces bridged into the physical device on the host) and separate IP addresses Namespaces are strictly divided from each other. Programs running in Network Namespace A can not connect to programs in Network Namespace B (to connect two namespaces to each other, routing between the two namespaces is required) Using network namespaces, many processes on a host can get their "own" networking environment while still all processes use the same physical NIC present in the system

NETWORK NAMESPACES IN THE OSI MODEL



Again, here's the OSI layer model - to understand what network namespaces are, imagine they integrate on OSI layers 2 and 3. Any application on top of them will not even realize they exist, for applications running within network namespaces, the network will just look "normal" The Linux kernel does all the rest and makes sure network namespaces work as expected

NETWORK NAMESPACE FEATURES

- Isolation
- Consolidation

Let's take a look at network namespace key features As already pointed out, different network namespaces are completely isolated from each other. Even if application A running in network namespace A gets compromised, that would have no effect on application B runinng in network namespace B.

Using network namespaces in conjunction with a powerful link allows for seamless integration of several dozens of customer's VM on one host via one network adapter; this massively helps to consolidate data centres without having to change the VM-internal network configuration or affecting the security of the separate systems

USING NAMESPACES

- Namespaces are semi-transparent - to use a namespace, a process must be started "within it"
- The program itself, however, will not notice any difference
- Any application can be run inside a network namespace
- The technology is application-agnostic

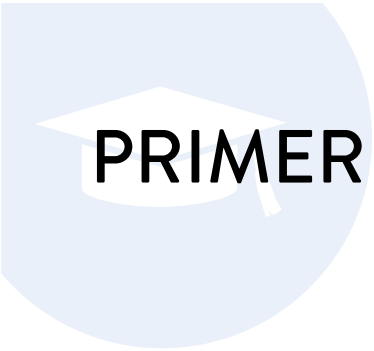
Namespaces are semi-transparent. Starting a process within a specific namespace requires special parameters (or a wrapper within which you execute the program). If the SDN solution in use knows about namespaces, the SDN administrator will not typically notice any difference Any application can be run within a network namespace, there are no application-specific changes required for network namespace support

REQUIREMENTS

- Kernel must support CLONE_NEWNET (part of Linux since 2.6.32)
- requires a compatible `iproute2` package
- Works on most enterprise distribution out of the box

To use networking namespaces in a meaningful way, the kernel needs to support the CLONE_NEWNET function. The first Linux version to have all patches required for this fully integrated was Linux 2.6.32 - so an enterprise distribution with Linux 2.6.32 will work DOES NOT WORK with RHEL 6.4 or earlier (and compatible): https://bugzilla.redhat.com/show_bug.cgi?id=869004 Red Hats OpenStack distributon contains a fixed version of iproute2, so if you want to use Namespaces on RHEL 6.4 or anything compatible, besure to install the iproute2 package from there

An Introduction to OpenFlow



- We've been talking about Software Defined Networking
- But how is it actually implemented?
- What technology is working behind the scenes?
- This is where OpenFlow comes in

Let's talk about OpenFlow. OpenFlow is a network technology used in conjunction with Open vSwitch. In fact, it is what Open vSwitch uses heavily to generate a switching structure within virtualized networks; yet, you will almost never be in direct touch with it, because most of the flow magic is hidden under Open vSwitch's front-end. Let's start with a simple question: What are flows in the SDN context?

CONVENTIONAL SWITCHES

Data Plane

Control Plane

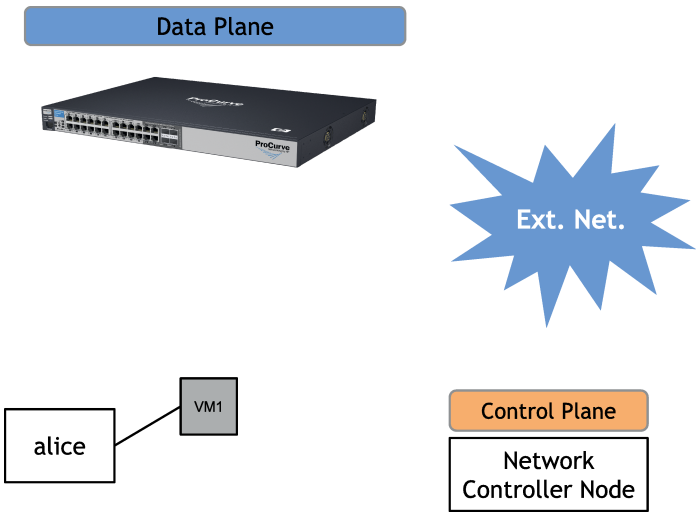
You already know this - this is what a typical Switch looks like; it has a data plane, which is responsible for holding the actual traffic payload, and it has a control plane, which decides which packages need to be delivered to which port. As we have said earlier, in virtualized environments, static switch forwarding planes aren't helpful, because they don't integrate well with the actual networking setup present. Thus, the idea of SDN is to integrate control over the switch forwarding plane(s) directly into the cloud or virtualization management software

OPENFLOW PRIMER

- Open Standard created to build control planes
- Replaces built-in control planes in Switches
- Can be controlled from within a cloud environment
- Works with manageable switches and "iron-only" switches

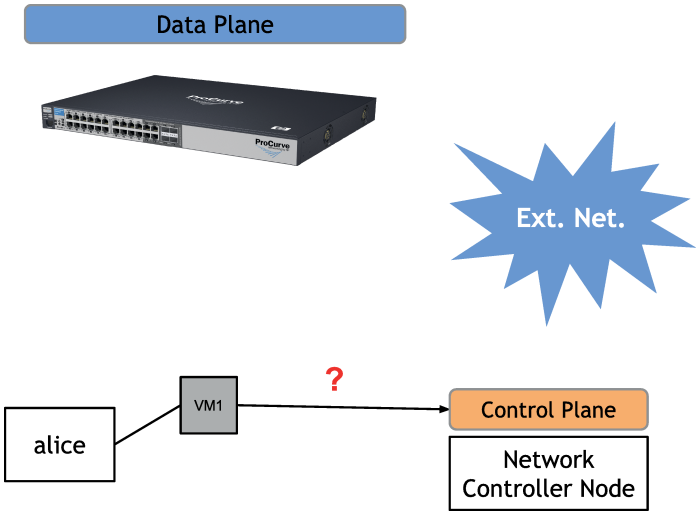
OpenFlow is an open standard created exactly for this purpose. It's a publically available definition of how forwarding planes can work and can be controlled from within the cloud / virtualization management software. "Flow", in this context, refers to the way packets may use to make it from the source -- typically a VM -- to their target. OpenFlow supports numerous operation modes, the two most prominent ways are:

NON-MANAGEABLE SWITCHES



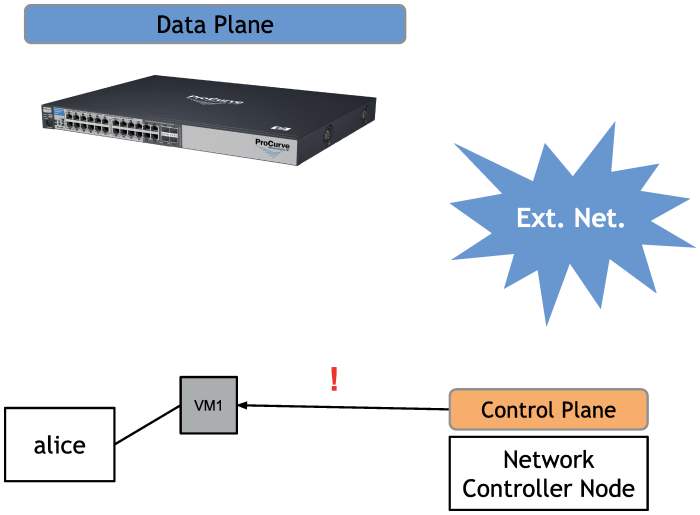
OpenFlow as a separate control plane running on its own host within the network. In such a scenario, the switch is downgraded to "iron only", i.e. the switch itself does not provide any actual management functionality, it doesn't even have to be a manageable switch.

NON-MANAGEABLE SWITCHES (CONT.)



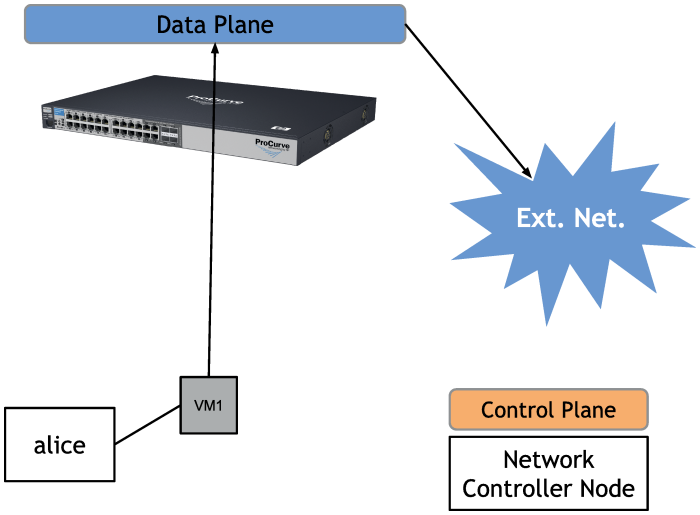
Because in a setup like this, where a VM is running and gets its flow information from the separate, OpenFlow compatible forwarding plane, a request is simply going the route it receives from the forwarding plane.

NON-MANAGEABLE SWITCHES (CONT.)



The information required for the VM's network to function properly is returned by the control plane, and right thereafter, the packet can make its way to its external destination

NON-MANAGEABLE SWITCHES (CONT.)



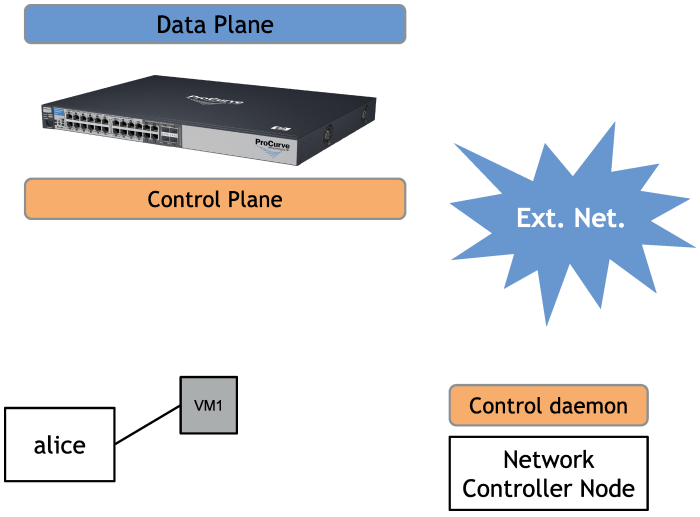
And in the end, the packet will reach its goal.

MANAGEABLE SWITCHES

- OpenFlow also is a nice match for setups in which the switches do speak OpenFlow
- Not many switches support OpenFlow at the moment, but the number is increasing

OpenFlow can also use the management capabilities found on some network switches. If the switch supports OpenFlow, which some switches of the "big" vendors already do, there is no reason not to make use of that. The previous example would look at bit differently with such a switch:

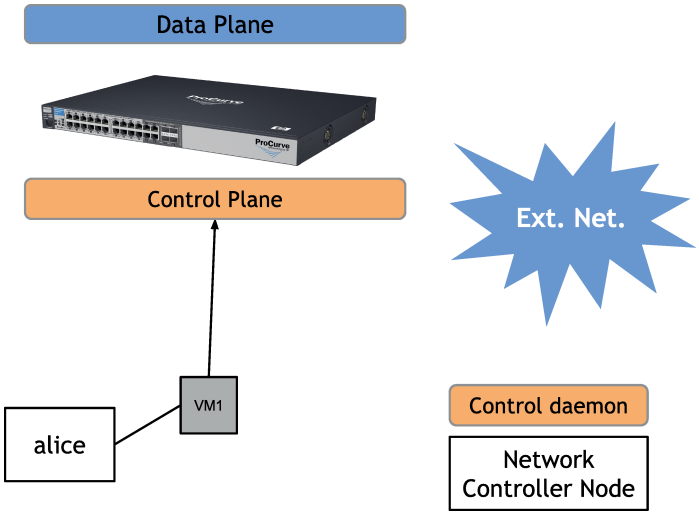
MANAGEABLE SWITCHES (CONT.)



OpenFlow as a separate control plane running on its own host within the network. In such a scenario, the switch is downgraded to "iron only", i.e. the switch itself does not provide any actual management functionality, it doesn't even have to be a manageable switch.

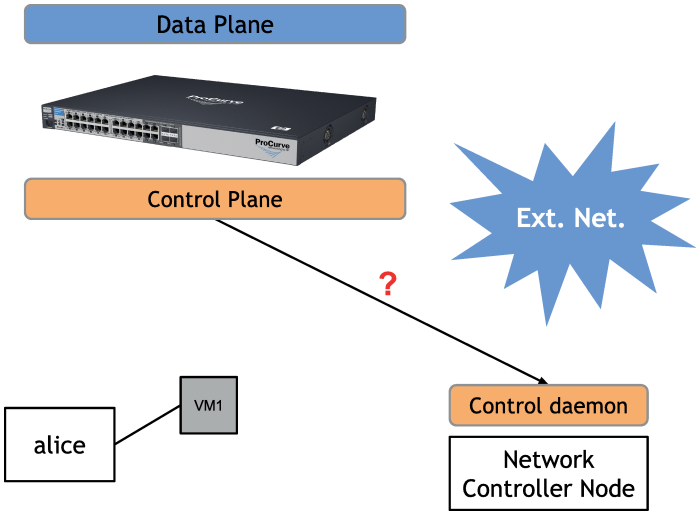
This is a very neat thing, but requires OpenFlow support on the switch, which not every switch has. If the switch is downgraded to an iron-only packet forwarding device, that is not necessary. For testing purposes, using a separate OpenFlow instance and a software-switch is the recommended method.

MANAGEABLE SWITCHES (CONT.)



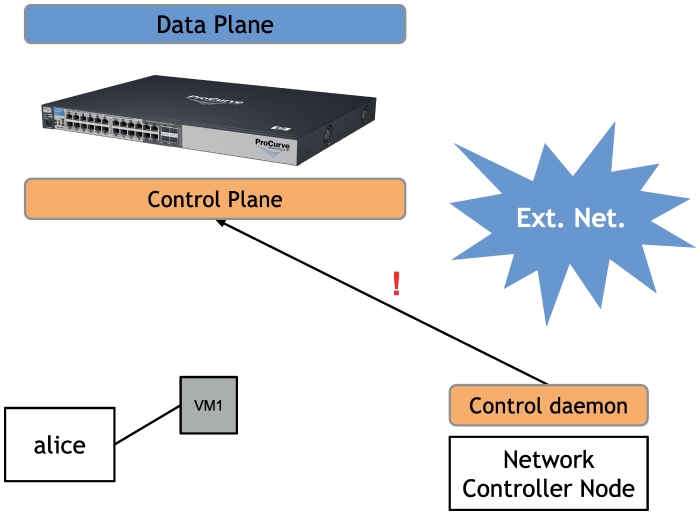
Here, the switch would receive the flows it needs from the SDN control daemon. So if a VM request hits the Switch forwarding plane, the Switch would ...

MANAGEABLE SWITCHES (CONT.)



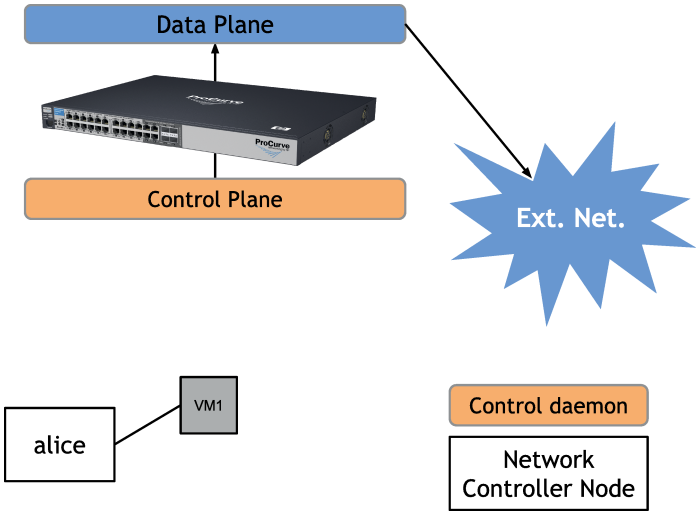
look up what route that packet is supposed to take ...

MANAGEABLE SWITCHES (CONT.)



receive a reply from the SDN and ...

MANAGEABLE SWITCHES (CONT.)



forward the packet accordingly.

An Introduction to Open vSwitch

OPEN FLOW & OPEN VSWITCH

- We have been talking about Open Flow, now what does Open vSwitch have to do with Open Flow?
- Open Flow is a technology that allows to create Network Packet Flows (thus its name)
- But creating flows manually would be tedious, wouldn't it?
- This is where Open vSwitch comes in

The previous session was talking about OpenFlow and how it is a technology that allows to define packet flows (thus its name) in virtualized networks (SDN). This session is about Open vSwitch - how are these two related to each other?

OPEN FLOW SWITCH REFRESHER

- Can be used with "smart" switches or "bare metal" switches
- Open Flow replaces a switches control plane, but where do the control commands come from?
- And what about switches that do not even speak Open Flow? Where do these get their instructions from?

Remember the two general operational modes presented in the last slide deck; OpenFlow setups can be run either in conjunction with OpenFlow capable network equipment (esp. the switches' forwarding planes need OpenFlow support for this to work). If the Switch does not have OpenFlow capabilities, that does not automatically render it unusable for SDN. However, there will have to be a component in the virtualization or cloud environment that acts as a virtual switch to control packet flow, downgrading the actual network equipment to iron-only forwarding devices.

If the Switch does not have OpenFlow capabilities, that does not automatically render it unusable for SDN. However, there will have to be a component in the virtualization or cloud environment that acts as a virtual switch to control packet flow, downgrading the actual network equipment to iron-only forwarding devices.

OPEN VSWITCH PRIMER

- Open vSwitch in fact is a a virtual switch
- Creates "virtual" switches within networks and virtual L2 networks
- Suitable and most commonly used in cloud environments

And that is exactly what open vSwitch does. vSwitch in this context is the abbreviation for "Virtual Switch", so Open vSwitch allows you to create virtual switches in virtualized or cloud environments. Roughly, it's legitimate to say that Open vSwitch is a configuration front-end for OpenFlow, allowing you to create packet flows with OpenFlow by taking care of configuring the network hardware and packet flows according to your requirements.

OPEN VSWITCH INTERNAL FUNCTIONALITY

- Layer 2 based
- Creates virtual interfaces
- Uses bridges on Linux
- Calls OpenFlow in the background

Let's take a closer look at how Open vSwitch actually works.

First of all, Open vSwitch is layer 2 based, so it uses the hardware interfaces provided by the kernel network hardware abstraction layer

On top of the actual network card interface exposed by the Linux kernel, Open vSwitch uses virtual interfaces (e.g. one virtual interface for every VM that is present on a host). You might wonder how this works - after all, typical servers will have one or two NICs but dozens of VMs. That is where bridging comes in:

Bridges in Linux have one specific purpose: They allow to integrate several (virtual) interfaces into one logical network. So to work properly, Open vSwitch needs the physical network interfaces it uses to bet set up as bridges. It then creates and attaches these to the bridge on top of the NIC ("bridges the virtual interfaces into the NIC") Recent Open vSwitch versions may either use the built-in kernel module for bridging or an own bridging module, which provides a compatibility layer for the kernel bridging driver and is thus compatible with it

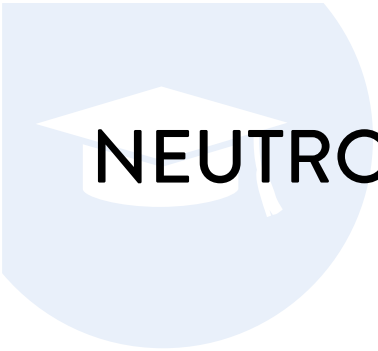
The rest of Open vSwitch's functional is OpenFlow based: Based on OpenFlow rules, packets are forwarded between virtual machines over the NICs -- even across physical borders, i.e. between several different machines.

The actual network switching hardware is iron-only and does not fulfill any management tasks anymore

OPEN FLOW, OPEN VSWITCH, OPENSTACK

- Open vSwitch is the default SDN driver for OpenStack
- Thus, OpenStack also uses Open Flow by default
- This is the recommended setup, but alternatives exist

OpenStack Neutron



NEUTRON PRIMER

- What is OpenStack Neutron?
- It's OpenStack's SDN component
- It allows to create SDNs for use in OpenStack
- What role does SDN play in OpenStack?

Just a quick reminder: From Wikipedia: Software defined networking (SDN) is an emerging architecture for computer networking. SDN separates the control plane from the data plane in network switches and routers. Under SDN, the control plane is implemented in software in servers separate from the network equipment and the data plane is implemented in commodity network equipment.

CONVENTIONAL VIRTUALIZED NETWORKING

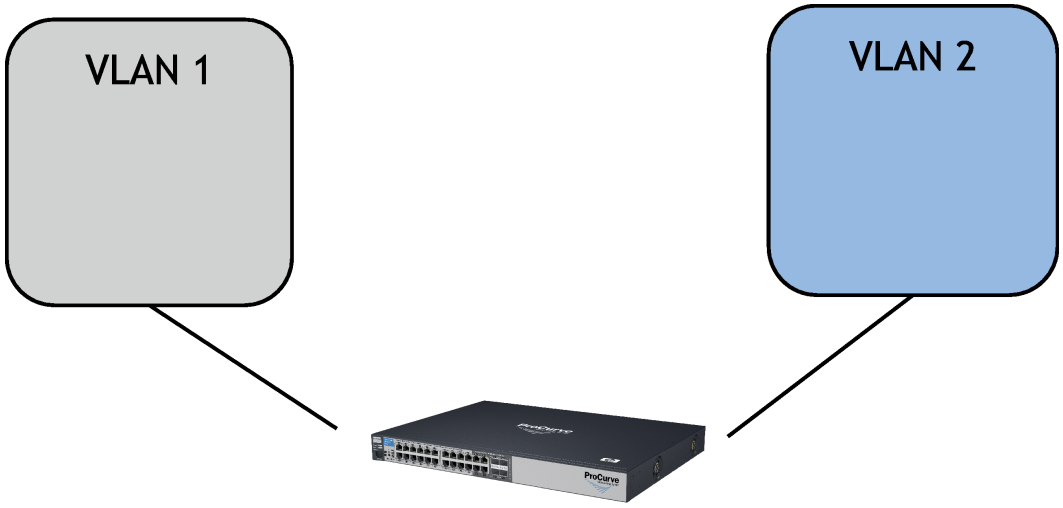
- Let's take a quick look again at how conventional network virtualization works
- This resembles standard virtualization setups, but also ...
- ... Cloud stacks without SDN (such as OpenStack up to the Essex version)

This might look like a waste of time initially, but it is not - understanding networking structures in cloud is essential for understanding what Neutron does

CONVENTIONAL VIRTUALIZED NETWORKING (CONT.)

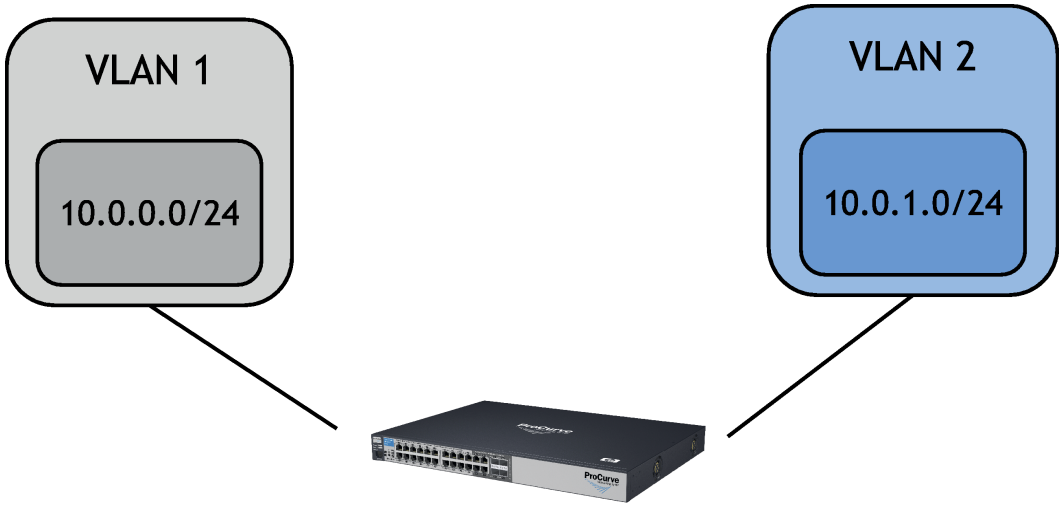
It all starts with a standard switch. All IT setups have these, no matter whether they use conventional networking or SDNs.

CONVENTIONAL VIRTUALIZED NETWORKING (CONT.)



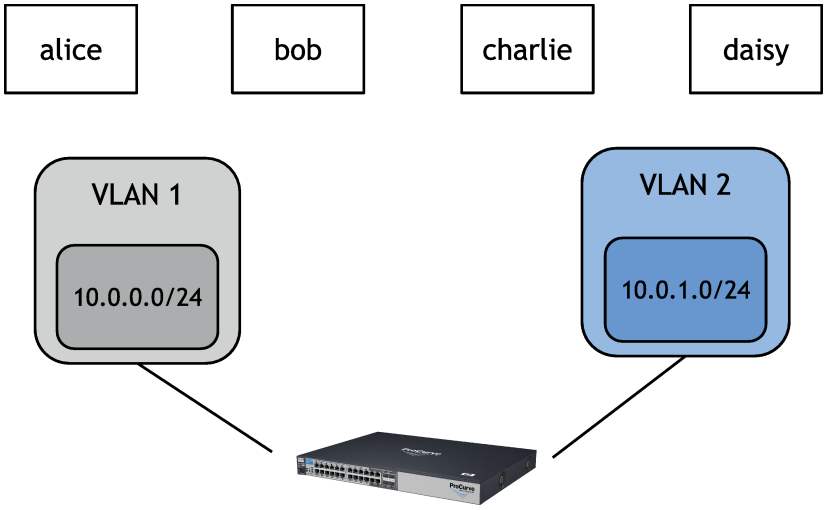
In conventional setups, additionally to switches, you will typically find VLANs, which are used for security reasons and to separate individual customers from each other.

CONVENTIONAL VIRTUALIZED NETWORKING (CONT.)



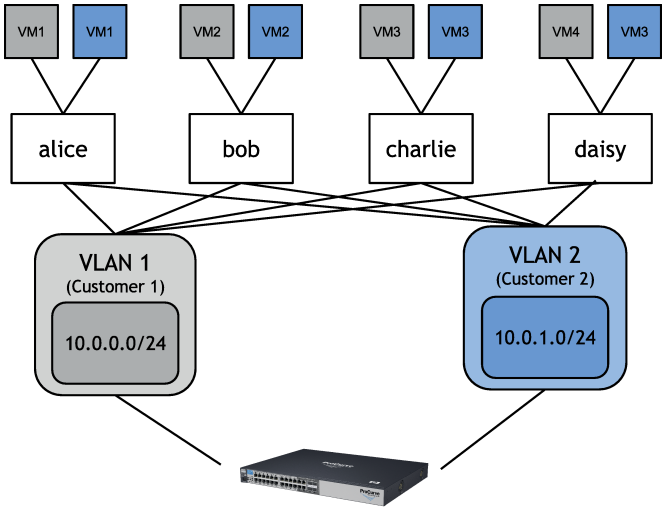
So every VLAN is assigned statically to a customer and will typically contain exactly one VLAN network (as it used to be in the good old days where everything was on iron, as just explained, too)

CONVENTIONAL VIRTUALIZED NETWORKING (CONT.)



Our compute nodes use bridge interfaces and if a VM is to boot up in a certain VLAN ... a virtual interface is created on top of that bridge for the VLAN

CONVENTIONAL VIRTUALIZED NETWORKING (CONT.)



So every compute node needs to have a port in every available VLAN, as it might have to bridge into it This isn't very flexible, is it? Creating new VLANs for customers is a pain, because every time you add a VLAN, you have to add a new VLAN interface on every available node, too And: This way of networking makes it impossible for customers to create their own network topology Because in virtualized environments, users will typically only have access to their actual VMs and their control panel but NOT to switch management -- and that is how it's supposed to be! So long story short: This sucks. That's what the OpenStack people thought, too. And here's what they did to work around it

SDN-ENABLED VIRTUALIZATION SETUPS

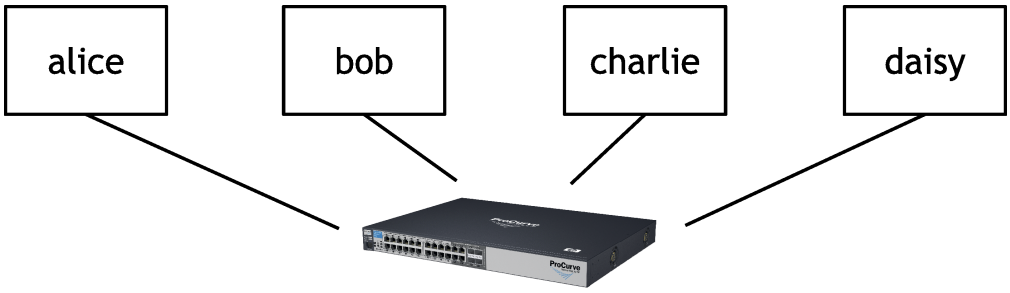
- The whole purpose of SDNs is to work around exactly this inflexibility
- Here, switches are degraded to "bare metal" devices, they do not do packet management anymore
- Instead, they solely act as packet forwarding device
- And the actual functionality is provided by software

Let's take a look at a typical SDN-enabled virtualization setup to see where the differences are

SDN VIRTUALIZATION SETUPS

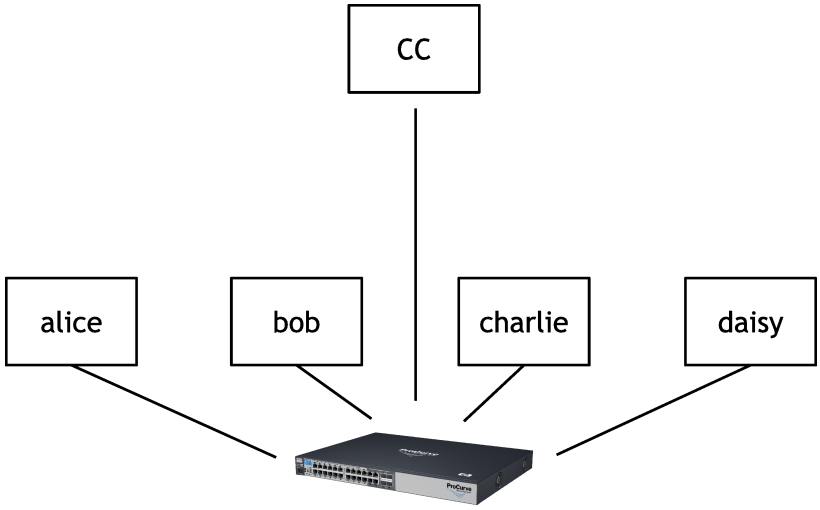
First of all, we have a standard Switch

SDN VIRTUALIZATION SETUPS (CONT.)



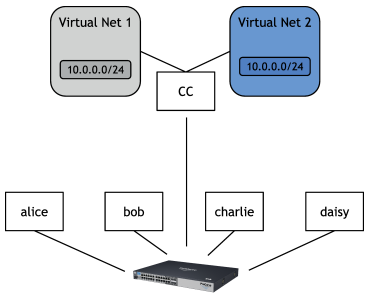
And all our compute nodes are simply plugged into this switch. So there are no VLANs anymore!

SDN VIRTUALIZATION SETUPS (CONT.)



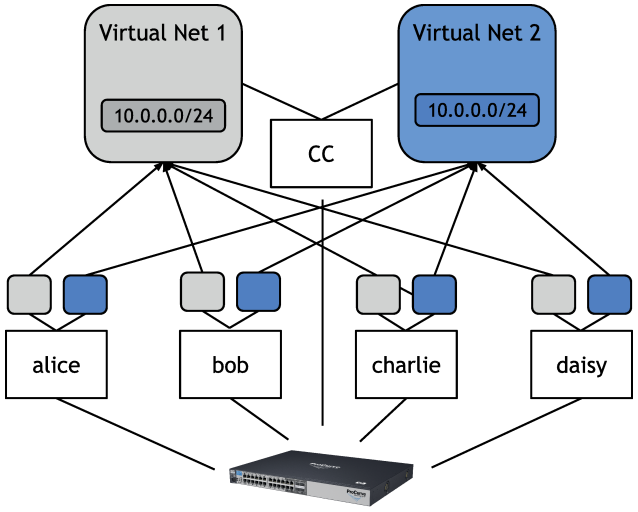
We also have a node that is our cloud controller. This host is running the neutron-api server

SDN VIRTUALIZATION SETUPS (CONT.)



Neutron creates virtual networks. Typically, a virtual network will be specific to a certain customer Cool thing is: Within such a virtual network, I can have multiple IP nets Think of a neutron virtual net like a virtual switch that we will connect our VMs to

SDN VIRTUALIZATION SETUPS (CONT.)



The second part of neutron is the appropriate agent that runs on every compute node. In fact, this agent will provide a direct connection from a compute node to a specific virtual network And these ports are what our VMs will use as their network connection So in fact, with Neutron, the whole networking is virtualized, just as the computing system is

OPENSTACK NEUTRON ARCHITECTURE

- So Neutron is OpenStack's central SDN management software
- How does it make sure that policies defined by the admin are met?
- How does it implement the virtual network across the controller nodes and hypervisors?

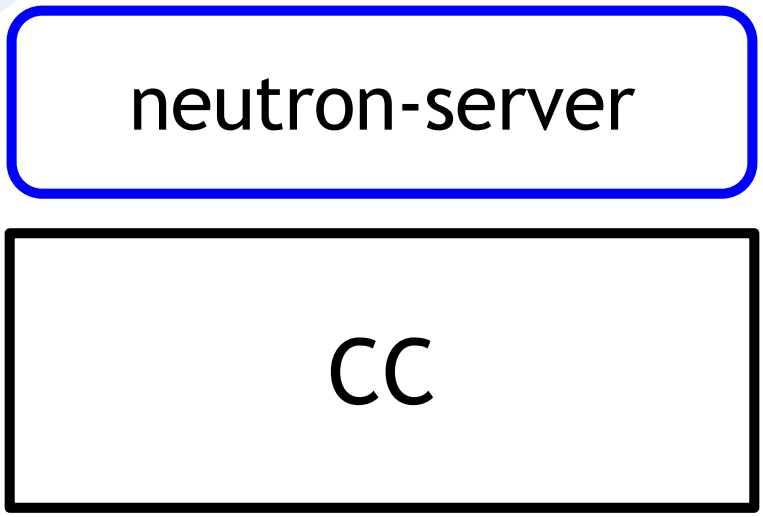
Let's dig a little bit into the actual neutron architecture. You will be confronted with some terms, and it's important to know these, so we'll start by explaining them

NEUTRON-SERVER

- First of all, there is the `neutron-server`
- This is what is running on our cloud controller node
- It's a standard-based, generic API for network operation
- In fact, it's just like any other API OpenStack has

The `neutron-server` provids a ReSTful interface to the outside but itself does not support any SDN functionality. It really is the logic for SDN only.

NEUTRON-SERVER (CONT.)



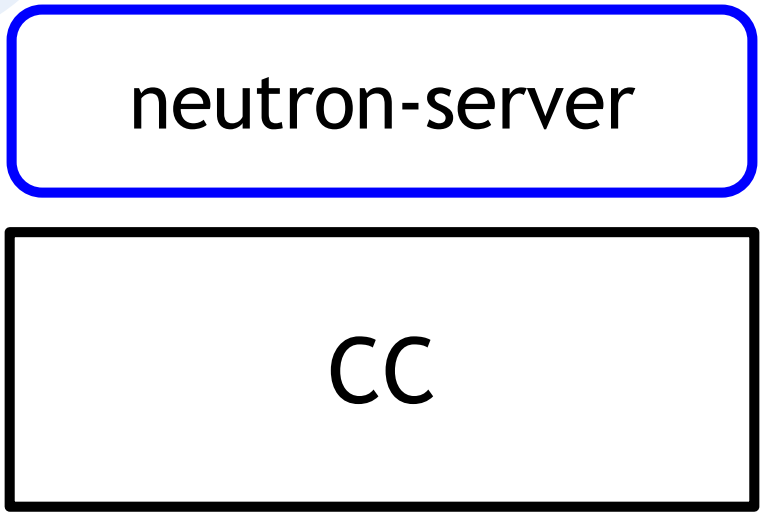
Supporting different standards (like openVswitch, Cisco or nicira NVP) was a design goal of Neutron. That is why `neutron-server` is modular. `neutron-server` itself really only is a generic API. The network magic comes from the ...

PLUGINS

- The `neutron-server` can load plugins
- These are what brings SDN functionality into Neutron
- Plugins allow to add support for a specific network technology to Neutron

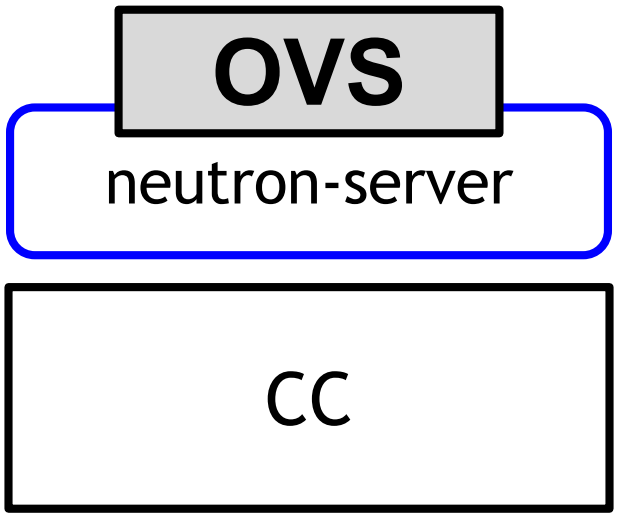
A Plugin in Neutron speak is what makes the Generic Neutron-Server API able to understand a certain networking technology (like OpenVSwitch, Cisco, Nicira NVP ...) Plugins are loaded by the `neutron-server` python process and their functions can then be used.

NEUTRON PLUGINS



So this turns into ...

NEUTRON PLUGINS (CONT.)



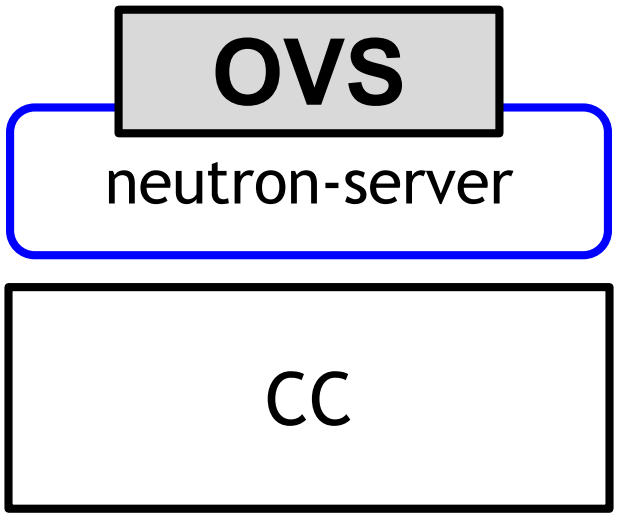
... this. Imagine we want our neutron-server to provide virtual networks based on OpenVSwitch, then we would have to make neutron-server load the Open vSwitch-Plugin (called ovs)

NEUTRON AGENTS

- Agents are a plugins counterpart on the different OpenStack nodes
- Every Neutron agent has a corresponding plugin to be used on
  - hypervisor nodes
  - the network controller node
- There are also generic agents

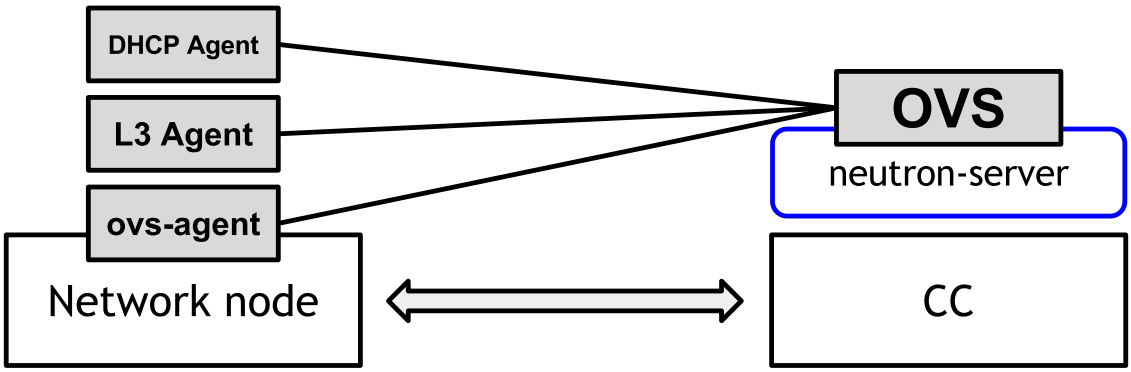
Then, there are Neutron Agents Agents are the Neutron-server plugin's counterpart on the computing node side. That means that computing nodes use Agents to receive settings from the neutron-server (and its running plugin) If you run a neutron-server that uses ovs for networking, you will need the appropriate ovs agent somewhere on your network. Please note: On the previous slides, we were assuming that all neutron components (the server, the plugin and the agents) are running on our cloud controller node; that is not a good idea for various reasons and may be asking for trouble. The official OpenStack documents recommend to have a single, dedicated "networking node" that runs the agents Neutron. So here's the basic overview when following this recommendation:

NEUTRON AGENTS (CONT.)



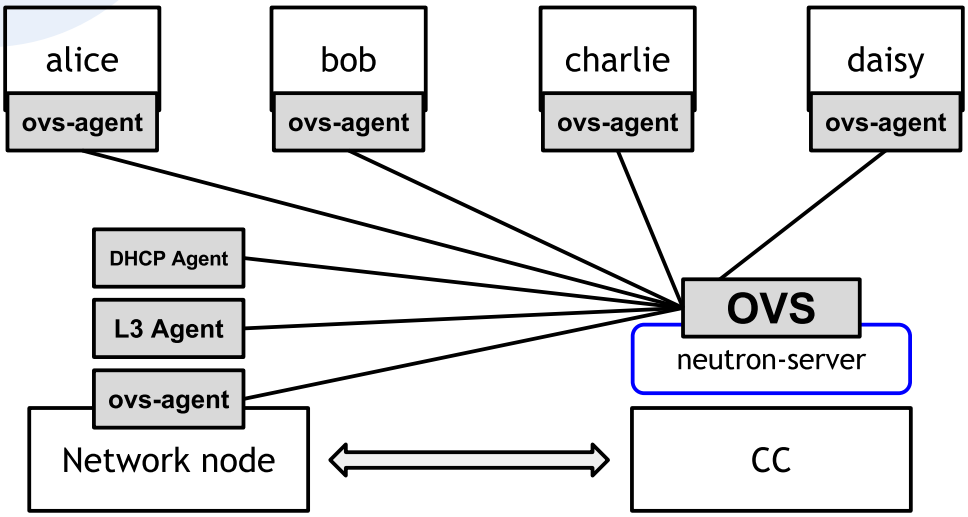
Just to remind you: This is our neutron-server with the OVS plugin enabled. Then there is our networking node running the Neutron agents:

NEUTRON AGENTS (CONT.)



We have three specific agents running here: The ovs-agent, which is the agent that talks to the OpenVSwitch plugin on our cloud controller node The L3 agent, which allows client VMs to be accessible via official ("floating") IPs by setting appropriate iptables rules on the network node The DHCP agent, which supplies client VMs with internal IP addresses via DHCP Then, we have our computing nodes:

NEUTRON AGENTS (CONT.)



Alice, Bob, Charlie and Daisy. And they all have to run their own OVS plugin, too Last but not least, all four compute nodes will talk to the DHCP and L3 agent on our network node to retrieve internal IPs and to get floating IPs when desired So this is what things really look like in this setup.

AGENT OVERVIEW

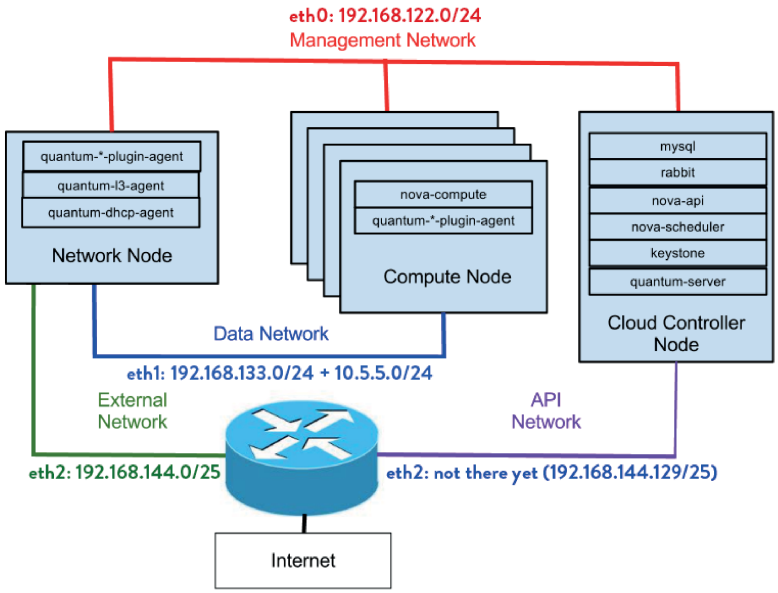
- `neutron-*-agent`
  - Plugin-specific agents, will be dealt with in more detail later
- `neutron-dhcp-agent`
  - Assigns DHCP addresses to freshly spawned VMs
- `neutron-l3-agent`
  - Supplies VMs with an external connection, manages floating IPs

There are numerous agents available for OpenStack: Obviously, we have one agent for every plugin that OpenStack Neutron currently supports.

Then, there is a more generic agent that's called neutron-dhcp-agent. This agent works across all plugins in neutron-server and may be used to supply tenant networks with DHCP services for auto IP assignment It will typically run on its own node, the "network node", which has the main purpose of running the DHCP and the L3 agent

There also is the neutron l3 agent. This agent allows us to provide tenant networks with L3/NAT forwarding to provide external network access via floating IPs. Just like the DHCP agent, this agent can work with any neutron-server plugin It will also typically run on its own node, the "network node", which has the main purpose of running the DHCP and the L3 agent Due to this, the network node will need access to the network with the public IPs in it -- all other servers don't necessarily need that!

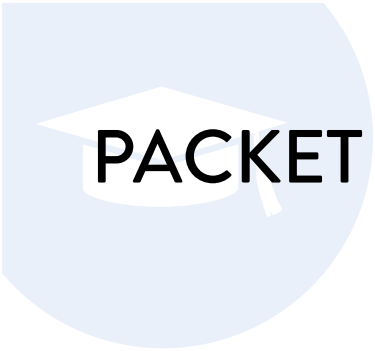
OPENSTACK NETWORK TYPES



Oh by the way - just a quick heads-up now that we know all the network connections existing between several different nodes in OpenStack. If you ever read the official documentation, you will see terms like "management network" and "data network". This picture helps you to understand which is which:

The "management" network is used by the physical hosts inside the cloud to talk to each other. Every node must be connected to this. The "data" network is the network built atop of GRE tunnels or VLANs. The network node and all hypervisor nodes must be connected to this. The "external" and "API" networks are supposed to be external networks with official IP addresses. They may be the same network, but for security reasons, it might be desired to have them separately.

OpenStack Neutron Packet Flows

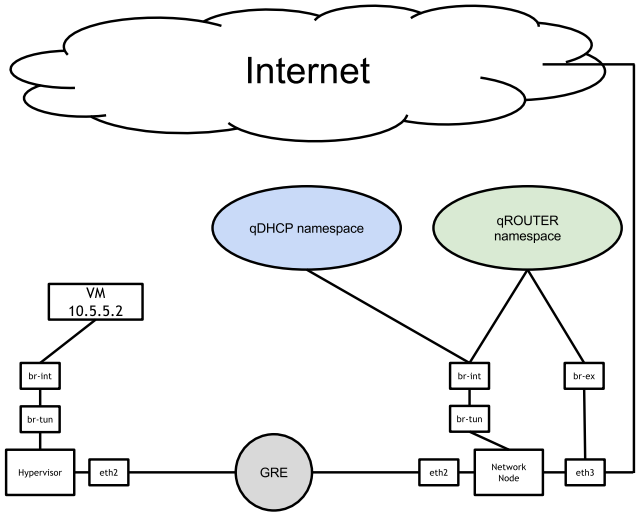


PACKET FLOW PRIMER

- In Neutron SDN installations, we have several interfaces
  - br-int / br-tun
  - br-ex
  - eth1/2/3
- What are these good for? And what routes do packets take on their way through our SDN?

As you have seen by now, we are talking about numerous interfaces here, br-int, br-tun, br-x, eth1/2/3, and for Neutron beginners, it's hard to understand what route within all these devices and hosts packets actually take. So let's take a closer look.

STANDARD SITUATION



This is the standard situation. We have

A hypervisor node with

br-int: This is what the VMs connect to; every VM running on a hypervisor will have its virtual network interface attached to br-int

br-tun: This is the tunnel interface that, in GRE mode, adds GRE tunnel headers to the packages

eth2: The actual internal physical interface where packages send out via eth2 go to

The GRE tunnel, which, of course, is virtual only and would have all hypervisors and the network node in it

The networking node with

br-tun/br-int, which serve the same purpose they serve on the hypervisor node

br-ex physically attached on top of eth3, which is used for communication to the outside

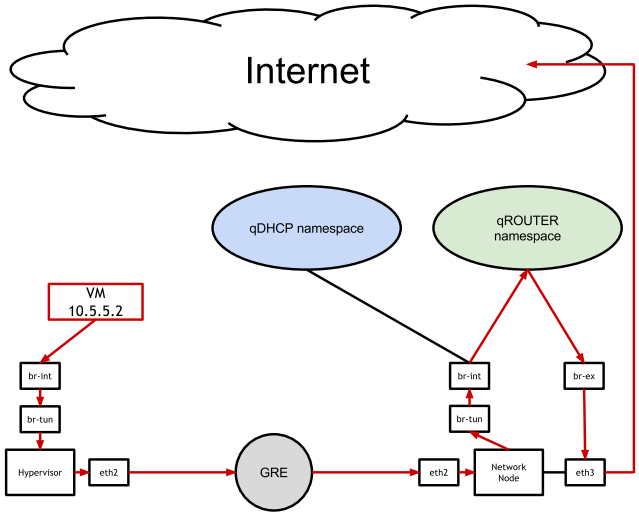
Network namespaces:

qDHCP as interface within the GRE tunnels (one for every tenant network)

qROUTER for connections to the outside (one for every external network)

Now, let's take a look at what route packets take. Let's start with an easy example

VMS ACCESSING THE INTERNET



The packet leaves the VM and hits the hypervisor node on br-int

From there, it is forwarded to br-tun, which adds the GRE encapsulation to it

Finally, it leaves the hypervisor on eth2 into the GRE network

Finally, it reaches eth2 on the network node

It gets send through br-tun, which removes the GRE Headers

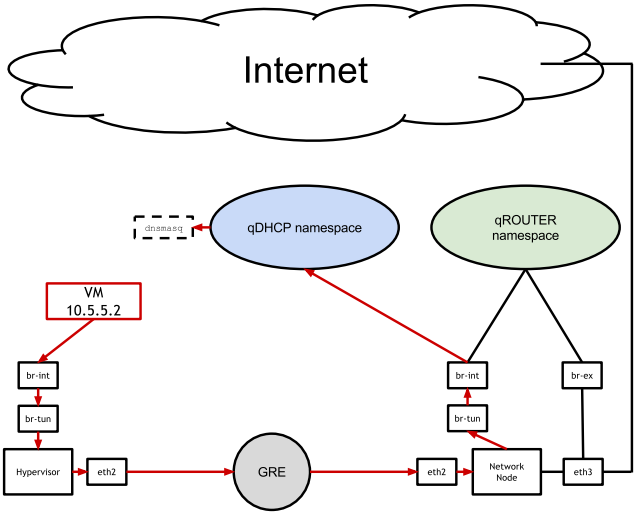
It is then forwarded to br-int

br-int on the network node is connected to the qROUTER namespace

That is why we defined our external network as router for the admin tenant network earlier!

From br-int, the packet hits the external bridge (br-ex) and finally leaves the network node into the Internet

VMS SENDING DHCP REQUESTS



The packet leaves the VM and hits the hypervisor node on br-int

From there, it is forwarded to br-tun, which adds the GRE encapsulation to it

Finally, it leaves the hypervisor on eth2 into the GRE network

Finally, it reaches eth2 on the network node

It gets send through br-tun, which removes the GRE Headers

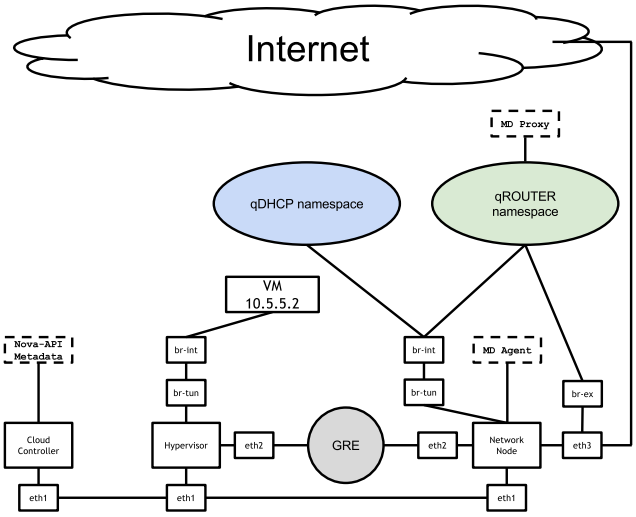
It is then forwarded to br-int

br-int on the network node is connected to the qDHCP namespace

The dnsmasq processes actually are running within the qDHCP namespace context

The qDHCP request is then answered accordingly

VMS LOOKING FOR META-DATA



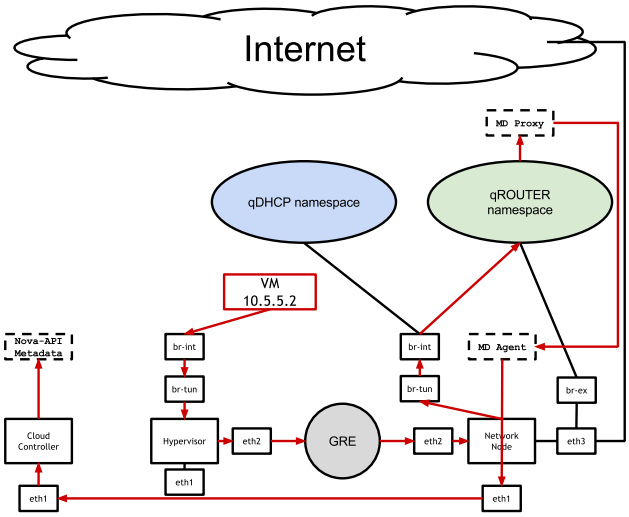
To get a better understanding of the process, first, we need to add the cloud controller to our scheme here, and also, we need to add the so-called management network, which all nodes are physically connected to, with no tunneling, via the eth1 physical interface

Also, we have the Nova-API metadata service running on our cloud controller. This is the source of metadata, this is the service that VMs eventually need to connect to to get meaningful metadata information

And then, we have the metadata-agent and the metadata proxy running on the network node

So how to packets flow?

VMS LOOKING FOR META-DATA (CONT.)



Let's start with the actual request coming from the VM. It takes its usual way and will eventually make it to the qROUTER namespace Remember: Targes to 169.254.169.254, thus leaves the VM to its default route, which is an IP inside the qROUTER namespace

Within the qROUTER namespace, the package will be forwarded via DNAT from port 80 to the qrouter's port 9697

... which is where the Metadata Proxy is running

The Metadata Proxy will relay the packets to the Metadata agent running on the Network node host

This happens outside of any TCP/IP network connectivity; in fact, the MD agent has a UNIX socket open in /var/lib/neutron/metadata_proxy which the Metadata proxy sends the packages into