Multiple L3 Backends in a cloud

Manjeet Singh Bhatia<manjeet.s.bhatia@intel.com>(Intel),
Isaku yamahata<isaku.yamahata@intel.com>(Intel)
Takashi Yamamoto<yamamoto@midokura.com>(Midokura)

OpenStack Summit Vancouver: May 22, 2018

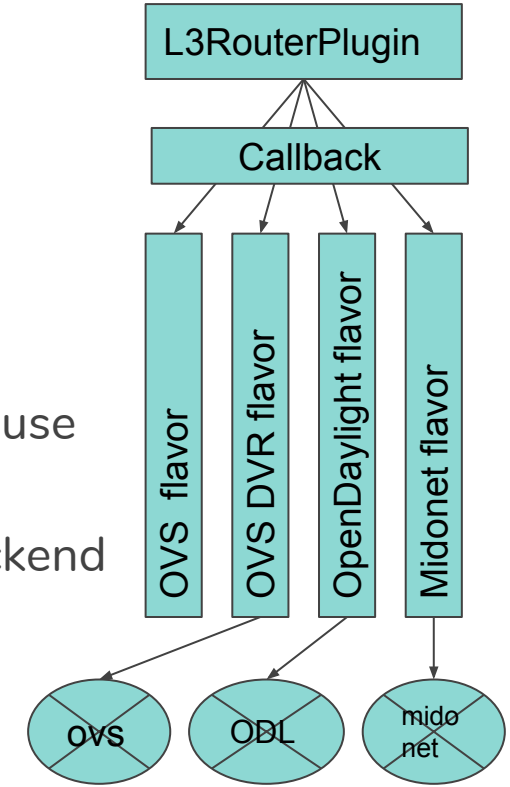
Agenda

- Neutron L3 flavors framework
- Why L3 flavors ?
- Use case
- Driver Enabling
- Sample L3 driver for a backend.
- Traffic b/w different backends
- Datapath connectivity among backends (pie in the sky)
- Challenges
- Summary

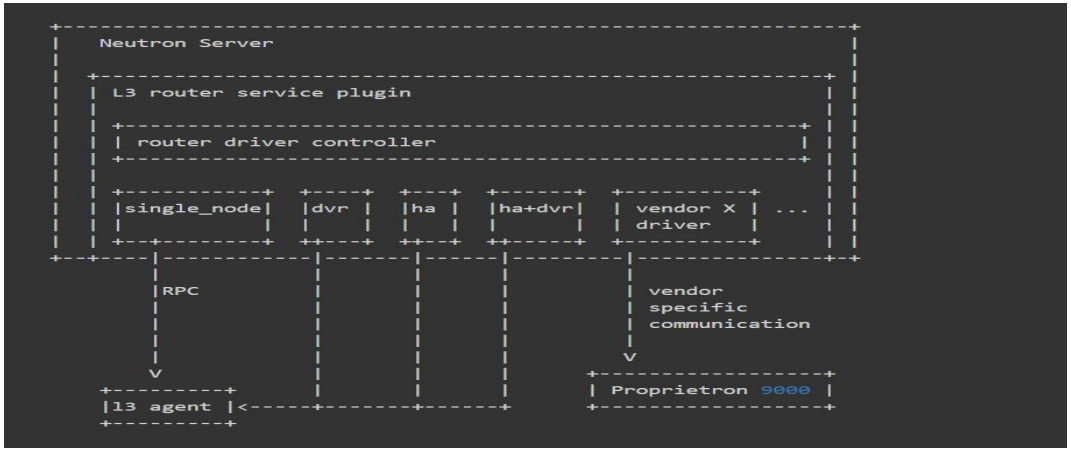
Neutron L3 flavor framework

L3 Flavor Framework:

- Single L3 Router Plugin with flavor support
  - with Neutron callbacks
  - Instead of backend specific L3 plugin
- Allows multiple L3 backends
  - User specifies flavor which L3 backends to use
  - Backends implements L3 flavor driver
- Router instance is associated with flavor=l3 backend



Neutron L3 Flavors framework !

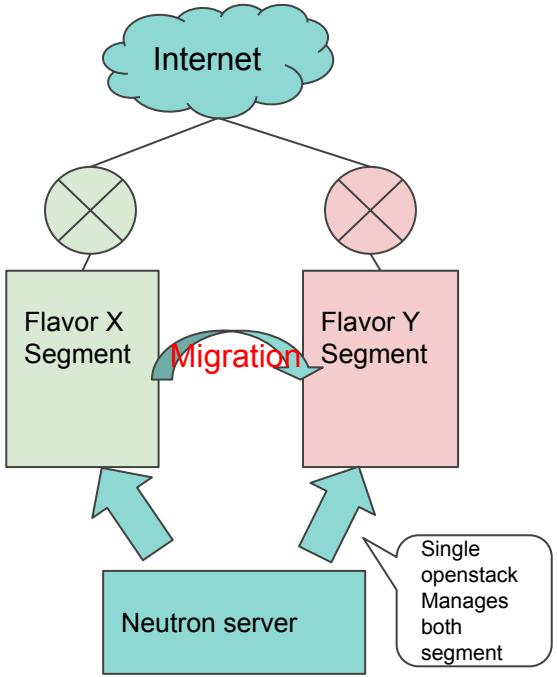


- Source: Neutron L3 Flavors Framework spec
- https://specs.openstack.org/openstack/neutron-specs/specs/newton/multi-l3-backends.html

Neutron L3 Flavors continued

- Neutron Flavors enables multiple L3 backends.
- Driver X can be used for subset of routers and Driver Y for another set of routers.
- Its similar to ML2 but there’s an important difference.

Use cases



- Multiple backends in a single Neutron deployment, each has its own logical network topology, completely separated each other
- It would allow incremental migrations from one backend to the other

Another motivation: Simplification

- DB transaction Issue
  - L2 plugin has its own db transactions
  - L2 plugin, e.g. create_port, shouldn't be called within a db transaction of L3 plugin
- Implementation consistency and code reduction
  - The reference L3 plugin has been refactored to avoid the above mentioned transaction issues.
  - It's better for vendors to use the same framework instead of keeping to improve their own monolithic L3 plugins.

How to use vendor L3 flavor

- Use **router** as L3 service_plugin
- Specify your flavor as a **L3_ROUTER_NAT** service provider

service_plugins = router, xyz, ...

[service_providers]

service_provider = L3_ROUTER_NAT:ODL:networking_odl.l3.l3_flavor.ODLL3ServiceProvider:default

How to use L3 flavor (Cont.)

Prepare a flavor and its profile

1. `openstack network flavor profile create --driver networking_odl.l3.l3_flavor.ODLL3ServiceProvider`
2. `openstack network flavor create --service-type L3_ROUTER_NAT odl`
3. `openstack network flavor add profile odl <flavorprofileid>`

Create a router with the flavor

4. `neutron router-create router1 --flavor odl`

Sample L3 flavor driver

```python
import copy

import six

from neutron_lib.callbacks import events
from neutron_lib.callbacks import priority_group
from neutron_lib.callbacks import registry
from neutron_lib.callbacks import resources
from neutron_lib import constants as q_const
from neutron_lib.plugins import constants as plugin_constants
from neutron_lib.plugins import directory
from oslo_log import helpers as log_helpers
from oslo_log import log as logging

from neutron.objects import router as l3_obj
from neutron.services.l3_router.service_providers import base

from networking_odl.common import constants as odl_const
from networking_odl.journal import full_sync
from networking_odl.journal import journal


LOG = logging.getLogger(__name__)
L3_PROVIDER = 'networking_odl.l3.l3_flavor.ODLL3ServiceProvider'


L3_RESOURCES = {
    odl_const.ODL_ROUTER: odl_const.ODL_ROUTERS,
    odl_const.ODL_FLOATINGIP: odl_const.ODL_FLOATINGIPS
}


@registry.has_registry_receivers
class ODLL3ServiceProvider(base.L3ServiceProvider):
    @log_helpers.log_method_call
    def __init__(self, l3_plugin):
        super(ODLL3ServiceProvider, self).__init__(l3_plugin)
        self.journal = journal.OpenDaylightJournalThread()
        # TODO(yamahata): add method for fullsync to retrieve
        # all the router with odl service provider.
        # other router with other service provider should be filtered.
        full_sync.register(plugin_constants.L3, L3_RESOURCES)

    def _get_kwargs(self, kwargs):
        return kwargs['context'], kwargs['router']

    def _validate_l3_flavor(self, context, router_id):
        if router_id is None:
            return False
        router = l3_obj.Router.get_object(context, id=router_id)
        flavor_plugin = directory.get_plugin(plugin_constants.FLAVORS)
        flavor = flavor_plugin.get_flavor(context, router.flavor_id)
        provider = flavor_plugin.get_flavor_next_provider(
            context, flavor['id'])[0]
        return str(provider['driver']) == L3_PROVIDER
```

```python
    @registry.receives(resources.ROUTER, [events.AFTER_CREATE],
                       priority_group.PRIORITY_ROUTER_DRIVER)
    @log_helpers.log_method_call
    def _router_create_postcommit(self, resource, event, trigger, **kwargs):
        self.journal.set_sync_event()

    @registry.receives(resources.ROUTER_CONTROLLER,
                       [events.PRECOMMIT_ADD_ASSOCIATION])
    @log_helpers.log_method_call
    @journal.call_thread_on_end
    def _router_add_association(self, resource, event, trigger, **kwargs):
        old_drv = kwargs['old_driver']
        new_drv = kwargs['new_driver']
        context, router_dict = self._get_kwargs(kwargs)
        router_dict['gw_port_id'] = kwargs['router_db'].gw_port_id
        if old_drv == new_drv:
            router_id = kwargs['router_id']
            journal.record(context, odl_const.ODL_ROUTER,
                           router_id, odl_const.ODL_UPDATE, router_dict)
        else:
            # TODO(yamahata): revise this.
            journal.record(context, odl_const.ODL_ROUTER, router_dict['id'],
                           odl_const.ODL_CREATE, router_dict)
            # TODO(yamahata): process floating ip etc. or just raise error?

    @registry.receives(resources.ROUTER, [events.PRECOMMIT_UPDATE],
                       priority_group.PRIORITY_ROUTER_DRIVER)
    @log_helpers.log_method_call
    def _router_update_precommit(self, resource, event, trigger, **kwargs):
        # NOTE(manjeets) router update bypasses the driver controller
        # and argument type is different.
        payload = kwargs.get('payload', None)
        if payload:
            router_id = payload.states[0]['id']
            router_dict = payload.request_body
            if 'gw_port_id' not in router_dict:
                router_dict['gw_port_id'] = payload.states[0]['gw_port_id']
            if self._validate_l3_flavor(payload.context, router_id):
                journal.record(payload.context, odl_const.ODL_ROUTER,
                               router_id, odl_const.ODL_UPDATE, router_dict)
```

Sample L3 flavor Driver

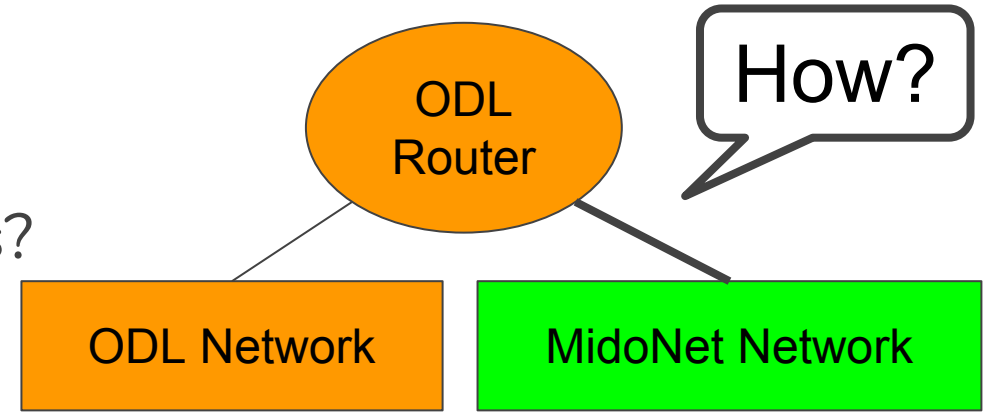
```
class ODLL3ServiceProvider(base.L3ServiceProvider):
    @log_helpers.log_method_call
    def __init__(self, l3_plugin):
        super(ODLL3ServiceProvider, self).__init__(l3_plugin)
        self.journal = journal.OpenDaylightJournalThread()
        full_sync.register(plugin_constants.L3, L3_RESOURCES)

    def _validate_l3_flavor(self, context, router_id):
        "implementation -----------------"

    @registry.receives(resources.ROUTER, [events.PRECOMMIT_CREATE],
                       priority_group.PRIORITY_ROUTER_DRIVER)
    @log_helpers.log_method_call
    def _router_create_precommit(self, resource, event, trigger, **kwargs):
        "implementation -----------------"

    @registry.receives(resources.ROUTER, [events.AFTER_CREATE],
                       priority_group.PRIORITY_ROUTER_DRIVER)
    @log_helpers.log_method_call
    def _router_create_postcommit(self, resource, event, trigger, **kwargs):
        "implementation -----------------"

    @registry.receives(resources.ROUTER_CONTROLLER,
                       [events.PRECOMMIT_ADD_ASSOCIATION])
    @log_helpers.log_method_call
    @journal.call_thread_on_end
    def _router_add_association(self, resource, event, trigger, **kwargs):
        "implementation -----------------"

    @registry.receives(resources.ROUTER, [events.PRECOMMIT_UPDATE],
                       priority_group.PRIORITY_ROUTER_DRIVER)
    @log_helpers.log_method_call
    def _router_update_precommit(self, resource, event, trigger, **kwargs):
        "implementation -----------------"

    @registry.receives(resources.ROUTER, [events.AFTER_UPDATE],
                       priority_group.PRIORITY_ROUTER_DRIVER)
    @log_helpers.log_method_call
    def _router_update_postcommit(self, resource, event, trigger, **kwargs):
        "implementation -----------------"

    @registry.receives(resources.ROUTER_CONTROLLER,
                       [events.PRECOMMIT_DELETE_ASSOCIATIONS])
    @log_helpers.log_method_call
    def _router_del_association(self, resource, event, trigger, **kwargs):
        "implementation -----------------"

    @registry.receives(resources.ROUTER, [events.PRECOMMIT_DELETE],
                       priority_group.PRIORITY_ROUTER_DRIVER)
    @log_helpers.log_method_call
    def _router_delete_precommit(self, resource, event, trigger, **kwargs):
        "implementation -----------------"

    @registry.receives(resources.ROUTER, [events.AFTER_DELETE],
                       priority_group.PRIORITY_ROUTER_DRIVER)
    @log_helpers.log_method_call
    def _router_delete_postcommit(self, resource, event, trigger, **kwargs):
        "implementation -----------------"
        self.journal.set_sync_event()

    @registry.receives(resources.FLOATING_IP, [events.PRECOMMIT_CREATE])
    @log_helpers.log_method_call
    @journal.call_thread_on_end
    def _floatingip_create_precommit(self, resource, event, trigger, **kwargs):
        "implementation -----------------"

    @registry.receives(resources.FLOATING_IP, [events.PRECOMMIT_UPDATE])
    @log_helpers.log_method_call
    @journal.call_thread_on_end
    def _floatingip_update_precommit(self, resource, event, trigger, **kwargs):
        "implementation -----------------"

    @registry.receives(resources.FLOATING_IP, [events.PRECOMMIT_DELETE])
    @log_helpers.log_method_call
    @journal.call_thread_on_end
    def _floatingip_delete_precommit(self, resource, event, trigger, **kwargs):
        "implementation -----------------"
```

Traffic between multiple backends: Pie in the sky

- East-west traffic between multiple L3 backends
- API wise, shared router connected to each L3 network or L2GW?
- Implementation wise: requires common router or gateway

Opens:

- Any requirements?
- Volunteers?



Traffic between multiple backends

- The simplest solution: Disallow such configurations
  - You can still provide connectivities using the other mechanisms.
- Use legacy L3-agent compatible port
  - Hopefully many of backends can support it trivially
- Design something distributed
  - Pie in the sky
  - More work for dubious usefulness
  - It’s actually more complicated
  - Floating-IP, A network can be backed by multiple backends, Live migration between backends (multiple port binding), Hierarchical port binding

Challenges

- There were missing notifications in neutron (needed a fix)
- Callback execution order was not guaranteed.
- Changes to neutron and neutron-lib.

Future work

- FloatingIP compatibility
  - Compatibility between L3 flavor and ML2 mech driver
- More tests. Tempest
- Tenants associated to l3 flavor
  - New tenants/user to use new backends
  - Existing tenants to use the the existing backend for migration

Summary

- L3 flavor works and L3 flavor drivers are coming

Call for action

- test/use it
- Convert your L3 plugin into L3 flavor driver

Reference Code

1. https://review.openstack.org/#/c/523257/
   (Adding callbacks to neutron)
2. https://review.openstack.org/#/c/504182/
   (ODL L3 Flavor Driver)
3. https://review.openstack.org/#/c/544116/
   (Functional tests)
4. https://review.openstack.org/#/c/483174/
   (MidoNet L3 Flavor Driver)