Quantum Multi-Switch Plugin Framework

Sumit Naiksatam, Ram Durairaj

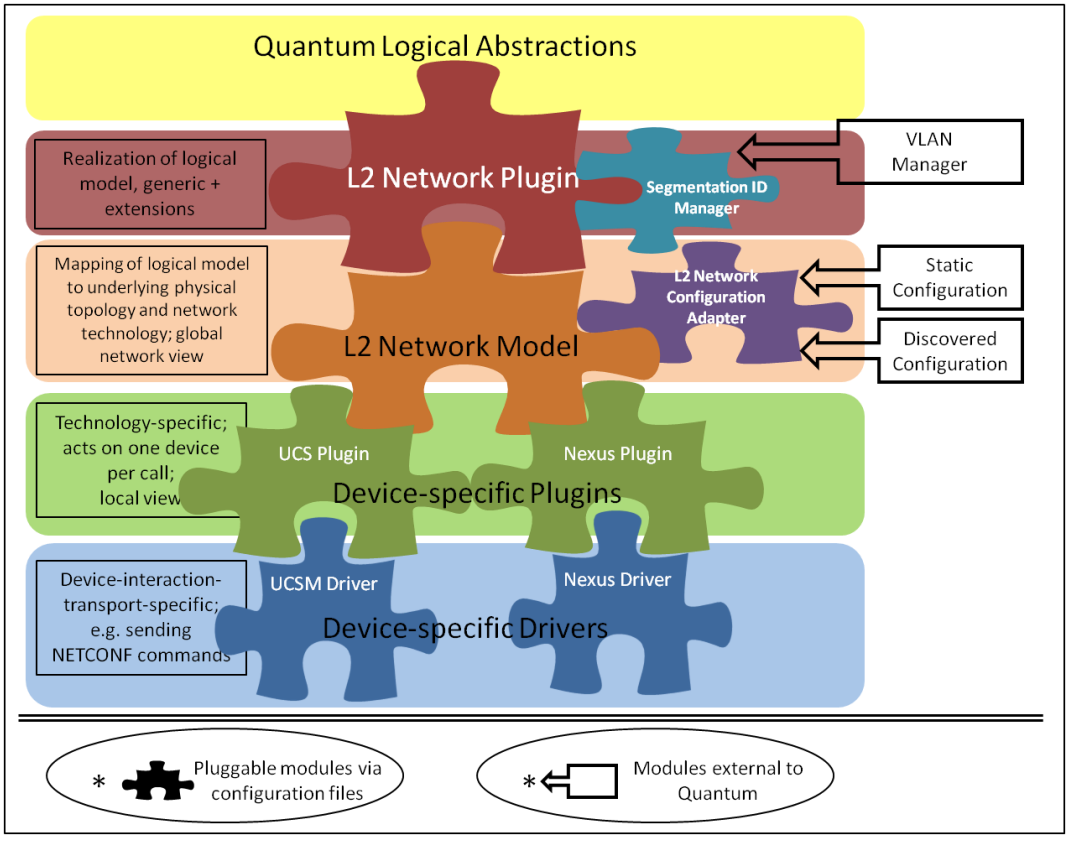
Objective

A L2 network may have multiple switches (physical and virtual), and these switches could be a combination of different types/technologies. Moreover, the L2 network consisting of these switches and the end hosts may be connected in one of several possible topologies.

This is usually a hard problem to solve in a generic way. We propose a plugin framework which can handle these generic requirements, and provide a reference implementation of this plugin framework such that it can be extended and customized.

Design

The above objectives can be achieved with a hierarchical and pluggable architecture. The diagram below shows this hierarchy of pluggable modules.



The description below takes the example of Cisco UCS, and Nexus switches to explain the above architecture.

- The Quantum logical abstractions refer to the network, ports, and virtual interface resources as defined and exposed by the community.

- L2-Network Plugin: The base plugin module which maintains the state of the core and extended resources in the system. It relies on the underlying network model to map the logical network to the physical network. It relies on a Segmentation ID Manager to obtain the segmentation ID information for a particular tenant and for a particular network.

- Segmentation ID Manager: Serves as an adapter to the manager of a particular segmentation scheme; for instance to interface with an external VLAN Manager module.

```
class SegementationIDManager(object):

    @abstractmethod
    def reserve_segmentation_id(self, tenant_id, **kwargs):

    Returns:
    Segmentation scheme
    Segementation ID

    @abstractmethod
    def release_segmentation_id(self, tenant_id, **kwargs):

    Returns:
    Success or failure
```

- L2-Network Model: It relies on a pluggable network configuration module[1] to gather knowledge of the system, but knows which device-specific plugins to invoke for a corresponding core API call, and what parameters to pass to that plugin. These parameters might be different for different plugins. For instance, a call to create_network() of the UCS plugin might require the device IP address, the VLAN-name, and VLAN-ID. However, the same call to the Nexus-plugin will require passing the physical port details as well, in addition to the earlier parameters. It might also not be required to invoke each device-specific plugin for every call; for instance the create_port() call might not require the participation of the Nexus-plugin. Since, this model is written for a specific deployment scenario, it will have the knowledge to make the appropriate calls with reference to the devices and technologies in that deployment. The following base interface is

[1] Note that the information to construct or discover the network configuration can be abstracted away and may reside in external systems which this adapter module helps to interface with.

proposed for the network model to facilitate the L2-Network plugin to delegate the core and extension API calls to the model.

```
class L2NetworkModelBase(object):

    @abstractmethod
    def get_all_networks(self, args):

    @abstractmethod
    def create_network(self, args):

    @abstractmethod
    def delete_network(self, args):

    @abstractmethod
    def get_network_details(self, args):

    @abstractmethod
    def rename_network(self, args):

    @abstractmethod
    def get_all_ports(self, args):

    @abstractmethod
    def create_port(self, args):

    @abstractmethod
    def delete_port(self, args):

    @abstractmethod
    def update_port(self, args):

    @abstractmethod
    def get_port_details(self, args):

    @abstractmethod
    def plug_interface(self, args):

    @abstractmethod
    def unplug_interface(self, args):
```

The L2-Network Model is a pluggable module implementing the above base definition and can be swapped via configuration.

- Device-category-specific Plugins: The network model relies on device-category-specific plugins to perform the configuration on each device. For example, we have a UCS plugin, and a Nexus plugin. These second level of plugins know how to realize the core abstractions and extensions for a particular device/technology. For instance, in the case of the UCS plugin, the create-network operation results in creating a VLAN in UCSM. Similarly, in the case of the Nexus plugin it results in the creation of a VLAN on the Nexus switch and also the configuration of specific interfaces to allow that VLAN. These plugins manage the device-specific

state. The following base interface is proposed for these plugins to facilitate the L2-Network Model to delegate the core and extension API calls to these plugins.

```
class L2DevicePluginBase(object):

    @abstractmethod
    def get_all_networks(self, tenant_id, **kwargs):

    @abstractmethod
    def   create_network(self,   tenant_id,   net_name,   net_id,
segmentation_scheme, segmentation_id, **kwargs):

    @abstractmethod
    def delete_network(self, tenant_id, net_id, **kwargs):

    @abstractmethod
    def get_network_details(self, tenant_id, net_id, **kwargs):

    @abstractmethod
    def   rename_network(self,   tenant_id,   net_id,   new_name,
**kwargs):

   @abstractmethod
   def get_all_ports(self, tenant_id, net_id, **kwargs):

   @abstractmethod
   def create_port(self, tenant_id, net_id, port_state, port_id,
**kwargs):

   @abstractmethod
   def delete_port(self, tenant_id, net_id, port_id, **kwargs):

   @abstractmethod
   def update_port(self, tenant_id, net_id, port_id, port_state,
**kwargs):

   @abstractmethod
   def   get_port_details(self,   tenant_id,   net_id,   port_id,
**kwargs):

   @abstractmethod
   def   plug_interface(self,   tenant_id,   net_id,   port_id,
remote_interface_id, **kwargs):

   @abstractmethod
   def   unplug_interface(self,   tenant_id,   net_id,   port_id,
**kwargs):
```

- Device-category-specific Drivers: The device-category-specific plugins in turn use drivers to communicate with the actual devices. For instance, the UCS plugin uses a UCSM driver to communicate with the UCSM using XML API over HTTP,

whereas the Nexus driver uses NETCONF over ssh. Drivers are configured via configuration files, and one driver can be replaced via another. We do not propose a generic base definition for the drivers, since this is very device/technology specific. There is a tight coupling between the device-specific plugin and the device-specific driver. A new driver has to make sure that this coupling is maintained (and it honors the expected contract).