Virtual Network Subsystem

Use-Cases

- Creating virtual SDN networks for tenants
- Slicing regions of networks for use by different tenants (M-CORD)
- Federation - exposing abstracted view to peer/parent controllers (E-CORD)

General Approach

There are many different ways through which network virtualization can be offered to ONOS applications. On one end of the spectrum is a scheme used by OpenStack where virtual network is exposed as a topology-less set of end-station hosts with have inherent L2/L3 connectivity provisioned between them; this is usually accomplished via overlay such as VLAN, VxLAN or GRE tunnels. On the other end of the spectrum is a scheme employed by OpenVirtex (OVX) where virtual networks are exposed as SDN networks without any inherent connectivity; in order to program connectivity, one must use OpenFlow to control and program virtualized network devices.

As part of OpenStack and OPNFV integration, ONOS already provides means to provision OpenStack-style virtual networks via overlays. The Virtual Tenant Network (VTN) application offers these capabilities.

In order for ONOS to offer applications and users the ability to program their own virtual networks, ONOS needs to also support the ability to create such SDN-capable virtual networks. This is where the virtual network subsystem comes in along with its set of first-class virtual network model constructs, e.g. virtual device, virtual link, virtual port.

Basic Concepts

Virtual network belongs to a tenant

In order to allow tracking virtual networks to tenants, ONOS supports a notion of a tenant. However, this notion is purposefully kept very shallow and a *tenant is represented merely by a tenant ID*. This way, the tenant ID effectively acts as a foreign key that can be used to tie information with other external systems, like OpenStack, which have been designed to keep deeper information about tenants.

Virtual network has virtual devices and links

In order to represent topology, ONOS *virtual networks form a graph consisting of virtual devices and virtual links*, derivatives of devices and links. In this manner, the virtual network topology graph can be traversed and treated in the same manner as a physical network topology graph.

Virtual device has virtual ports

Virtual devices consist of virtual ports, each of which is *realized by a port from the underlying network*. Therefore, the only ties between the virtual network and the underlying physical one are the virtual port to physical port mappings;*a virtual device is effectively a slicing/splicing of devices from the underlying network*.

TODO: Add a diagram showing possible scenarios with virtual to physical mappings, including interior ports as well as edge ports and virtual hosts.

Ali: Virtual hosts should continue to belong to the virtual network regardless of their location.

Virtual devices & links are elastic entities

Since the definition of virtual devices and virtual links are expressed merely via their constituent port entities, they are not backed by a specific path and therefore do not necessarily depend on a specific structure of the underlying network.

Q: Should a virtual port be represented by a group of ports? Group of infra ports?
Ali: Suggestion to leave this for later.

Ali: Allow ability to specify "mode" when creating links, e.g. fail-safe, specific paths, etc. The idea is to allow different recovery models.

Basic Operation

Since virtual networks have topology, apps can traverse topology graph using a regular `TopologyService` and/or `PathService` capabilities, though only in the context narrowed to a specific virtual network.

Due to the elastic nature of the binding between the virtual network and the underlying network, the virtual network topology changes only in response to SCC changes in the underlying network, meaning when the continuity of the underlying network changes. Therefore, the virtual devices and links go up/down only when potential for connectivity and availability changes in the underlay.

Similarly, because virtual networks have no inherent connectivity, no action needs to be taken when a network is created. Instead, the act of programming the virtual network drives action on the underlying network; any logical tunnels need to be created on the underlying network only when connectivity is requested at the virtual network topology.

Virtual Network Admin Service

To permit construction of virtual network, a privileged `VirtualNetworkAdminService` is provided offering the following operations:

- Register/unregister tenants (`TenantId`)
- Create/remove networks (for tenant)
- Create/remove device/link/host (for network)
- Create/remove port (for device)
  - This also binds the port to the physical network

Virtual Network Service

To query the virtual network inventory, the following list of operations is provided by the `VirtualNetworkService` interface:

- List of tenants (`TenantId`)
- List of networks (by tenant)
- List of devices/links/hosts (by network)
- List of ports (by device)

Apps can also obtain “narrowed” services using which they can operate on the virtual network just as they would on a regular physical network. These services are confined to the only those entities comprising the given virtual network and include the following:

- `DeviceService`, `LinkService`, `HostService`, `TopologyService`, `PathService`
- `IntentService`

Virtual Network Provider & Service

In order to allow the virtualization mechanism to vary, the virtual network subsystem purposefully separates the modeling and tracking of the virtual networks apart from the mechanism used to actually realize the networks on the data-plane. This separation is enforced using the `VirtualNetworkProvider` and `VirtualNetworkProviderService` interfaces. These interfaces are still in flux as we experiment with different mechanisms in order to find the minimal necessary interface with sufficient signaling between the virtual network subsystem and data-plane providers.

In general, the responsibilities of a virtual network provider include the following:

- Provide elastic connectivity “tunnels”
  - for link traversal
  - for device “backplane” traversal, i.e. port connectivity
  - set up “on-demand” in response to programming edicts
- Maintain tunnels in face of failures

    - device and link failures in underlying network
    - congestion in underlying network
- Report events to provider service
    - irrecoverable network outages

Virtual Networks & Apps

On-Platform Apps

Because ONOS on-platform apps are effectively core-extensions that are deployed in symmetric fashion across the entire ONOS cluster and because the base OSGi platform is capable of only limited means of compute/memory resource isolation, ONOS applications cannot be deployed on behalf of specific tenants or specific virtual networks. As such they cannot be tenant or network-specific.

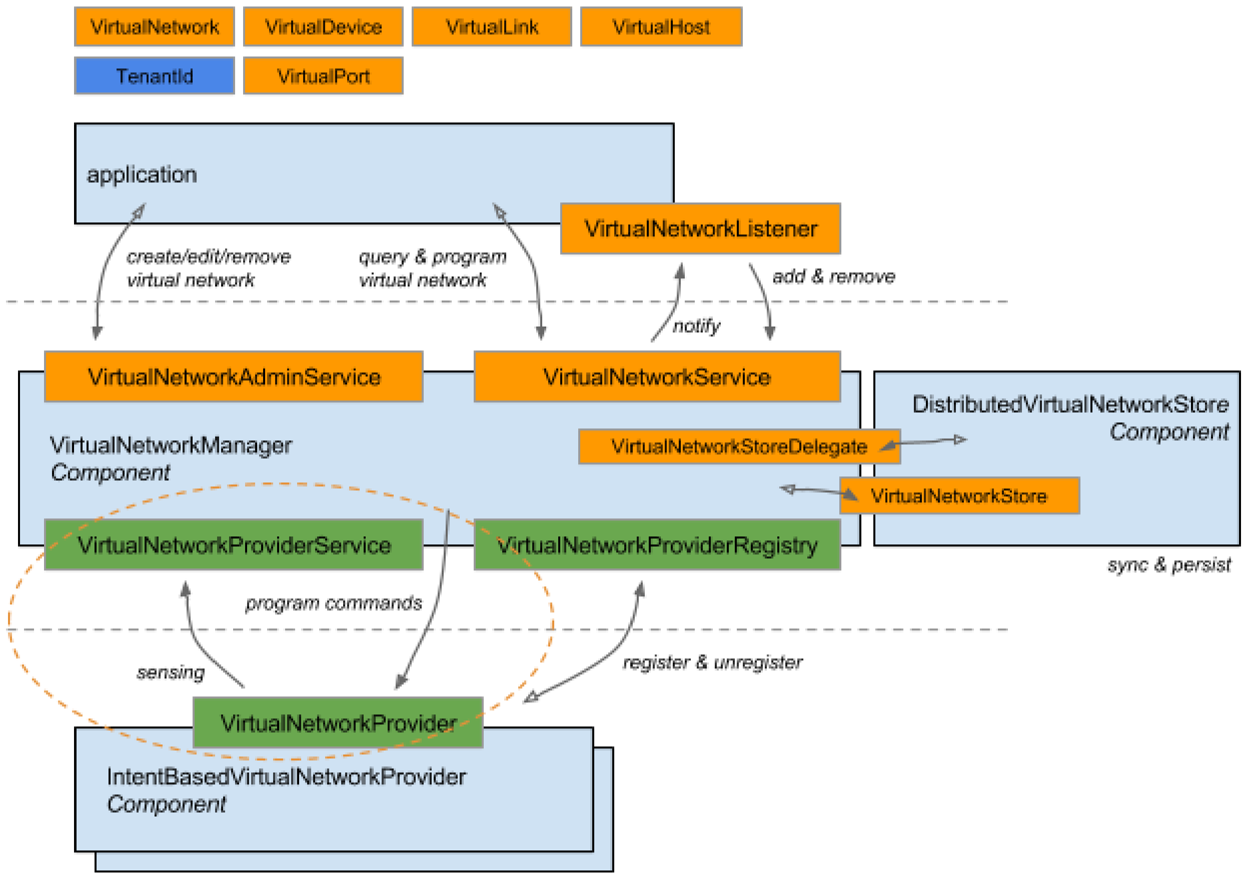
On-platform apps can, however, be tenant and network-aware, meaning that they can operate on behalf of all tenants or networks in their respective contexts.

Off-Platform Apps

Off-platform applications, which by definition run external to the ONOS OSGi platform, and which do not have to adhere to the symmetric deployment constraint are free to be tenant-specific or virtual network-specific and operate using the ONOS REST API to create and control virtual networks. Because such apps run on a separate platform, compute & memory resource isolation is outside of ONOS' scope and would typically be done using OS specific means or by using dedicated hardware.

It should be noted that use of REST API for tenants to interact with their networks is predicated on existence of a scoped authorization scheme, which is one of the features on the roadmap - see section below.

High-Level Design


VirtualNetwork
VirtualDevice
VirtualLink
VirtualHost
TenantId
VirtualPort
application
VirtualNetworkListener
create/edit/remove virtual network
query & program virtual network
add & remove
notify
VirtualNetworkAdminService
VirtualNetworkService
DistributedVirtualNetworkStore Component
VirtualNetworkManager Component
VirtualNetworkStoreDelegate
VirtualNetworkStore
VirtualNetworkProviderService
VirtualNetworkProviderRegistry
sync & persist
program commands
sensing
register & unregister
VirtualNetworkProvider
IntentBasedVirtualNetworkProvider Component

Note: The circled area continues to be subject of design and experimentation.

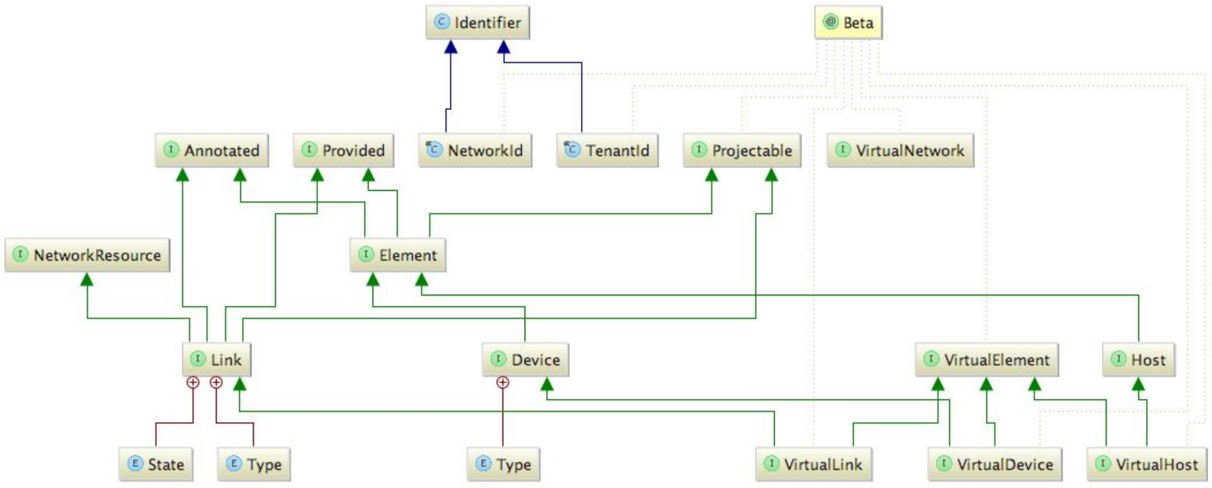
The general idea of this design is to provide a clear separation between modeling of the virtual networks and their realization on the physical underlays. The model and the methods to manipulate the structure of the virtual network remains the same regardless of how the virtual networks are realized.

In order to provide this separation, the `VirtualNetworkProvider` and `VirtualNetworkProviderService` interfaces, which provide the basis for this boundary, need to be design with the minimum necessary surface to allow these parts of the system to communicate, but without introducing dependencies on specific tunneling mechanisms that may be used to implement the virtual network providers. This continues to be an area of active development.

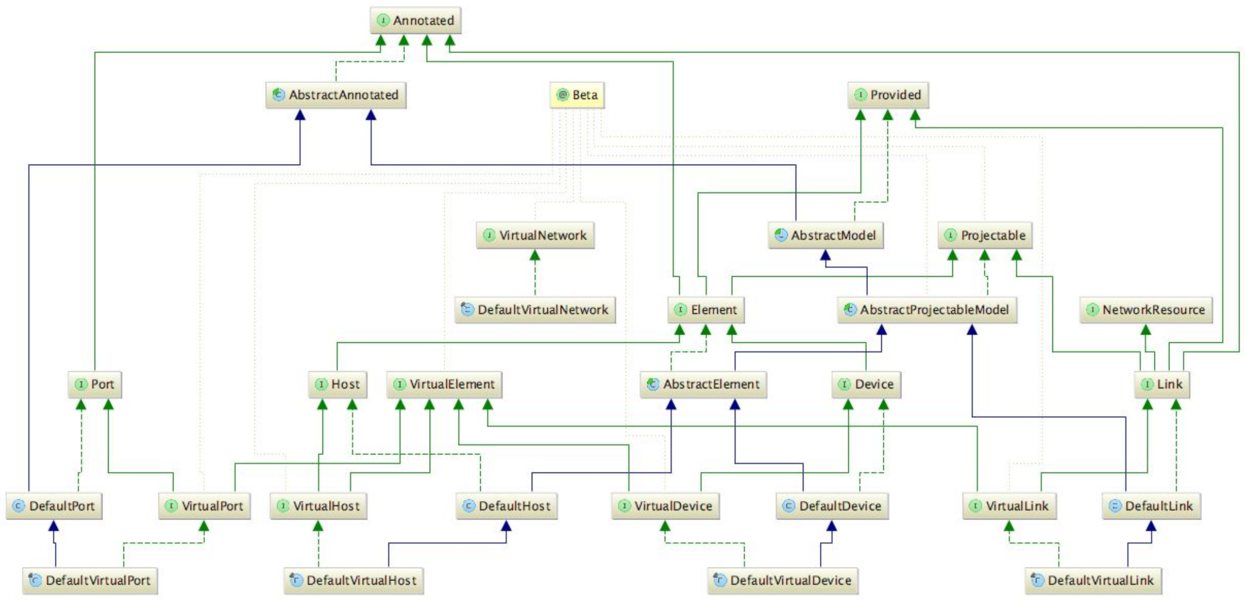
The code is located in the incubator portion of the source tree and under the `org.onosproject.incubator.net.virtual` Java package hierarchy. To reflect the fact that the APIs are still in flux, the model and service interface code is marked with `@Beta` annotation.

Model Object Class Hierarchy

The following UML diagram depicts the interface hierarchy of the various virtual network model entities and their relationships with the rest of the ONOS core model.

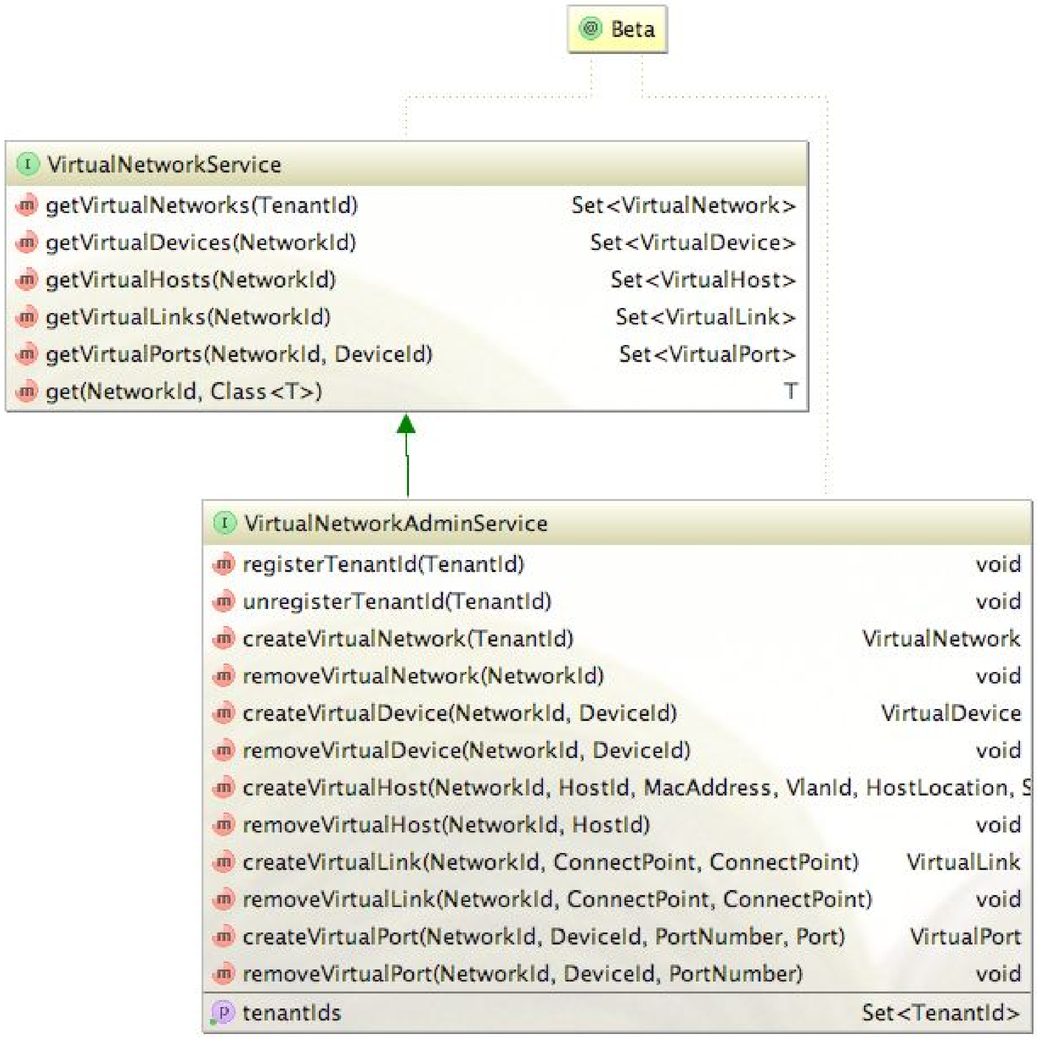


Note that the virtual elements such as virtual device, link and host descend from their core model counterparts.



Virtual Network Service & Admin Service

The following depicts the functionality available from the `VirtualNetworkService` and the administrative extension `VirtualNetworkAdminService`. Note that the create/edit/delete functionality is available solely through the administrative interface, whereas the query capabilities are available through the regular service interface.


Beta
VirtualNetworkService
getVirtualNetworks(TenantId) Set<VirtualNetwork>
getVirtualDevices(NetworkId) Set<VirtualDevice>
getVirtualHosts(NetworkId) Set<VirtualHost>
getVirtualLinks(NetworkId) Set<VirtualLink>
getVirtualPorts(NetworkId, DeviceId) Set<VirtualPort>
get(NetworkId, Class<T>) T
VirtualNetworkAdminService
registerTenantId(TenantId) void
unregisterTenantId(TenantId) void
createVirtualNetwork(TenantId) VirtualNetwork
removeVirtualNetwork(NetworkId) void
createVirtualDevice(NetworkId, DeviceId) VirtualDevice
removeVirtualDevice(NetworkId, DeviceId) void
createVirtualHost(NetworkId, HostId, MacAddress, VlanId, HostLocation, S
removeVirtualHost(NetworkId, HostId) void
createVirtualLink(NetworkId, ConnectPoint, ConnectPoint) VirtualLink
removeVirtualLink(NetworkId, ConnectPoint, ConnectPoint) void
createVirtualPort(NetworkId, DeviceId, PortNumber, Port) VirtualPort
removeVirtualPort(NetworkId, DeviceId, PortNumber) void
tenantIds Set<TenantId>

In order to operate on the virtual network, applications are expected to use the `T VirtualNetworkService.get(NetworkId, Class<T>)` method to obtain a reference to the specified class of service, which has been narrowed to operate solely in the confines of the specified virtual network. In this way, an application can use these services to operate on the virtual network in exactly the same way as it would if it were operating on a physical network.

Virtual Gateways

An expected requirement is to allow customers to connect virtual networks to internet and to other virtual networks via a gateway.

There are several ways in which this capability can be accomplished, but presently no specific approach has been selected yet.

REST API

Presently, the virtual network subsystem exposes a REST API end-point to allow off-platform applications to query, create, edit and remove virtual networks and to program them via intents.

Presently, the end-point does not provide any authorization capabilities to limit tenants to operate only on their own networks, but this functionality is expected to be added in the future as the implementation of the subsystem matures.

CLI

A set of CLI commands have been provided to allow the administrator to use the ONOS shell to query, create, edit and remove virtual networks. The commands are listed below:

- vnet-tenants - list tenant ids
- vnet-add-tenant - add a new tenant
- vnet-remove-tenant - remove a tenant

- vnets - list virtual networks
- vnet-create networkId tenantId - create a new virtual network
- vnet-remove networkId - remove virtual network

- vnet-devices - list all virtual devices in a virtual network
- vnet-create-device networkId deviceId - create a virtual device in a virtual network
- vnet-remove-device networkId deviceId - remove a virtual device from a virtual network

- vnet-ports - list all virtual ports in a virtual network
- vnet-create-port networkId deviceId/portNumber physDevice/physPortNumber - create a virtual port
- vnet-remove-port networkId deviceId/portNumber

- vnet-links - list all virtual links in a virtual network
- vnet-create-link networkId srcDeviceId/portNumber dstDeviceId/portNumber [bidirectional]
- vnet-remove-link networkId srcDeviceId/portNumber dstDeviceId/portNumber [bidirectional]

Limitations & Caveats

OpenFlow programming

Current approach does not expose a device-specific configuration of programming API for virtual devices. However, the design does not preclude it and the long-term goal is to enable virtual device programming via Flow Objectives and based on that expose OpenFlow interface through which virtual devices can be controlled via network controllers that support OpenFlow protocol.

Virtual Network Gateways

Design for connecting virtual networks to each other and to internet has not been selected/specified yet.

REST API Authorization

Current REST API has no authorization mechanism.

Summary

- Virtual networks have no inherent connectivity
- Instead, connectivity is programmed via intents (in future flow objectives)
- Connectivity for link & device traversal are setup only when connectivity is programmed
- Virtual port is realized by underlying port
- Virtual networks must be able to share ports
    - to share exterior ports, ingress traffic must be tagged
    - to share interior ports, pass-through traffic is tunneled

Pending Questions, Suggestions & TODOs

Thomas: Add diagrams showing various forms of mappings between virtual & physical.
Aihua: Are the virtual networks L2/L3 or are they capable of virtualizing L0/L1 as well, e.g. transport network virtualization?
Bill: Advertise the document & video to solicit feedback and comments.
Bill: Can we get use cases from {E,R,M}-CORD how this will be used?
E.g. Running rcord and ecord together; MVNO; how to get the same functionality as VTN

Yuta: E-CORD has a crude big switch app that is used to expose DC to parent cluster. Maybe we should move to the virtualization abstractions.

Ali: this is more of a federation controller, OF isn't desired here, this is more of an API call interaction