Blueprint for Dynamic Network Resource Management

Scope:

This blueprint proposes the addition to OpenStack of a framework for dynamic network resource management (DNRM). This framework includes a new OpenStack resource management and provisioning service, a refactored scheme for Neutron API extensions, a policy-based resource allocation system, and dynamic mapping of resources to plugins. It is intended to address a number of use cases, including multivendor environments, policy-based resource scheduling, and virtual appliance provisioning. We are proposing this as a single blueprint in order to create an efficiently integrated implementation. We intend to demonstrate a proof-of-concept in Hong Kong, and to develop this work for inclusion in Icehouse.

The motivation for this work can best be summarized by an RFP from a large customer. They wanted to be able to deploy a scale-out (load-balanced) application pattern in both dev/test and production. For production deployments, each LBaaS VIP request would be assigned to a slice of one of their fleet of hardware load balancers from vendor A. For dev/test, each VIP would be handled by a virtual appliance from vendor B. The dev/test workloads varied significantly, and so virtual appliances should be provisioned only as needed. They were also unhappy with the isolation provided by the L3 agent, and wanted the option of deploying individual vRouters. And they wanted to do all of this by using the native APIs of OpenStack, rather than interacting directly with the CLI or API for each vendor's device.

Many early deployments of OpenStack were "greenfield" systems, in which the infrastructure was intentionally homogeneous. Recently we have seen an increased interest in using OpenStack to automate the management of existing enterprise data centers containing compute, network and storage infrastructure of different types from different vendors. The addition of fiber channel SAN support for Cinder is a good example of this trend.

The increase in heterogeneity is not limited to physical infrastructure. There is significant interest in using virtual network appliances – virtual machines running dedicated network functions in software – as well as traditional networking gear. (In the telco space, this is referred to as NFV – network function virtualization.) The cost benefits and relative ease of deploying such virtual appliances will increase the diversity of network resources in these data centers. And with the increasing variety of network resources comes the need for a mechanism to select the best resource to fulfill each request.

Just as the Nova scheduler selects the best VM to satisfy a request for an instance, we need a network resource scheduler that has access to the complete pool of network resources. And because this mechanism is used to choose a resource – and hence the plugin which supports it – it cannot be implemented inside any one plugin.

Populating such a resource pool is not straightforward. For physical resources – a dedicated box, or a slice of a multitenant box – traditional discovery mechanisms may suffice. However virtual network appliances are a different matter. Today, each vendor who provides a virtual appliance for OpenStack is forced to implement a private scheme for provisioning VM instances and communicating their existence to their Neutron plugin. There is no standard mechanism in OpenStack for working with so-called “service instances” – VMs managed by OpenStack as part of the infrastructure or OpenStack core. (The OOO project is addressing some of these concerns, but will not meet the needs of DNRM.)

Since we need a way of managing resource pools in a centralized manner (i.e. not embedded in any plugin), we feel that it makes sense to offer a shared service for network resource provisioning and discovery. This makes it possible to implement dynamic (elastic) provisioning for virtual appliances. It allows for the easy packaging and installation of a new virtual appliance, by bundling a Glance image, a plugin, and a provisioning policy.

Use Cases:

DNRM enables a large number of L3-L7 Use Cases, only some of which are enumerated here. The first seven are written in terms of load balancing (LB) functions, to illustrate how the various aspects of DNRM contribute to the various scenarios. We have also included two L3 Router use cases. However the DNRM framework is intended to address FWaaS, VPNaaS, and other security functions. DNRM does not address the L2 use cases that are the subject of the Neutron Modular L2 (ML2) work.

LBaaS Use Cases

1. User sets up LB by calling the Neutron LBaaS Extension API to create a Pool, and then binding a VIP to the Pool. The VIP is allocated by selecting one of a number of physical LBs from vendor X. (Each LB may be implemented as a slice of a multitenant device.) [This is feasible today, but only if the allocation mechanism is embedded in the Neutron plugin supplied by vendor X.]
2. User sets up LB as above. For tenant A, the VIP should be allocated from the set of physical LBs from vendor X. For all other tenants, the VIP should be allocated from the set of physical LBs from vendor Y. Vendors X and Y have each created their own plugin.
3. User sets up LB as above. For tenant A, the VIP should be allocated from the set of physical LBs from vendor X. For all other tenants, the VIP should be allocated from the available set of virtual LB appliances from vendor Y.
4. User sets up LB as above. For all tenants, the VIP should, if possible, be allocated from the set of physical LBs from vendor X. If no physical LBs are available, a virtual LB from vendor Y

should be used instead.

5. User sets up LB as above. For all tenants, the VIP should be allocated from the set of virtual LBs from vendor Y. The selection should be based on the execution of a black-box scheduler (policy engine) that can take into account quotas, costs, network locality, current traffic, and other factors. The policy engine implementation is independent of any particular vendor or plugin.
6. User sets up LB as above. For all tenants, the VIP should be allocated from either the set of physical LBs from vendor X, or the set of virtual LBs from vendor Y. The selection should be based on the execution of the scheduler.
7. An OpenStack installation is configured to use virtual LBs from vendor Y for all LBaaS VIP requests. Normally, a fleet of 16 virtual LBs is sufficient for operations. However during an unscheduled service disruption, the development team decides to run a large number of tests (up to 64), each of which requires a VIP. The system should automatically provision addition virtual LBs to meet the spike in demand, and should release them when the demand returns to normal.

L3 Use Cases

1. For compliance reasons, a customer requires that every L3 router correspond to a physical or virtual appliance with limited network access. Instead of using the default OpenStack L3 agent, every L3 *create-router* request is associated with a dedicated virtual router appliance running in a VM.
2. A customer has configured a fleet of physical routers from vendor A to handle DC-to-DC traffic. Every L3 *create-router* request with an *ExternalGateway* corresponding to the DC-DC networks is associated with one of the physical routers from vendor A. All other create-router requests are mapped to a dedicated virtual router appliance from vendor B running in a VM.

Implementation Overview:

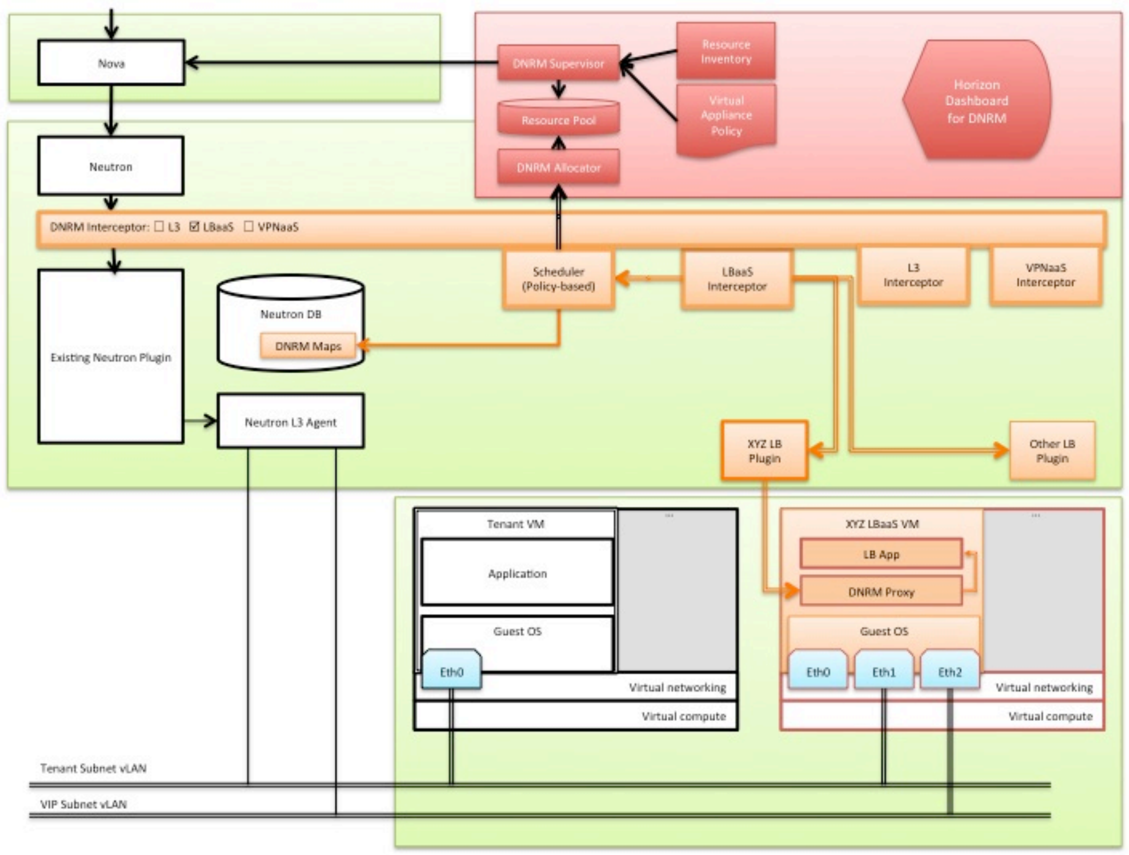
DNRM is a complex extension to the OpenStack architecture. There are at least three major areas of implementation:

The Resource Manager.

This is the service, shown in red on the diagram below, which manages the pool of network resources and makes them available for use by the Interceptor. It includes the Supervisor, which populates the resource pool, and the Allocator, which implements a REST API for querying the state of the pool and allocating and freeing resource instances. There is also a Horizon dashboard to allow the OpenStack operator to monitor and control the system. The Supervisor is responsible for provisioning virtual appliances as needed by calling Nova to instantiate VMs using the appropriate images.

The Interceptor.

This includes a shim which is injected between Neutron and the existing Neutron Plugin. For every API call, the Interceptor either passes it through to the existing Plugin or diverts it to the appropriate Interceptor subsystem. If no Interceptors are enabled, all traffic is simply passed through.



Each Interceptor subsystem performs the entire device-independent workflow associated with the API extension. (For example, in LBaaS this includes creating and managing Pools.) For those API calls that involve the allocation of a resource (e.g. *Create VIP* in LBaaS), the LBaaS Interceptor calls the Scheduler to request a resource allocation and mapping. The Scheduler uses the Allocator REST API to find candidate resources and reserve the instance that fits best. The resource description includes type information that is used to identify the plugin that will handle the device-dependent processing. (See the Data Model Changes section below.)

DNRM Plugin

A DNRM plugin is similar to a traditional OpenStack plugin. The API from Interceptor to Plugin includes both the original Neutron API parameters and the resource description for the allocated instance.

The plugin uses this information (specifically the Address and Slice) to communicate with the specific resource. The protocol used between plugin and resource is not specified; it's up to the vendor. In the diagram, we show one possible implementation model, using a proxy running in an appliance VM to translate OpenStack API requests into local configuration actions.

As shown above, the DNRM architecture maintains a reasonable level of decoupling between the Neutron/Interceptor/Plugin layer and the Resource Manager. One potential source of complication involves dynamic vNIC configuration for virtual appliances. Today, the only way that these functions are exposed is through the Nova *interface-attach* and *interface-detach* functions. We need to examine the consequences of using this API calls from the Plugin. (It may be easier to do it from the VM itself.)

For simplicity, the architecture diagram does not show the use of DNRM to manage physical network resources.

Data Model Changes:

There are a number of additions to the existing data models. The most obvious is the resource description which forms the basis of the resource pool and the REST API for the Allocator. Here's a first attempt:

Property Name	Type	Role	Usage
ResClass	String enum	One of "L3", "LB", "FW", "VPN". The set may be extended as new Neutron extensions are introduced.	Immutable; shared by all resources in a class
ResType	String	Identifies the plugin which handles this resource. Managed by resource provider or vendor, e.g. `"com.brocade.vyatta.6000"`	Immutable; shared by all resources of the same type
Description	String (opt)	Human-readable description, e.g. `"Vyatta 6000 vRouter"`	Immutable; shared by all resources of the same type
ResID	UUID	Durable ID for resource	Immutable; unique to a resource instance; never re-used. Where possible, the Interceptor should use the ResID to identify the domain-specific resource identifier (e.g. Router, Pool, VIP), so that the operator can associate a Neutron API request with any updates to the DNRM Resource Pool.
Address	String (usually	Address by which the plugin may communicate with the resource instance	Immutable; assigned by Supervisor at resource provisioning or discovery time

	IP address)		
VM	UUID (opt)	For resources that are provisioned as Nova VMs, the VM ID as returned by Nova	Immutable; assigned by Supervisor at resource provisioning time. If null, this resource is either a physical device, or it is a virtual appliance that is not provisioned through the Resource Manager.
Slice	String (opt)	Used by the plugin to distinguish between multiple resource instances that share the same Address	Immutable; assigned by Supervisor at resource provisioning or discovery time
AvZone	String (opt)	If present, the resource is only usable in the specified availability zone	Immutable; assigned by Supervisor at resource provisioning or discovery time
HostAgg	String (opt)	If present, the resource is only usable in the specified host aggregate	Immutable; assigned by Supervisor at resource provisioning or discovery time
Cost	String (opt)	If present, this string is used by the Interceptor policy engine as one of the inputs to the algorithm to determine which resource should be selected from the available set.	Assigned by the Supervisor.
Seq	uint	Incremented by the Allocator whenever a resource object is changed (transactionally). Every Update request must include the last Seq seen by the client; the Allocator will reject any request if the Seq values do not match.	Mutable by the Allocator
State	String enum	One of “allocated”, “free”, “locked”.	Mutable by the allocator.
Project	UUID(?) (opt)	When a resource is “allocated”, the Project identifies the OpenStack project associated with the resource allocation.	Supplied by the Allocator client as part of an AllocateResource request

There are several other important tables to be added to the Neutron data store. One maps resource IDs to the allocated resource instances (including full descriptions). A second maps each *ResType* attribute to the plugin which handles resources of that type.

Configuration variables:

There are many elements of DNRM that will require configuration. Of these, the most important is a set of Booleans which specifies for each Neutron API Extension [better term needed] whether requests are handled by the Interceptor or passed through to the existing Neutron plugin. If all are false, the Interceptor becomes a no-op.

API's:

TBD

Plugin Interface:

Existing plugins should work fine. This project introduces a new type of plugin, which includes just the resource-specific aspects of an API.

We may need a new terminology to distinguish between the various plugins, agents, handlers, proxies, etc.

Required Plugin support:
TBD

Dependencies:
TBD

CLI Requirements:
TBD

Horizon Requirements:
The Resource Manager will have a Horizon dashboard which allows the cloud operator to monitor the allocation of resources and set allocation policies. A future enhancement might be to provide a Horizon UI to the Scheduler’s policy engine.

Usage Example:
The goal of this project is that there should be no need for any changes to the usage of the OpenStack system: the existing APIs should work correctly, and abstract away all of the complexities associated with multivendor configurations and the differences between physical and virtual resources.

Test Cases:
There will be many.