Networking SFC Documentation

Release 19.0.0.0rc2.dev3

OpenStack Foundation

Sep 23, 2024

CONTENTS

1 Service Function Chaining Extension for OpenStack Networking 1
- 1.1 Team and repository tags . . . 1
- 1.2 Service Function Chaining API . . . 1
- 1.3 Features . . . 2
- 1.4 Service Function Chaining Key Contributors . . . 2
- 1.5 Background on the Subject of Service Function Chaining . . . 3

2 Install Guide 4
- 2.1 Installation . . . 4
- 2.2 Configuration . . . 4
  - 2.2.1 Controller nodes . . . 4
  - 2.2.2 Compute nodes . . . 5
  - 2.2.3 Database setup . . . 5

3 Using the Service Function Chaining 6
- 3.1 Usage . . . 6
- 3.2 Command extension . . . 6
  - 3.2.1 List of New Neutron CLI Commands: . . . 6

4 Configuration Guide 8
- 4.1 Configuration . . . 8
  - 4.1.1 networking-sfc.conf . . . 8
  - 4.1.2 Sample networking-sfc.conf . . . 9
- 4.2 Policy . . . 11
  - 4.2.1 networking-sfc policies . . . 11
  - 4.2.2 Sample networking-sfc Policy File . . . 14

5 Contributor Guide 17
- 5.1 Programming HowTos and Tutorials . . . 17
  - 5.1.1 Contribution . . . 17
  - 5.1.2 Alembic-migration . . . 17
- 5.2 Networking-SFC Internals . . . 19
  - 5.2.1 API Model . . . 19
  - 5.2.2 System Design and Workflow . . . 31
  - 5.2.3 OVS Driver and Agent Workflow . . . 36
  - 5.2.4 Networking-sfc / OVN Driver . . . 42
  - 5.2.5 OVS Driver and Agent for Symmetric Port Chains . . . 46
  - 5.2.6 Service Function Tap for Port Chains . . . 49
  - 5.2.7 Non-Transparent Service Functions for Port Chains . . . 53

5.2.8 IETF SFC Encapsulation 57
5.2.9 Exclusive Port-Pair Group for Non-Transparent Service Functions 66

CHAPTER

ONE

SERVICE FUNCTION CHAINING EXTENSION FOR OPENSTACK NETWORKING

1.1 Team and repository tags

1.2 Service Function Chaining API

This project provides APIs and implementations to support Service Function Chaining in Neutron.

Service Function Chaining is a mechanism for overriding the basic destination based forwarding that is typical of IP networks. It is conceptually related to Policy Based Routing in physical networks but it is typically thought of as a Software Defined Networking technology. It is often used in conjunction with security functions although it may be used for a broader range of features. Fundamentally SFC is the ability to cause network packet flows to route through a network via a path other than the one that would be chosen by routing table lookups on the packets destination IP address. It is most commonly used in conjunction with Network Function Virtualization when recreating in a virtual environment a series of network functions that would have traditionally been implemented as a collection of physical network devices connected in series by cables.

A very simple example of a service chain would be one that forces all traffic from point A to point B to go through a firewall even though the firewall is not literally between point A and B from a routing table perspective.

A more complex example is an ordered series of functions, each implemented in multiple VMs, such that traffic must flow through one VM at each hop in the chain but the network uses a hashing algorithm to distribute different flows across multiple VMs at each hop.

This is an initial release, feedback is requested from users and the API may evolve based on that feedback.

- Free software: Apache license
- Source: https://opendev.org/openstack/networking-sfc
- Documentation: https://docs.openstack.org/networking-sfc/latest
- Overview: https://launchpad.net/networking-sfc
- Bugs: https://bugs.launchpad.net/networking-sfc
- Blueprints: https://blueprints.launchpad.net/networking-sfc
- Wiki: https://wiki.openstack.org/wiki/Neutron/ServiceInsertionAndChaining

- Release notes: https://docs.openstack.org/releasenotes/networking-sfc/

1.3 Features

- Creation of Service Function Chains consisting of an ordered sequence of Service Functions. SFs are virtual machines (or potentially physical devices) that perform a network function such as firewall, content cache, packet inspection, or any other function that requires processing of packets in a flow from point A to point B.
- Reference implementation with Open vSwitch
- Flow classification mechanism (ability to select and act on traffic)
- Vendor neutral API
- Modular plugin driver architecture

1.4 Service Function Chaining Key Contributors

- Cathy Zhang (Project Lead): https://launchpad.net/~cathy-h-zhang
- Louis Fourie: https://launchpad.net/~lfourie
- Paul Carver: https://launchpad.net/~pcarver
- Vikram: https://launchpad.net/~vikschw
- Mohankumar: https://blueprints.launchpad.net/~mohankumar-n
- Rao Fei: https://launchpad.net/~milo-frao
- Xiaodong Wang: https://launchpad.net/~xiaodongwang991481
- Ramanjaneya Reddy Palleti: https://launchpad.net/~ramanjieee
- Stephen Wong: https://launchpad.net/~s3wong
- Igor Duarte Cardoso: https://launchpad.net/~igordcard
- Prithiv: https://launchpad.net/~prithiv
- Akihiro Motoki: https://launchpad.net/~amotoki
- Swaminathan Vasudevan: https://launchpad.net/~swaminathan-vasudevan
- Armando Migliaccio https://launchpad.net/~armando-migliaccio
- Kyle Mestery https://launchpad.net/~mestery

1.5 Background on the Subject of Service Function Chaining

- Original Neutron bug (request for enhancement): https://bugs.launchpad.net/neutron/+bug/1450617
- https://blueprints.launchpad.net/neutron/+spec/neutron-api-extension-for-service-chaining
- https://blueprints.launchpad.net/neutron/+spec/common-service-chaining-driver-api
- https://wiki.opnfv.org/display/VFG/Openstack+Based+VNF+Forwarding+Graph

CHAPTER

TWO

INSTALL GUIDE

2.1 Installation

If possible, you should rely on packages provided by your Linux and/or OpenStack distribution:

- For Fedora or CentOS, you can install the `python-networking-sfc` RPM package provided by the RDO project.

If you use `pip`, follow these steps to install networking-sfc:

- identify the version of the networking-sfc package that matches your OpenStack version:
  - 2023.1 Antelope: latest 16.0.x version
  - Zed: latest 15.0.x version
  - Yoga: latest 14.0.x version
- indicate pip to (a) install precisely this version and (b) take into account OpenStack upper constraints on package versions for dependencies (example for Antelope):

```
pip install -c https://opendev.org/openstack/requirements/raw/branch/
↪stable/2023.1/upper-constraints.txt networking-sfc==16.0.0
```

2.2 Configuration

2.2.1 Controller nodes

After installing the package, enable the service plugins in neutron-server by adding them in `neutron.conf` (typically found in `/etc/neutron/`):

```
[DEFAULT]
service_plugins = flow_classifier,sfc
```

In the same configuration file, specify the driver to use in the plugins. Here we use the OVS driver:

```
[sfc]
drivers = ovs

[flowclassifier]
drivers = ovs
```

After that, restart the neutron-server. In devstack, run:

```
systemctl restart devstack@q-svc
```

In a similar way with systemd setups, you can run:

```
systemctl restart neutron-server
```

2.2.2 Compute nodes

After installing the package, enable the networking-sfc extension in the Open vSwitch agent. The configuration file name can change, the default one is `/etc/neutron/plugins/ml2/ml2_conf.ini`. Add the sfc extension:

```
[agent]
extensions = sfc
```

And restart the neutron-openvswitch-agent process. In devstack, run:

```
systemctl restart devstack@q-agt
```

And with systemd setups you can run:

```
systemctl restart neutron-openvswitch-agent
```

2.2.3 Database setup

The database is the standard Neutron database with a few more tables, which can be configured with `neutron-db-manage` command-line tool:

```
neutron-db-manage --subproject networking-sfc upgrade head
```

CHAPTER THREE

USING THE SERVICE FUNCTION CHAINING

3.1 Usage

To use networking-sfc in a project:

```
import networking_sfc
```

3.2 Command extension

Networking-sfc uses python-neutronclients existing command extension framework for adding required command lines for realizing service function chaining functionality. Refer to Python-neutronclient command extension for further details.

3.2.1 List of New Neutron CLI Commands:

Below listed command lines are introduced for realizing service function chaining.

```
flow-classifier-create   Create a flow-classifier.
flow-classifier-delete   Delete a given flow-classifier.
flow-classifier-list     List flow-classifiers that belong to a given tenant.
flow-classifier-show     Show information of a given flow-classifier.
flow-classifier-update   Update flow-classifier information.

port-pair-create         Create a port-pair.
port-pair-delete         Delete a given port-pair.
port-pair-list           List port-pairs that belongs to a given tenant.
port-pair-show           Show information of a given port-pair.
port-pair-update         Update port-pair's information.

port-pair-group-create   Create a port-pair-group.
port-pair-group-delete   Delete a given port-pair-group.
port-pair-group-list     List port-pair-groups that belongs to a given tenant.
port-pair-group-show     Show information of a given port-pair-group.
port-pair-group-update   Update port-pair-group's information.

port-chain-create        Create a port-chain.
port-chain-delete        Delete a given port-chain.
```

(continues on next page)

(continued from previous page)

```
port-chain-list          List port-chains that belong to a given tenant.
port-chain-show          Show information of a given port-chain.
port-chain-update        Update port-chain's information.
```

CHAPTER

FOUR

CONFIGURATION GUIDE

4.1 Configuration

This section provides a list of all possible options for each configuration file.

networking-sfc uses the following configuration file.

4.1.1 networking-sfc.conf

flowclassifier

drivers

Type
list

Default
['dummy']

An ordered list of flow classifier drivers entrypoints to be loaded from the networking_sfc.flowclassifier.drivers namespace.

quotas

quota_flow_classifier

Type
integer

Default
100

Maximum number of Flow Classifiers per tenant. A negative value means unlimited.

quota_port_chain

Type
integer

Default
10

Maximum number of port chains per tenant. A negative value means unlimited.

`quota_port_pair_group`

Type
integer

Default
`10`

maximum number of port pair group per tenant. a negative value means unlimited.

`quota_port_pair`

Type
integer

Default
`100`

maximum number of port pair per tenant. a negative value means unlimited.

`quota_service_graphs`

Type
integer

Default
`10`

maximum number of Service Graphs per project. a negative value means unlimited.

sfc

`drivers`

Type
list

Default
`['dummy']`

An ordered list of service chain drivers entrypoints to be loaded from the networking_sfc.sfc.drivers namespace.

The following is a sample configuration file for networking-sfc. It is generated from code and reflect the current state of code in the networking-sfc repository.

4.1.2 Sample networking-sfc.conf

This sample configuration can also be viewed in the raw format.

```
[DEFAULT]


[flowclassifier]

#
```

(continues on next page)

(continued from previous page)

```
# From networking-sfc
#

# An ordered list of flow classifier drivers entrypoints to be loaded from the
# networking_sfc.flowclassifier.drivers namespace. (list value)
#drivers = dummy


[quotas]

#
# From networking-sfc.quotas
#

# Maximum number of Flow Classifiers per tenant. A negative value means
# unlimited. (integer value)
#quota_flow_classifier = 100

# Maximum number of port chains per tenant. A negative value means unlimited.
# (integer value)
#quota_port_chain = 10

# maximum number of port pair group per tenant. a negative value means
# unlimited. (integer value)
#quota_port_pair_group = 10

# maximum number of port pair per tenant. a negative value means unlimited.
# (integer value)
#quota_port_pair = 100

# maximum number of Service Graphs per project. a negative value means
# unlimited. (integer value)
#quota_service_graphs = 10


[sfc]

#
# From networking-sfc
#

# An ordered list of service chain drivers entrypoints to be loaded from the
# networking_sfc.sfc.drivers namespace. (list value)
#drivers = dummy
```

4.2 Policy

networking-sfc, like most OpenStack projects, uses a policy language to restrict permissions on REST API actions.

4.2.1 networking-sfc policies

The following is an overview of all available policies in networking-sfc. For a sample configuration file, refer to *Sample networking-sfc Policy File*.

networking-sfc

create_flow_classifier

Default
`rule:regular_user`

Operations

- **POST** `/sfc/flow_classifiers`

Create a flow classifier

update_flow_classifier

Default
`rule:admin_or_owner`

Operations

- **PUT** `/sfc/flow_classifiers/{id}`

Update a flow classifier

delete_flow_classifier

Default
`rule:admin_or_owner`

Operations

- **DELETE** `/sfc/flow_classifiers/{id}`

Delete a flow classifier

get_flow_classifier

Default
`rule:admin_or_owner`

Operations

- **GET** `/sfc/flow_classifiers`
- **GET** `/sfc/flow_classifiers/{id}`

Get flow classifiers

create_port_chain

Default
`rule:regular_user`

Operations

- **POST** `/sfc/port_chains`

Create a port chain

`update_port_chain`

Default
`rule:admin_or_owner`

Operations

- **PUT** `/sfc/port_chains/{id}`

Update a port chain

`delete_port_chain`

Default
`rule:admin_or_owner`

Operations

- **DELETE** `/sfc/port_chains/{id}`

Delete a port chain

`get_port_chain`

Default
`rule:admin_or_owner`

Operations

- **GET** `/sfc/port_chains`
- **GET** `/sfc/port_chains/{id}`

Get port chains

`create_port_pair_group`

Default
`rule:regular_user`

Operations

- **POST** `/sfc/port_pair_groups`

Create a port pair group

`update_port_pair_group`

Default
`rule:admin_or_owner`

Operations

- **PUT** `/sfc/port_pair_groups/{id}`

Update a port pair group

`delete_port_pair_group`

Default
`rule:admin_or_owner`

Operations

- **DELETE** `/sfc/port_pair_groups/{id}`

Delete a port pair group

`get_port_pair_group`

Default
`rule:admin_or_owner`

Operations

- **GET** `/sfc/port_pair_groups`
- **GET** `/sfc/port_pair_groups/{id}`

Get port pair groups

`create_port_pair`

Default
`rule:regular_user`

Operations

- **POST** `/sfc/port_pairs`

Create a port pair

`update_port_pair`

Default
`rule:admin_or_owner`

Operations

- **PUT** `/sfc/port_pairs/{id}`

Update a port pair

`delete_port_pair`

Default
`rule:admin_or_owner`

Operations

- **DELETE** `/sfc/port_pairs/{id}`

Delete a port pair

`get_port_pair`

Default
`rule:admin_or_owner`

Operations

- **GET** `/sfc/port_pairs`

- **GET** `/sfc/port_pairs/{id}`

Get port pairs

`create_service_graph`

Default
`rule:regular_user`

Operations

- **POST** `/sfc/service_graphs`

Create a service graph

`update_service_graph`

Default
`rule:admin_or_owner`

Operations

- **PUT** `/sfc/service_graphs/{id}`

Update a service graph

`delete_service_graph`

Default
`rule:admin_or_owner`

Operations

- **DELETE** `/sfc/service_graphs/{id}`

Delete a service graph

`get_service_graph`

Default
`rule:admin_or_owner`

Operations

- **GET** `/sfc/service_graphs`
- **GET** `/sfc/service_graphs/{id}`

Get service graphs

4.2.2 Sample networking-sfc Policy File

The following is a sample networking-sfc policy file for adaptation and use.

The sample policy can also be viewed in `file form`.

Important

The sample policy file is auto-generated from networking-sfc when this documentation is built. You must ensure your version of networking-sfc matches the version of this documentation.

```
# Create a flow classifier
# POST  /sfc/flow_classifiers
#"create_flow_classifier": "rule:regular_user"

# Update a flow classifier
# PUT  /sfc/flow_classifiers/{id}
#"update_flow_classifier": "rule:admin_or_owner"

# Delete a flow classifier
# DELETE  /sfc/flow_classifiers/{id}
#"delete_flow_classifier": "rule:admin_or_owner"

# Get flow classifiers
# GET  /sfc/flow_classifiers
# GET  /sfc/flow_classifiers/{id}
#"get_flow_classifier": "rule:admin_or_owner"

# Create a port chain
# POST  /sfc/port_chains
#"create_port_chain": "rule:regular_user"

# Update a port chain
# PUT  /sfc/port_chains/{id}
#"update_port_chain": "rule:admin_or_owner"

# Delete a port chain
# DELETE  /sfc/port_chains/{id}
#"delete_port_chain": "rule:admin_or_owner"

# Get port chains
# GET  /sfc/port_chains
# GET  /sfc/port_chains/{id}
#"get_port_chain": "rule:admin_or_owner"

# Create a port pair group
# POST  /sfc/port_pair_groups
#"create_port_pair_group": "rule:regular_user"

# Update a port pair group
# PUT  /sfc/port_pair_groups/{id}
#"update_port_pair_group": "rule:admin_or_owner"

# Delete a port pair group
# DELETE  /sfc/port_pair_groups/{id}
#"delete_port_pair_group": "rule:admin_or_owner"

# Get port pair groups
# GET  /sfc/port_pair_groups
# GET  /sfc/port_pair_groups/{id}
#"get_port_pair_group": "rule:admin_or_owner"
```

(continues on next page)

(continued from previous page)

```
# Create a port pair
# POST  /sfc/port_pairs
#"create_port_pair": "rule:regular_user"

# Update a port pair
# PUT  /sfc/port_pairs/{id}
#"update_port_pair": "rule:admin_or_owner"

# Delete a port pair
# DELETE  /sfc/port_pairs/{id}
#"delete_port_pair": "rule:admin_or_owner"

# Get port pairs
# GET  /sfc/port_pairs
# GET  /sfc/port_pairs/{id}
#"get_port_pair": "rule:admin_or_owner"

# Create a service graph
# POST  /sfc/service_graphs
#"create_service_graph": "rule:regular_user"

# Update a service graph
# PUT  /sfc/service_graphs/{id}
#"update_service_graph": "rule:admin_or_owner"

# Delete a service graph
# DELETE  /sfc/service_graphs/{id}
#"delete_service_graph": "rule:admin_or_owner"

# Get service graphs
# GET  /sfc/service_graphs
# GET  /sfc/service_graphs/{id}
#"get_service_graph": "rule:admin_or_owner"
```

CHAPTER

FIVE

CONTRIBUTOR GUIDE

In the Contributor Guide, you will find information on Networking-SFC lower level programming APIs. There are sections that cover the core pieces of networking-sfc, including its api, command-lines, database, system-design, alembic-migration etc. There are also subsections that describe specific plugins inside networking-sfc. Finally, the developer guide includes information about testing infrastructure.

5.1 Programming HowTos and Tutorials

5.1.1 Contribution

If you would like to contribute to the development of OpenStack, you must follow the steps in this page: https://docs.openstack.org/infra/manual/developers.html

If you already have a good understanding of how the system works and your OpenStack accounts are set up, you can skip to the development workflow section of this documentation to learn how changes to OpenStack should be submitted for review via the Gerrit tool: https://docs.openstack.org/infra/manual/developers.html#development-workflow

Pull requests submitted through GitHub will be ignored.

Bugs should be filed on Launchpad, not GitHub: https://bugs.launchpad.net/networking-sfc

5.1.2 Alembic-migration

Using alembic-migration, required data modeling for networking-sfc is defined and applied to the database. Refer to Neutron alembic migration process for further details.

The important operations are listed below.

Checking migration

```
neutron-db-manage --subproject networking-sfc check_migration
  Running branches for networking-sfc ...
start_networking_sfc (branchpoint)
                     -> 48072cb59133 (contract) (head)
                     -> 24fc7241aa5 (expand)

  OK
```

Checking branch information

```
neutron-db-manage --subproject networking-sfc branches
  Running branches for networking-sfc ...
start_networking_sfc (branchpoint)
                     -> 48072cb59133 (contract) (head)
                     -> 24fc7241aa5 (expand)

  OK
```

Checking migration history

```
neutron-db-manage --subproject networking-sfc history
  Running history for networking-sfc ...
9768e6a66c9 -> 5a475fc853e6 (expand) (head), Defining OVS data-model
24fc7241aa5 -> 9768e6a66c9 (expand), Defining flow-classifier data-model
start_networking_sfc -> 24fc7241aa5 (expand), Defining Port Chain data-model.
start_networking_sfc -> 48072cb59133 (contract) (head), Initial Liberty no-op␣
↪script.
<base> -> start_networking_sfc (branchpoint), start networking-sfc chain
```

Applying changes

```
neutron-db-manage --subproject networking-sfc upgrade head
INFO  [alembic.runtime.migration] Context impl MySQLImpl.
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
  Running upgrade for networking-sfc ...
INFO  [alembic.runtime.migration] Context impl MySQLImpl.
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
INFO  [alembic.runtime.migration] Running upgrade  -> start_networking_sfc,␣
↪start networking-sfc chain
INFO  [alembic.runtime.migration] Running upgrade start_networking_sfc ->␣
↪48072cb59133, Initial Liberty no-op script.
INFO  [alembic.runtime.migration] Running upgrade start_networking_sfc ->␣
↪24fc7241aa5, Defining Port Chain data-model.
INFO  [alembic.runtime.migration] Running upgrade 24fc7241aa5 -> 9768e6a66c9,␣
↪Defining flow-classifier data-model
INFO  [alembic.runtime.migration] Running upgrade 9768e6a66c9 -> 5a475fc853e6,
↪ Defining OVS data-model
  OK
```

Checking current version

```
neutron-db-manage --subproject networking-sfc current
  Running current for networking-sfc ...
INFO  [alembic.runtime.migration] Context impl MySQLImpl.
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
48072cb59133 (head)
5a475fc853e6 (head)
  OK
```

5.2 Networking-SFC Internals

5.2.1 API Model

Problem Description

Currently Neutron does not support service function chaining. To support service function chaining, Service VMs must be attached at points in the network and then traffic must be steered between these attachment points. Please refer to Neutron Service Chain blue-print and Bugs [1] [2] related to this specification for more information.

Proposed Change

All Neutron network services and VMs are connected to a Neutron network via Neutron ports. This makes it possible to create a traffic steering model for service chaining that uses only Neutron ports. This traffic steering model has no notion of the actual services attached to these Neutron ports.

The service VM hosting the service functions is instantiated and configured, then VNICs are added to the VM and then these VNICs are attached to the network by Neutron ports. Once the service function is attached to Neutron ports, the ports may be included in a port chain to allow the service function to provide treatment to the users traffic.

A Port Chain (Service Function Path) consists of:

- a set of Neutron ports, to define the sequence of service functions
- a set of flow classifiers, to specify the classified traffic flows to enter the chain

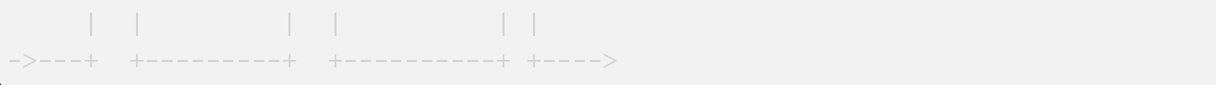
If a service function has a pair of ports, the first port in the port-pair is the ingress port of the service function, and the second port is the egress port of the service function. If a service function has one bidirectional port, then both ports in the port-pair have the same value. A Port Chain is a directional service chain. The first port of the first port-pair is the head of the service chain. The second port of the last port-pair is the tail of the service chain. A bidirectional service chain would be composed of two unidirectional Port Chains.

For example, [{p1: p2}, {p3: p4}, {p5: p6}] represents:

```
   +-------+      +-------+      +-------+
   | SF1   |      | SF2   |      | SF3   |
   +-------+      +-------+      +-------+
   p1|   |p2      p3|   |p4       p5|  |p6
```

(continues on next page)

(continued from previous page)



where p1 is the head of the Port Chain and p6 is the tail of the Port Chain, and SF1 has ports p1 and p2, SF2 has ports p3 and p4, and SF3 has ports p5 and p6.

In order to create a chain, the user needs to have the actual port objects. The work flow would typically be:

1. create the ports
2. create the chain
3. boot the vms passing the ports as nics parameters

The sequence of 2. and 3. can be switched.

A SFs Neutron port may be associated with more than one Port Chain to allow a service function to be shared by multiple chains.

If there is more than one service function instance of a specific type available to meet the users service requirement, their Neutron ports are included in the port chain as a sub-list. For example, if {p3, p4}, {p7, p8} are the port-pairs of two FW instances, they both may be included in a port chain for load distribution as shown below:

```
[{'p1': 'p2'}, [{'p3': 'p4'},{'p7': 'p8'}], {'p5': 'p6'}]
```

Flow classifiers are used to select the traffic that can access the chain. Traffic that matches any flow classifier will be directed to the first port in the chain. The flow classifier will be a generic independent module and may be used by other projects like FW, QOS, etc.

A flow classifier cannot be part of two different port-chains otherwise ambiguity will arise as to which chain path that flows packets should go. A check will be made to ensure no ambiguity. But multiple flow classifiers can be associated with a port chain since multiple different types of flows can request the same service treatment path.

CLI Commands

Syntax:

```
openstack sfc port pair create [-h]
        [--description <description>]
        --ingress <port-id>
        --egress <port-id>
        [--service-function-parameters <parameter>] PORT-PAIR-NAME

openstack sfc port pair group create [-h]
        [--description <description>]
        --port-pair <port-pair-id>
        [--port-pair-group-parameters <parameter>] PORT-PAIR-GROUP-NAME

openstack sfc flow classifier create [-h]
        [--description <description>]
```

(continues on next page)

(continued from previous page)

```
        [--protocol <protocol>]
        [--ethertype <Ethertype>]
        [--source-port <Minimum source protocol port>:<Maximum source␣
↪protocol port>]
        [--destination-port <Minimum destination protocol port>:<Maximum␣
↪destination protocol port>]
        [--source-ip-prefix <Source IP prefix>]
        [--destination-ip-prefix <Destination IP prefix>]
        [--logical-source-port <Neutron source port>]
        [--logical-destination-port <Neutron destination port>]
        [--l7-parameters <L7 parameter>] FLOW-CLASSIFIER-NAME

openstack sfc port chain create [-h]
        [--description <description>]
         --port-pair-group <port-pair-group-id>
        [--flow-classifier <classifier-id>]
        [--chain-parameters <chain-parameter>] PORT-CHAIN-NAME
```

openstack sfc port chain create

The `sfc port chain create` command returns the ID of the Port Chain.

Each `--port-pair-group` option specifies a type of SF. If a chain consists of a sequence of different types of SFs, then the chain will have multiple port-pair-groups. There must be at least one port-pair-group in the Port Chain.

The `-flow-classifier` option may be repeated to associate multiple flow classifiers with a port chain, with each classifier identifying a flow. If the flow-classifier is not specified, then no traffic will be steered through the chain.

One chain parameter option is currently defined. More parameter options can be added in future extensions to accommodate new requirements. The `correlation` parameter is used to specify the type of chain correlation mechanism. This parameter allows different correlation mechanisms to be selected. The chain correlation concept is equivalent to SFC Encapsulation, as defined in RFC 7665. The default is mpls, but nsh is also supported.

The `sfc port chain create` command returns the ID of a Port chain.

A port chain can be created, read, updated and deleted, and when a chain is created/read/updated/deleted, the options that are involved would be based on the CRUD in the Port Chain resource table below.

openstack sfc port pair group create

Inside each port-pair-group, there could be one or more port-pairs. Multiple port-pairs may be included in a port-pair-group to allow the specification of a set of functionally equivalent SFs that can be used for load distribution, i.e., the `--port-pair` option may be repeated for multiple port-pairs of functionally equivalent SFs.

The `sfc port pair group create` command returns the ID of a Port Pair group.

openstack sfc port pair create

A Port Pair represents a service function instance. The ingress port and the egress port of the service function may be specified. If a service function has one bidirectional port, the ingress port has the same value as the egress port. The `--service-function-parameters` option allows the passing of SF specific parameter information to the data path. These include:

- The `correlation` parameter is used to specify the type of chain correlation mechanism supported by a specific SF. This is needed by the data plane switch to determine how to associate a packet with a chain. This will be set to none for now since there is no correlation mechanism supported by the SF. In the future, it can be extended to include mpls, nsh, etc.. If this parameter is not specified, it will default to none.
- The `weight` parameter is used to specify the weight for each SF for load distribution in a port pair group. This represents a percentage of the traffic to be sent to each SF.

The `sfc port pair create` command returns the ID of a Port Pair.

openstack sfc flow classifier create

A combination of the source options defines the source of the flow. A combination of the destination options defines the destination of the flow. The l7_parameter is a place-holder that may be used to support flow classification using L7 fields, such as URL. If an option is not specified, it will default to wildcard value except for ethertype which defaults to IPv4, for logical-source-port and logical-destination-port which defaults to none.

The `sfc flow classifier create` command returns the ID of a flow classifier.

Data Model Impact

Data model:

```
 +--------+        +-----------+        +------------+
 | Port   |--------| Port Pair|--------| Port Pairs |
 | Chain  |*      *| Groups    | 1     *|            |
 +--------+        +-----------+        +------------+
   |1
   |
   |*
+--------------+
| Flow         |
| Classifiers  |
+--------------+
```

New objects:

Port Chain

- id - Port chain ID.
- project_id - Tenant ID.
- name - Readable name.

- description - Readable description.
- port_pair_groups - List of port-pair-group IDs.
- flow_classifiers - List of flow-classifier IDs.
- chain_parameters - Dict. of chain parameters.
- chain_id - Data-plane chain path ID.

Port Pair Group

- id - Port pair group ID.
- project_id - Tenant ID.
- name - Readable name.
- description - Readable description.
- port_pairs - List of service function (Neutron) port-pairs.
- port_pair_group_parameters - Dict. of port pair group parameters.

Port Pair

- id - Port pair ID.
- project_id - Tenant ID.
- name - Readable name.
- description - Readable description.
- ingress - Ingress port.
- egress - Egress port.
- service_function_parameters - Dict. of service function parameters

Flow Classifier

- id - Flow classifier ID.
- project_id - Tenant ID.
- name - Readable name.
- description - Readable description.
- ethertype - Ethertype (IPv4/IPv6).
- protocol - IP protocol.
- source_port_range_min - Minimum source protocol port.
- source_port_range_max - Maximum source protocol port.
- destination_port_range_min - Minimum destination protocol port.
- destination_port_range_max - Maximum destination protocol port.
- source_ip_prefix - Source IP address or prefix.
- destination_ip_prefix - Destination IP address or prefix.
- logical_source_port - Neutron source port.

- logical_destination_port - Neutron destination port.
- l7_parameters - Dictionary of L7 parameters.

REST API

Port Chain Operations:

Operation	URL	Description
POST	/sfc/port_chains	Create a Port Chain
PUT	/sfc/port_chains/{chain_id}	Update a specific Port Chain
DELETE	/sfc/port_chains/{chain_id}	Delete a specific Port Chain
GET	/sfc/port_chains	List all Port Chains for specified tenant
GET	/sfc/port_chains/{chain_id}	Show information for a specific Port Chain

Port Pair Group Operations:

Operation	URL	Description
POST	/sfc/port_pair_groups	Create a Port Pair Group
PUT	/sfc/port_pair_groups/{group_id}	Update a specific Port Pair Group
DELETE	/sfc/port_pair_groups/{group_id}	Delete a specific Port Pair Group
GET	/sfc/port_pair_groups	List all Port Pair Groups for specified tenant
GET	/sfc/port_pair_groups/{group_id}	Show information for a specific Port Pair

Port Pair Operations:

Operation	URL	Description
POST	/sfc/port_pairs	Create a Port Pair
PUT	/sfc/port_pairs/{pair_id}	Update a specific Port Pair
DELETE	/sfc/port_pairs/{pair_id}	Delete a specific Port Pair
GET	/sfc/port_pairs	List all Port Pairs for specified tenant
GET	/sfc/port_pairs/{pair_id}	Show information for a specific Port Pair

Flow Classifier Operations:

Operation	URL	Description
POST	/sfc/flow_classifiers	Create a Flow-classifier
PUT	/sfc/flow_classifiers/{flow_id}	Update a specific Flow-classifier
DELETE	/sfc/flow_classifiers/{flow_id}	Delete a specific Flow-classifier
GET	/sfc/flow_classifiers	List all Flow-classifiers for specified tenant
GET	/sfc/flow_classifiers/{flow_id}	Show information for a specific Flow-classifier

REST API Impact

The following new resources will be created as a result of the API handling.

Port Chain resource:

Attribute Name	Type	Access	Default Value	CRUD	Description
id	uuid	RO, all	generated	R	Port Chain ID.
project_id	uuid	RO, all	from auth token	CR	Tenant ID.
name	string	RW, all		CRU	Port Chain name.
description	string	RW, all		CRU	Port Chain description.
port_pair_groups	list(uuid)	RW, all	N/A	CRU	List of port-pair-groups.
flow_classifiers	list(uuid)	RW, all	[]	CRU	List of flow-classifiers.
chain_parameters	dict	RW, all	mpls	CR	Dict. of parameters: correlation:String
chain_id	integer	RW, all	Any	CR	Data-plane Chain Path ID.

The data-plane chain path ID is normally generated by the data-plane implementation. However, an application may optionally generate its own data-plane chain path ID and apply it to the Port Chain using the chain_id attribute.

Port Pair Group resource:

Attribute Name	Type	Access	Default Value	CRUD	Description
id	uuid	RO, all	generated	R	Port pair group ID.
project_id	uuid	RO, all	from auth token	CR	Tenant ID.
name	string	RW, all		CRU	Port pair group name.
description	string	RW, all		CRU	Port pair group description.
port_pairs	list	RW, all	N/A	CRU	List of port-pairs.
port_pair_group _parameters	dict	RW, all		CR	Dict. of parameters: lb_fields:String service_type:String

Port Pair resource:

Attribute Name	Type	Access	Default	CRUD	Description
id	uuid	RO, all	generated	R	Port pair ID.
project_id	uuid	RO, all	from auth token	CR	Tenant ID.
name	string	RW, all		CRU	Port pair name.
description	string	RW, all		CRU	Port pair description.
ingress	uuid	RW, all	N/A	CR	Ingress port ID.
egress	uuid	RW, all	N/A	CR	Egress port ID.
service_function_parame	dict	RW, all	None	CR	Dict. of parameters: correlation:String weight:Integer

Flow Classifier resource:

Attribute Name	Type	Access	Default Value	CRUD	Description
id	uuid	RO, all	generated	R	Flow-classifier ID.
project_id	uuid	RO, all	from auth token	CR	Tenant ID.
name	string	RW, all		CRU	Flow-classifier name.
description	string	RW, all		CRU	Flow-classifier description.
ethertype	string	RW, all	IPv4	CR	L2 ethertype. Can be IPv4 or IPv6 only.
protocol	string	RW, all	Any	CR	IP protocol name.
source_port_range_min	integer	RW, all	Any	CR	Minimum source protocol port.
source_port_range_max	integer	RW, all	Any	CR	Maximum source protocol port.
destination_port_range_min	integer	RW, all	Any	CR	Minimum destination protocol port.
destination_port_range_max	integer	RW, all	Any	CR	Maximum destination protocol port.
source_ip_prefix	CIDR	RW, all	Any	CR	Source IPv4 or IPv6 prefix.
destination_ip_prefix	CIDR	RW, all	Any	CR	Destination IPv4 or IPv6 prefix.
logical_source_port	uuid	RW, all	None	CR	Neutron source port.
logical_destination_port	uuid	RW, all	None	CR	Neutron destination port.
l7_parameters	dict	RW, all	Any	CR	Dict. of L7 parameters.

Json Port-pair create request example:

```
{"port_pair": {"name": "SF1",
       "project_id": "d382007aa9904763a801f68ecf065cf5",
       "description": "Firewall SF instance",
       "ingress": "dace4513-24fc-4fae-af4b-321c5e2eb3d1",
       "egress": "aef3478a-4a56-2a6e-cd3a-9dee4e2ec345",
   }
}

{"port_pair":  {"name": "SF2",
       "project_id": "d382007aa9904763a801f68ecf065cf5",
       "description": "Loadbalancer SF instance",
       "ingress": "797f899e-73d4-11e5-b392-2c27d72acb4c",
       "egress": "797f899e-73d4-11e5-b392-2c27d72acb4c",
   }
```

(continues on next page)

(continued from previous page)

```
}
```

Json Port-pair create response example:

```
{"port_pair": {"name": "SF1",
       "project_id": "d382007aa9904763a801f68ecf065cf5",
       "description": "Firewall SF instance",
       "ingress": "dace4513-24fc-4fae-af4b-321c5e2eb3d1",
       "egress": "aef3478a-4a56-2a6e-cd3a-9dee4e2ec345",
       "id": "78dcd363-fc23-aeb6-f44b-56dc5e2fb3ae",
   }
 }

{"port_pair":  {"name": "SF2",
       "project_id": "d382007aa9904763a801f68ecf065cf5",
       "description": "Loadbalancer SF instance",
       "ingress": "797f899e-73d4-11e5-b392-2c27d72acb4c",
       "egress": "797f899e-73d4-11e5-b392-2c27d72acb4c",
       "id": "d11e9190-73d4-11e5-b392-2c27d72acb4c"
   }
}
```

Json Port Pair Group create request example:

```
{"port_pair_group": {"name": "Firewall_PortPairGroup",
       "project_id": "d382007aa9904763a801f68ecf065cf5",
       "description": "Grouping Firewall SF instances",
       "port_pairs": [
           "78dcd363-fc23-aeb6-f44b-56dc5e2fb3ae"
       ],
       "port_pair_group_parameters": [
           "lb_fields: ip_src"
       ]
   }
 }

{"port_pair_group": {"name": "Loadbalancer_PortPairGroup",
       "project_id": "d382007aa9904763a801f68ecf065cf5",
       "description": "Grouping Loadbalancer SF instances",
       "port_pairs": [
           "d11e9190-73d4-11e5-b392-2c27d72acb4c"
       ]
       "port_pair_group_parameters": [
           "lb_fields: ip_src"
       ]
   }
}
```

Json Port Pair Group create response example:

```
{"port_pair_group": {"name": "Firewall_PortPairGroup",
       "project_id": "d382007aa9904763a801f68ecf065cf5",
       "description": "Grouping Firewall SF instances",
       "port_pairs": [
           "78dcd363-fc23-aeb6-f44b-56dc5e2fb3ae
       ],
       "port_pair_group_parameters": [
           "lb_fields: ip_src"
       ]
       "id": "4512d643-24fc-4fae-af4b-321c5e2eb3d1",
   }
}

{"port_pair_group":  {"name": "Loadbalancer_PortPairGroup",
       "project_id": "d382007aa9904763a801f68ecf065cf5",
       "description": "Grouping Loadbalancer SF instances",
       "port_pairs": [
           "d11e9190-73d4-11e5-b392-2c27d72acb4c"
       ],
       "port_pair_group_parameters": [
           "lb_fields: ip_src"
       ]
       "id": "4a634d49-76dc-4fae-af4b-321c5e23d651",
   }
}
```

Json Flow Classifier create request example:

```
{"flow_classifier": {"name": "FC1",
       "project_id": "1814726e2d22407b8ca76db5e567dcf1",
       "description": "Flow rule for classifying TCP traffic",
       "protocol": "TCP",
       "source_port_range_min": 22, "source_port_range_max": 4000,
       "destination_port_range_min": 80, "destination_port_range_max": 80,
       "source_ip_prefix": null, "destination_ip_prefix": "22.12.34.45"
   }
}

{"flow_classifier": {"name": "FC2",
       "project_id": "1814726e2d22407b8ca76db5e567dcf1",
       "description": "Flow rule for classifying UDP traffic",
       "protocol": "UDP",
       "source_port_range_min": 22, "source_port_range_max": 22,
       "destination_port_range_min": 80, "destination_port_range_max": 80,
       "source_ip_prefix": null, "destination_ip_prefix": "22.12.34.45"
   }
}
```

Json Flow Classifier create response example:

```
{"flow_classifier": {"name": "FC1",
       "project_id": "1814726e2d22407b8ca76db5e567dcf1",
       "description": "Flow rule for classifying TCP traffic",
       "protocol": "TCP",
       "source_port_range_min": 22, "source_port_range_max": 4000,
       "destination_port_range_min": 80, "destination_port_range_max": 80,
       "source_ip_prefix": null , "destination_ip_prefix": "22.12.34.45",
       "id": "4a334cd4-fe9c-4fae-af4b-321c5e2eb051"
   }
}

{"flow_classifier": {"name": "FC2",
       "project_id": "1814726e2d22407b8ca76db5e567dcf1",
       "description": "Flow rule for classifying UDP traffic",
       "protocol": "UDP",
       "source_port_range_min": 22, "source_port_range_max": 22,
       "destination_port_range_min": 80, "destination_port_range_max": 80,
       "source_ip_prefix": null , "destination_ip_prefix": "22.12.34.45",
       "id": "105a4b0a-73d6-11e5-b392-2c27d72acb4c"
   }
}
```

Json Port Chain create request example:

```
{"port_chain": {"name": "PC1",
       "project_id": "d382007aa9904763a801f68ecf065cf5",
       "description": "Steering TCP and UDP traffic first to Firewall and␣
↪then to Loadbalancer",
       "flow_classifiers": [
           "4a334cd4-fe9c-4fae-af4b-321c5e2eb051",
           "105a4b0a-73d6-11e5-b392-2c27d72acb4c"
       ],
       "port_pair_groups": [
           "4512d643-24fc-4fae-af4b-321c5e2eb3d1",
           "4a634d49-76dc-4fae-af4b-321c5e23d651"
       ],
       "chain_id": "10034"
   }
}
```

Json Port Chain create response example:

```
{"port_chain": {"name": "PC1",
       "project_id": "d382007aa9904763a801f68ecf065cf5",
       "description": "Steering TCP and UDP traffic first to Firewall and␣
↪then to Loadbalancer",
       "flow_classifiers": [
           "4a334cd4-fe9c-4fae-af4b-321c5e2eb051",
           "105a4b0a-73d6-11e5-b392-2c27d72acb4c"
       ],
```

(continues on next page)

(continued from previous page)

```
        "port_pair_groups": [
            "4512d643-24fc-4fae-af4b-321c5e2eb3d1",
            "4a634d49-76dc-4fae-af4b-321c5e23d651"
        ],
        "chain_id": "10034",
        "id": "1278dcd4-459f-62ed-754b-87fc5e4a6751"
    }
}
```

Implementation

Assignee(s)

Authors of the Specification and Primary contributors:

- Cathy Zhang (cathy.h.zhang@huawei.com)
- Louis Fourie (louis.fourie@huawei.com)

Other contributors:

- Vikram Choudhary (vikram.choudhary@huawei.com)
- Swaminathan Vasudevan (swaminathan.vasudevan@hp.com)
- Yuji Azama (yuj-azama@rc.jp.nec.com)
- Mohan Kumar (nmohankumar1011@gmail.com)
- Ramanjaneya (ramanjieee@gmail.com)
- Stephen Wong (stephen.kf.wong@gmail.com)
- Nicolas Bouthors (Nicolas.BOUTHORS@qosmos.com)
- Akihiro Motoki <amotoki@gmail.com>
- Paul Carver <pcarver@att.com>

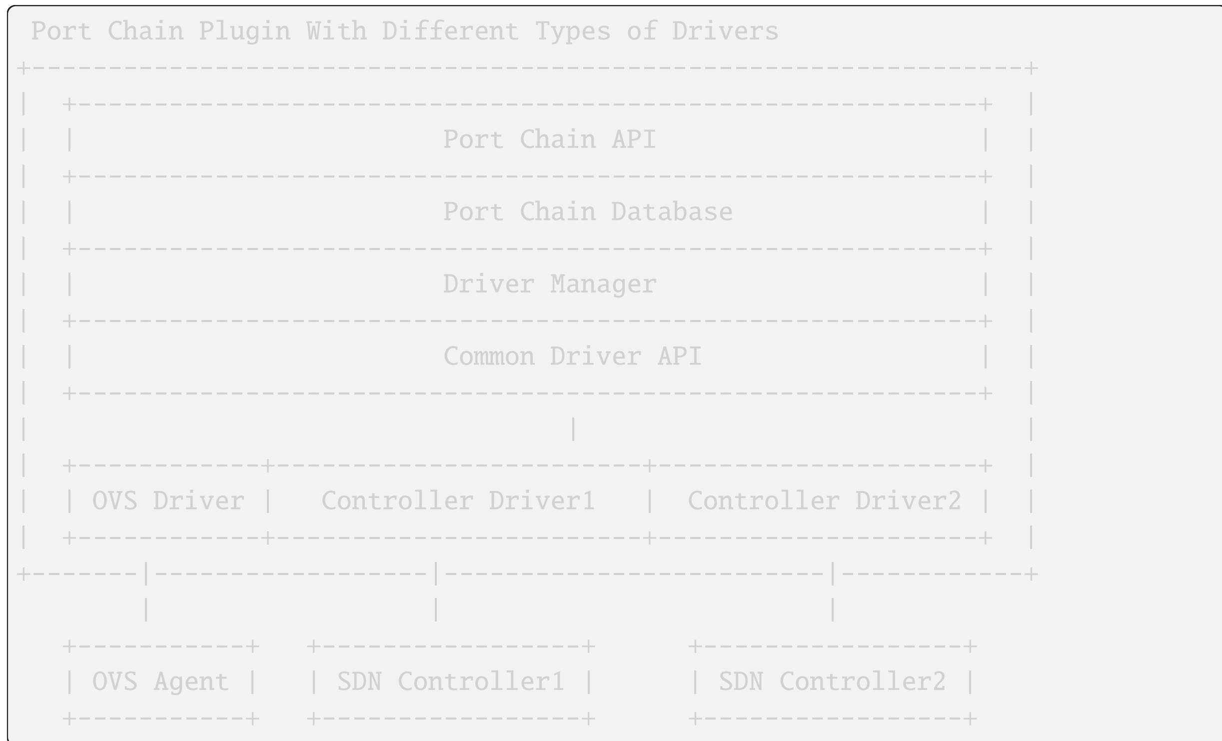
5.2.2 System Design and Workflow

Problem Description

The Service Chaining API specification proposes a Neutron port based solution for setting up a service chain. A specification on the system architecture and related API work flow is needed to guide the code design.

System Architecture

The following figure shows the generic architecture of the Port Chain Plugin. As shown in the diagram, Port Chain Plugin can be backed by different service providers such as OVS Driver and/or different types of SDN Controller Drivers. Through the Common Driver API, these different drivers can provide different implementations for the service chain path rendering. In the first release and deployment based on this release, we will only deliver codes for the OVS driver. In the next release, we can add codes to support multiple active drivers:



The second figure below shows the reference implementation architecture, which is through the OVS Driver path. The figure shows the components that will be added on the Neutron Server and the compute nodes to support this Neutron Based SFC functionality. As shown in the diagram, a new Port Chain Plugin will be added to the Neutron Server. The existing OVS Driver and OVS Agent will be extended to support the service chain functionality. The OVS Driver will communicate with each OVS Agent to program its OVS forwarding table properly so that a tenants traffic flow can be steered through the user defined sequence of Neutron ports to get the desired service treatment from the Service Function running on the VMs.

A separate OVS Driver and Agent specification will describe in more detail on the design consideration of the Driver, Agent, and how to set up the classification rules on the OVS to identify different flows and how to set up the OVS forwarding table. In the reference implementation, the OVS Driver communicates with OVS Agent through RPC to program the OVS. The communication between the OVS Agent and the OVS is through OVSDB/Openflow:

(continues on next page)

(continued from previous page)

Port Chain Creation Workflow

The following example shows how the Neutron CLI commands may be used to create a port-chain consisting of a service VM vm1 and a service VM vm2. The user can be an Admin/Tenant or an Application built on top.

Traffic flow into the Port Chain will be from source IP address 22.1.20.1 TCP port 23 to destination IP address 171.4.5.6 TCP port 100. The flow needs to be treated by SF1 running on VM1 identified by Neutron port pair [p1, p2], SF2 running on VM2 identified by Neutron port pair [p3, p4], and SF3 running on VM3 identified by Neutron port pair [p5, p6].

The net1 should be created before creating Neutron port using existing Neutron API. The design has no restriction on the type of net1, i.e. it can be any type of Neutron network since SFC traffic will be tunneled transparently through the type of communication channels of net1. If the transport between vSwitches is VXLAN, then we will use that VXLAN tunnel (and NOT create another new tunnel) to transport the SFC traffic through. If the transport between vSwitches is Ethernet, then the SFC traffic will be transported through Ethernet. In other words, the SFC traffic will be carried over existing transport channel between vSwitches and the external transport channel between vSwitches is set up for net1 through existing Neutron API and ML2. The built-in OVS backend implements tunneling the original flow packets over VXLAN tunnel. The detailed outer VXLAN tunnel transport format and inner SFC flow format including how to leverage existing OVSs support for MPLS label to carry chain ID will be described in the Port Chain OVS Driver and Agent specification. In the future we can add implementation of tunneling the SFC flow packets over flat L2 Ethernet or L3 IP network or GRE tunnel etc.

Boot service VMs and attach ports

Create Neutron ports on network net1:

```
openstack port create --network net1 p1
openstack port create --network net1 p2
openstack port create --network net1 p3
openstack port create --network net1 p4
openstack port create --network net1 p5
openstack port create --network net1 p6
```

Boot VM1 from Nova with ports p1 and p2 using two nic options:

```
openstack server create --image xxx --nic port-id=p1-id --nic port-id=p2-id␣
↪vm1 --flavor <image-flavour>
```

Boot VM2 from Nova with ports p3 and p4 using two nic options:

```
openstack server create --image yyy --nic port-id=p3-id --nic port-id=p4-id␣
↪vm2 --flavor <image-flavour>
```

Boot VM3 from Nova with ports p5 and p6 using two nic options:

```
openstack server create --image zzz --nic port-id=p5-id --nic port-id=p6-id␣
↪vm3 --flavor <image-flavour>
```

Alternatively, the user can create each VM with one VNIC and then attach another Neutron port to the VM:

```
openstack server create --image xxx --nic port-id=p1-id vm1
openstack server add port vm1 p2-id
openstack server create --image yyy --nic port-id=p3-id vm2
openstack server add port vm2 p4-id
openstack server create --image zzz --nic port-id=p5-id vm3
openstack server add port vm3 p6-id
```

Once the Neutron ports p1 - p6 exist, the Port Chain is created using the steps described below.

Create Flow Classifier

Create flow-classifier FC1 that matches on source IP address 22.1.20.1 (ingress direction) and destination IP address 171.4.5.6 (egress direction) with TCP connection, source port 23 and destination port 100:

```
openstack sfc flow classifier create \
 --ethertype IPv4 \
 --source-ip-prefix 22.1.20.1/32 \
 --destination-ip-prefix 172.4.5.6/32 \
 --protocol tcp \
 --source-port 23:23 \
 --destination-port 100:100 FC1
```

Note

When using the (default) OVS driver, the `--logical-source-port` parameter is also required

Create Port Pair

Create port-pair PP1 with ports p1 and p2, port-pair PP2 with ports p3 and p4, port-pair PP3 with ports P5 and P6:

```
openstack sfc port pair create \
       --ingress=p1 \
       --egress=p2 PP1

openstack sfc port pair create \
       --ingress=p3 \
       --egress=p4 PP2

openstack sfc port pair create \
       --ingress=p5 \
       --egress=p6 PP3
```

Create Port Group

Create port-pair-group PG1 with port-pair PP1 and PP2, and port-pair-group PG2 with port-pair PP3:

```
openstack sfc port pair group create \
       --port-pair PP1 --port-pair PP2 PG1

openstack sfc port pair group create \
       --port-pair PP3 PG2
```

Create Port Chain

Create port-chain PC1 with port-group PG1 and PG2, and flow classifier FC1:

```
openstack sfc port chain create \
       --port-pair-group PG1 --port-pair-group PG2 --flow-classifier FC1 PC1
```

This will result in the Port chain driver being invoked to create the Port Chain.

The following diagram illustrates the code execution flow (not the exact codes) for the port chain creation:

```
PortChainAPIParsingAndValidation: create_port_chain
               |
               V
PortChainPlugin: create_port_chain
               |
               V
PortChainDbPlugin: create_port_chain
               |
               V
DriverManager: create_port_chain
               |
```

(continues on next page)

(continued from previous page)

```
                V
portchain.drivers.OVSDriver: create_port_chain
```

The vSwitch Driver needs to figure out which switch VM1 is connecting with and which switch VM2 is connecting with (for OVS case, the OVS driver has that information given the VMs port info). As to the connection setup between the two vSwitches, it should be done through existing ML2 plugin mechanism. The connection between these two vSwitches should already be set up before the user initiates the SFC request. The service chain flow packets will be tunneled through the connecting type/technology (e.g. VXLAN or GRE) between the two vSwitches. For our reference code implementation, we will use VXLAN to show a complete data path setup. Please refer to the OVS Driver and OVS Agent specification for more detail info.

5.2.3 OVS Driver and Agent Workflow

Blueprint about Common Service chaining driver describes the OVS driver and agent necessity for realizing service function chaining.

Problem Description

The service chain OVS driver and agents are used to configure back-end Openvswitch devices to render service chaining in the data-plane. The driver manager controls a common service chain API which provides a consistent interface between the service chain manager and different device drivers.

Proposed Change

Design:

```
  Port Chain Plugin
+-------------------------------+
|  +-------------------------+  |
|  |    Port Chain API       |  |
|  +-------------------------+  |
|  |    Port Chain Database  |  |
|  +-------------------------+  |
|  |    Driver Manager       |  |
|  +-------------------------+  |
|  |    Common Driver API    |  |
|  +-------------------------+  |
|              |                |
|  +-------------------------+  |
|  |        OVS Driver       |  |
|  +-------------------------+  |
+-------|----------------|------+
        |rpc             |rpc
  +-----------+   +-----------+
  | OVS Agent |   | OVS Agent |
  +-----------+   +-----------+
```

A OVS service chain driver and agents communicate via rpc.

OVS Driver

The OVS Driver is extended to support service chaining. The driver interfaces with the OVS agents that reside on each Compute node. The OVS driver is responsible for the following:

- Identify the OVS agents that directly connects to the SF instances and establish communication with OVS agents on the Compute nodes.
- Send commands to the OVS agents to create bridges, flow tables and flows to steer chain traffic to the SF instances.

OVS Agent

The OVS agent will manage the OVS using OVSDB commands to create bridges and tables, and install flows to steer chain traffic to the SF instances.

Existing tunnels between the Tunnel bridges on each Compute node are used to transport Port Chain traffic between the CNs.

The OVS Agent will create these tunnels to transport SFC traffic between Compute nodes on which there are SFs. Each tunnel port has the following attributes:

- Name
- Local tunnel IP address
- Remote tunnel IP address
- Tunnel Type: VXLAN, GRE

The OVS agent installs additional flows on the Integration bridge and the Tunnel bridge to perform the following functions:

- Traffic classification. The Integration bridge classifies traffic from a VM port or Service VM port attached to the Integration bridge. The flow classification is based on the n-tuple rules.
- Service function forwarding. The Tunnel bridge forwards service chain packets to the next-hop Compute node via tunnels, or to the next Service VM port on that Compute node. Integration bridge will terminate a Service Function Path.

The OVS Agent will use the MPLS header to transport the chain path identifier and chain hop index. The MPLS label will transport the chain path identifier, and the MPLS ttl will transport the chain hop index. The following packet encapsulation will be used:

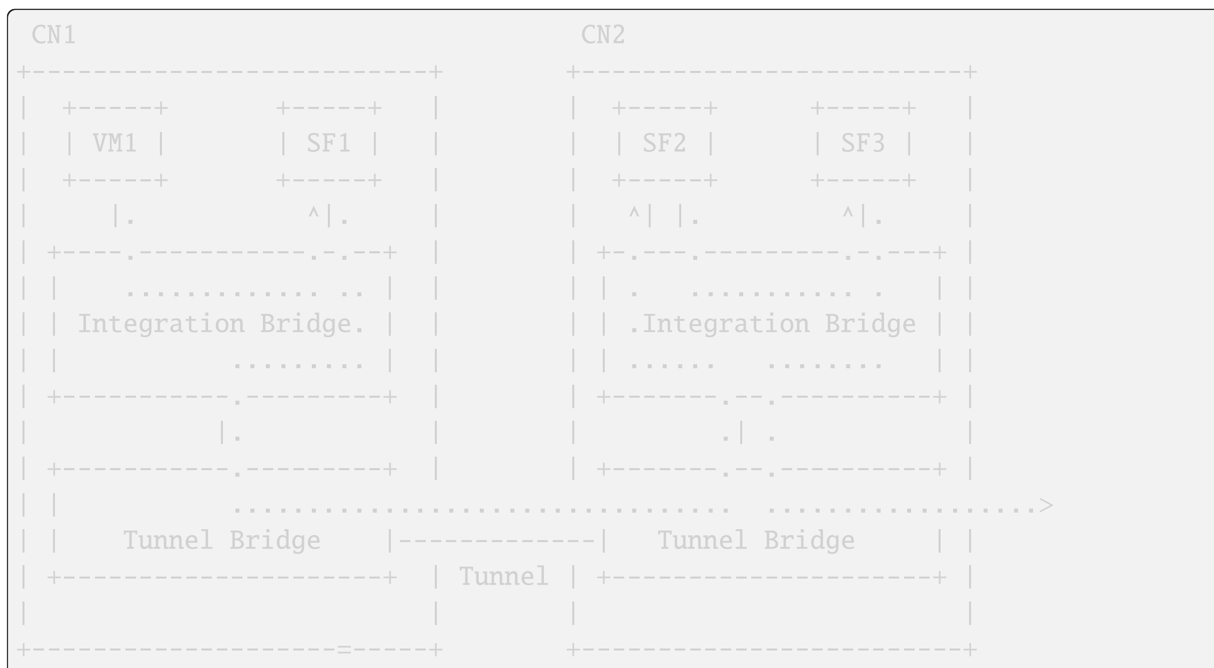
```
IPv4 Packet:
+----------+------------------------+-------+
|L2 header | IP + UDP dst port=4790 | VXLAN |
+----------+------------------------+-------+
-----------------------------+---------------+--------------------+
Original Ethernet, ET=0x8847 | MPLS header   | Original IP Packet |
-----------------------------+---------------+--------------------+
```

This is not intended as a general purpose MPLS implementation but rather as a temporary internal mechanism. It is anticipated that the MPLS label will be replaced with an NSH encapsulation (https://datatracker.ietf.org/doc/draft-ietf-sfc-nsh/) once NSH support is available upstream in Open vSwitch.

If the service function does not support the header, then the vSwitch will act as Service Function Forwarder (SFF) Proxy which will strip off the header when forwarding the packet to the SF and re-add the header when receiving the packet from the SF.

OVS Bridge and Tunnel

Existing tunnels between the Tunnel bridges on each Compute node are used to transport Port Chain traffic between the CNs:



Flow Tables and Flow Rules

The OVS Agent adds additional flows (shown above) on the Integration bridge to support Port Chains:

1. Egress Port Chain flows to steer traffic from SFs attached to the Integration bridge to a Tunnel bridge to the next-hop Compute node. These flows may be handled using the OpenFlow Group in the case where there are multiple port-pairs in the next-hop port-pair group.
2. Ingress Port Chain flows on the Tunnel bridge to steer service chain traffic from a tunnel from a previous Compute node to SFs attached to the Integration bridge.
3. Internal Port Chain flows are used to steer service chain traffic from one SF to another SF on the same Compute Node.

The Port Chain flow rules have the higher priority, and will not impact the existing flow rules on the Integration bridge. If traffic from SF is not part of a service chain, e.g., DHCP messages, ARP packets etc., it will match the existing flow rules on the Integration bridge.

The following tables are used to process Port Chain traffic:

- Local Switching Table (Table 0). This existing table has two new flows to handle incoming traffic from the SF egress port and the tunnel port between Compute nodes.

- Group Table. This new table is used to select multiple paths for load-balancing across multiple port-pairs in a port-pair group. There are multiple buckets in the group if the next hop is a port-pair group with multiple port-pairs. The group actions will be to send the packet to next hop SF instance. If the next hop port-pair is on another Compute node, the action output to the tunnel port to the next hop Compute node. If the next hop port-pair is on the same Compute node, then the action will be to resubmit to the TUN_TABLE for local chaining process.

Local Switching Table (Table 0) Flows

Traffic from SF Egress port: classify for chain and direct to group:

```
priority=10,in_port=SF_EGRESS_port,traffic_match_field,
 actions=strip_vlan,set_tunnel:VNI,group:gid.
```

Traffic from Tunnel port:

```
priority=10,in_port=TUNNEL_port,
 actions=resubmit(,TUN_TABLE[type]).
```

Group Table Flows

The Group table is used for load distribution to spread the traffic load across a port-pair group of multiple port-pairs (SFs of the same type). This uses the hashing of several fields in the packet. There are multiple buckets in the group if the next hop is a port-pair group with multiple port-pairs.

The group actions will be to send the packet to next hop SF instances. If the next hop port-pair is on another Compute node, the action output to the tunnel port to the next hop Compute node. If the next hop port-pair is on the same Compute node, then the action will be to resubmit to the TUN_TABLE for local chaining process.

The OVSDB command to create a group of type Select with a hash selection method and two buckets is shown below. This is existing OVS functionality. The ip_src,nw_proto,tp_src packet fields are used for the hash:

```
group_id=gid,type=select,selection_method=hash,fields=ip_src,nw_proto,tp_src
 bucket=set_field:10.1.1.3->ip_dst,output:10,
 bucket=set_field:10.1.1.4->ip_dst,output:10
```

Data Model Impact

None

Alternatives

None

Security Impact

None.

Notifications Impact

There will be logging to trouble-shoot and verify correct operation.

Other End User Impact

None.

Performance Impact

It is not expected that these flows will have a significant performance impact.

IPv6 Impact

None.

Other Deployer Impact

None

Developer Impact

None

Community Impact

Existing OVS driver and agent functionality will not be affected.

Implementation

Assignee(s)

- Cathy Zhang (cathy.h.zhang@huawei.com)
- Louis Fourie (louis.fourie@huawei.com)
- Stephen Wong (stephen.kf.wong@gmail.com)

Work Items

- Port Chain OVS driver.
- Port Chain OVS agent.
- Unit test.

Dependencies

This design depends upon the proposed Neutron Service Chaining API extensions

Openvswitch.

Testing

Tempest and functional tests will be created.

Documentation Impact

Documented as extension.

User Documentation

Update networking API reference. Update admin guide.

Developer Documentation

None

5.2.4 Networking-sfc / OVN Driver

https://blueprints.launchpad.net/networking-sfc/+spec/networking-sfc-ovn-driver

This specification describes a networking-sfc driver that will interface with a new Logical Port Chain resource API for the OVN infrastructure. The driver will translate networking-sfc requests into Logical Port Chain resources in the OVN northbound DB. These Logical Port Chain resources are created in OVN by updating the appropriate tables in the OVN northbound database (an ovsdb database).

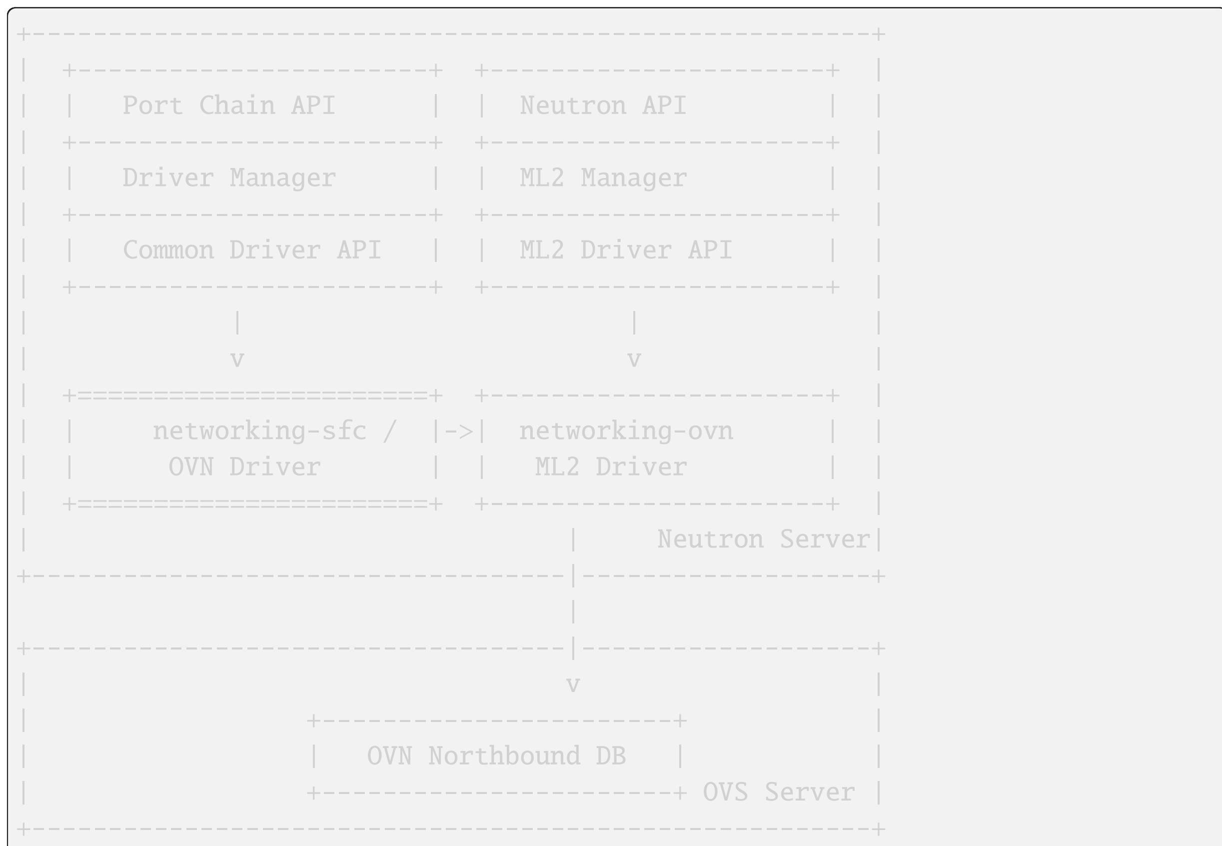
Problem Description

networking-sfc allows various drivers to be used. Currently, drivers exist for OVS, ONOS and ODL infrastructures. Service chaining is being added to OVN and a driver is required to interface between networking-sfc and the OVN infrastructure.

Proposed Changes

The proposed extensions to the OVN northbound DB schema and API are described briefly here. Refer to openvswitch documentation for details. In addition the new OVN driver for networking-sfc will map from networking-sfc requests to Logical Port Chain resources in the OVN northbound DB via the networking-ovn driver.

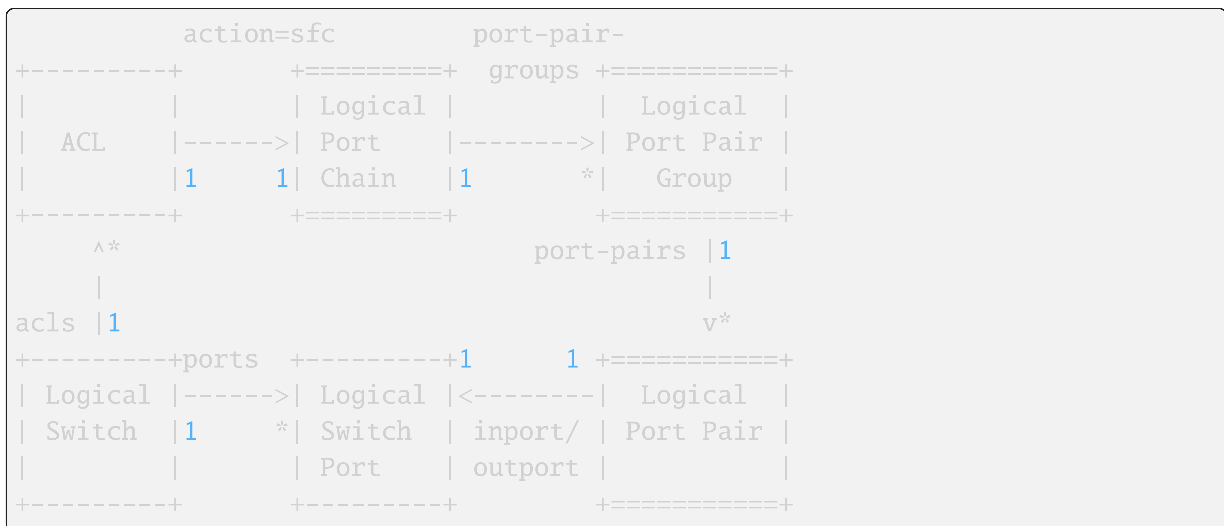
The OVN driver for networking-sfc is shown below.



OVN Northbound Port Chain DB

The proposed OVN northbound DB extensions for Logical Port Chains are shown below with three new resources:

- Logical Port Chain
- Logical Port Pair Group
- Logical Port Pair



```
            action=sfc          port-pair-
+----------+       +==========+  groups +============+
|          |       | Logical  |         |  Logical   |
|   ACL    |------>| Port     |-------->| Port Pair  |
|          |1     1| Chain    |1       *|   Group    |
+----------+       +==========+         +============+
     ^*                                 port-pairs |1
     |                                             |
acls |1                                            v*
+----------+ports  +----------+1      1 +============+
| Logical  |------>| Logical  |<--------|  Logical   |
| Switch   |1     *| Switch   | inport/ | Port Pair  |
|          |       | Port     | outport |            |
+----------+       +----------+         +============+
```

The OVN ACL actions are extended to include a SFC action with an external_id to reference the name of the Logical Port Chain (lchain) with which the ACL is associated. The sfc action means that the packet is allowed and steered into the port-chain.

Logical Port Chain

A Logical Port Chain can contain one or more Logical Port Pair Groups. The order of Logical Port Pair Groups in the Logical Port Chain specifies the order of steering packets through the Port Chain from the outport of a Logical Port Pair in one Logical Port Pair Group to the inport of a Logical Port Pair in the next Logical Port Pair Group.

Logical Port Pair Group

A Logical Port Pair Group can contain one or more Logical Port Pairs and is used to load balance traffic across the Service Functions (Logical Port Pairs) in the Logical Port Pair Group. A Logical Port Pair Group can be a member of multiple Logical Port Chains.

Logical Port Pair

A Logical Port Pair represents the ingress Logical Switch Port and the egress Logical Switch Port of a Service Function. A Logical Port Pair can be a member of only one Logical Port Pair Group. An OVN Logical Switch Port can be a member of only one Logical Port Pair.

ACL

The existing OVN ACL action will be extended to add a sfc action with an external_id to reference the name of the Logical Port Chain with which the ACL is associated.

Networking-sfc / OVN Driver

The networking-sfc / OVN driver maps the Port Chain commands to OVN ovn-nbctl commands.

Port-chain to lport-chain Mapping

A Port-chain is mapped to a single lport-chain.

Port-pair-group to lport-pair-group Mapping

A Port-pair-group is mapped to a single lport-pair-group.

Port-pair to lport-pair Mapping

A Port-pair is mapped to a single lport-pair.

Flow-classifier to OVN ACL Mapping

Flow-classifers will be mapped to OVN ACLs as follows. A flow-classifier is mapped to a single OVN ACL.

When a flow-classifier is created its OVN ACL is created at that time. The OVN ACL is only created when the flow-classifier is associated with the port-chain: Then the driver does:

> acl-add lswitch direction priority match sfc [lchain=<lport-chain>]

When a port-chain is updated to add/remove flow-classifiers then the necessary OVN ACLs are created and deleted.

If a port-chain that has flow-classifiers associated with it is deleted, then the OVN ACLs associated with those flow-classifiers are deleted.

Function Mapping

Port Chain Function	OVN Command	Description
create_port_chain	lchain-add, acl-add	Use acl-add when a port-chain is created with flow-classifiers
delete_port_chain	lchain-del, acl-del	Use acl-del to delete all flow-classifiers associated with a port-chain
update_port_chain	lchain-set-port- pair-group	Use this OVN command when PPGs are added to or removed from a port-chain
	acl-add, acl-del	Use acl-add/del when flow-classifiers are added or removed to a port-chain
create_port_pair_group	lport-pair-group-add	
delete_port_pair_group	lport-pair-group-del	
update_port_pair_group	**lport-pair-group-** set-port-pair	Use this command to add / port-pairs to a PPG
create_port_pair	lport-pair-add	
delete_port_pair	lport-pair-del	
create_flow_classifier	No action	OVN ACLs are only created when flow-classifiers are attached to a port-chain
delete_flow_classifier	No action	

Flow-Classifier Mapping

Flow Classifier	OVN ACL Field
protocol	ip.protocol
ethertype	eth.type
source_port_range_m	If protocol = tcp: min < tcp.src < max, if protocol = udp: min < udp.src < max
destination_port_range_min	If protocol = tcp: min < tcp.dst < max, if protocol = udp: min < udp.dst < max
src_ip_prefix	If ethertype = IPv4: ip4.src/mask, if ethertype = IPv6: ip6.src/mask
destination_ip_prefix	If ethertype = IPv4: ip4.dst/mask, if ethertype = IPv6 ip6.dst/mask
logical_source_port	If the logical-source-port is specified in the classifier then OVN ACL inport= logical_source_port.id and OVN ACL direction=from-port
logical_destination_port	A single asymmetric port chain will use only the logical-source-port, and not the logical-destination-port

A symmetric port chain is defined with a classifier that must have both a logical-source-port and a logical-destination-port. In this case, symmetric forward and reverse OVN port chains are created. The OVN ACL for the forward chain uses the logical-source-port, and the OVN ACL for the reverse chain uses the logical-destination-port.

The OVN ACL for the forward chain has inport=logical-source-port.id and OVN ACL direction=from-port. The OVN ACL for the reverse chain has inport=logical-destination-port.id and OVN ACL direction=from-port.

Implementation

Assignee(s)

Authors of the Specification and Primary contributors:

- Cathy Zhang (cathy.h.zhang@huawei.com)
- Louis Fourie (louis.fourie@huawei.com)
- Farhad Sunavala (farhad.sunavala@huawei.com)
- John McDowall (jmcdowall@paloaltonetworks.com)

5.2.5 OVS Driver and Agent for Symmetric Port Chains

Include the URL of your launchpad blueprint:

https://blueprints.launchpad.net/networking-sfc/+spec/symmetric-port-chain-ovs-agent

This specification describes OVS driver and agent enhancements to support symmetric Port Chains.

Problem Description

Work to add the symmetric parameter to the Port Chain API [1] is in progress. This describes the extensions to the networking-sfc OVS driver and agent to support symmetric Port Chain paths.

Proposed Changes

Two port chain paths are created for a symmetric Port Chain: one path for the forward direction and one for the reverse direction. The SFs in the reverse path (from destination to source) are traversed in reverse order to the SFs in the forward path (from source to destination).

Forward path: SF1 SFn

Reverse path: SFn SF1

A symmetric Port Chain is defined with the symmetric attribute. Both the source and destination Logical Ports must be defined for a symmetric Port Chain. If a Port Chain terminates externally via a vrouter the vrouter port attached to the local subnet is used as the destination Logical Port. When a symmetric Port Chain is deleted both the forward and reverse paths are deleted.

The steering of chain traffic in the data-plane ensures symmetry:

- The source Logical Port in the flow-classifier is used to install OVS rules to match traffic for the forward path. The destination Logical Port in the flow-classifier is used to install OVS rules to match traffic for the reverse path.
- Rules must be installed so that the SFs in the reverse path are traversed in reverse order to that of the forward path.
- Each Port Pair Group must have a Load Balancer pair: one for the forward direction and the other for the reverse direction. In addition, to ensure that traffic in the forward and reverse directions is delivered to the same SF in a Port Pair Group, these LB pairs must use symmetric hash functions.

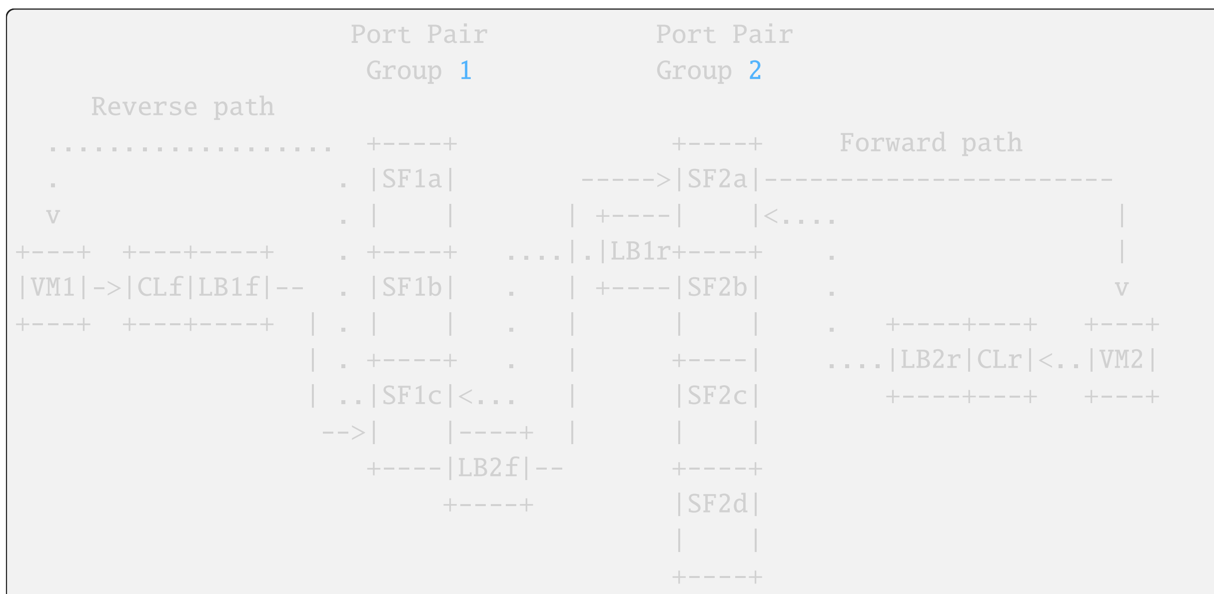
For symmetric hashing, the source and destination fields from packet header used in the hash function of the reverse LB must be the reverse of the packet header fields used in the hash function of the forward LB. If a source field, such as the source IP address, is used as a hash field in the forward direction, the corresponding destination field, the destination IP address, must be used as the hash field in the reverse direction.

The example below shows a symmetric Port Chain that has a forward path and a symmetric reverse path. The Port Chain transits Port Pair Group 1 and Port Pair Group 2. PPG1 consists of service functions SF1a - SF1c, and PPG2 has service functions SF2a - SF2d.

Classification rule CLf matches traffic from the source Logical Port and steers it to the forward path. Classification rule CLr matches traffic from the destination Logical Port and steers it to the reverse path.

Port Pair Group 1 has a pair of Load Balancers, LB1f to load balance traffic in the forward direction, and LB1r to load balance traffic in the reverse direction. Port Pair Group 2 also has a pair of Load Balancers, LB2f and LB2r.

LB1f hashes a certain forward traffic flow to SF1c, and LB1r, using symmetric hashing, hashes the reverse traffic for the same flow to the same SF, SF1c. Similarly, LB2f hashes that forward traffic flow to SF2a, and LB2r hashes the reverse traffic for the same flow to SF2a.



The Load Balancers of the LB pairs may reside on different Compute Nodes. For example, LB1f may be hosted on one Compute Node and LB1r on another Compute Node.

Alternatives

None

Data model impact

None

REST API impact

None

Security impact

None

Notifications impact

None

Other end user impact

None

Performance Impact

None

Other deployer impact

None.

Developer impact

None.

Implementation

Assignee(s)

- Cathy Zhang (cathy.h.zhang@huawei.com)
- Louis Fourie (louis.fourie@huawei.com)
- Farhad Sunavala (farhad.sunavala@huawei.com)

Work Items

1. Extend networking-sfc OVS driver to support symmetric port chains.
2. Add unit tests.
3. Add tempest tests.
4. Update documentation.

Dependencies

None

Testing

Unit tests and function tests will be added.

Documentation Impact

None

References

[1] https://review.openstack.org/#/c/308274/

5.2.6 Service Function Tap for Port Chains

Include the URL of your launchpad blueprint:

https://blueprints.launchpad.net/networking-sfc/+spec/sfc-tap-port-pair

This specification describes the support for passive Service Functions in SFC Port Chains.

Problem Description

There are some Service Functions (SF) that operate in a passive mode and only receive packets on the ingress port but do not send packets on an egress port. An example of this is a Service Function that has an Intrusion Detection Service (IDS). In order to include such a SF in a port chain, the packets must be delivered to this SF and also forwarded on to the next downstream SF in the port chain.

Proposed Changes

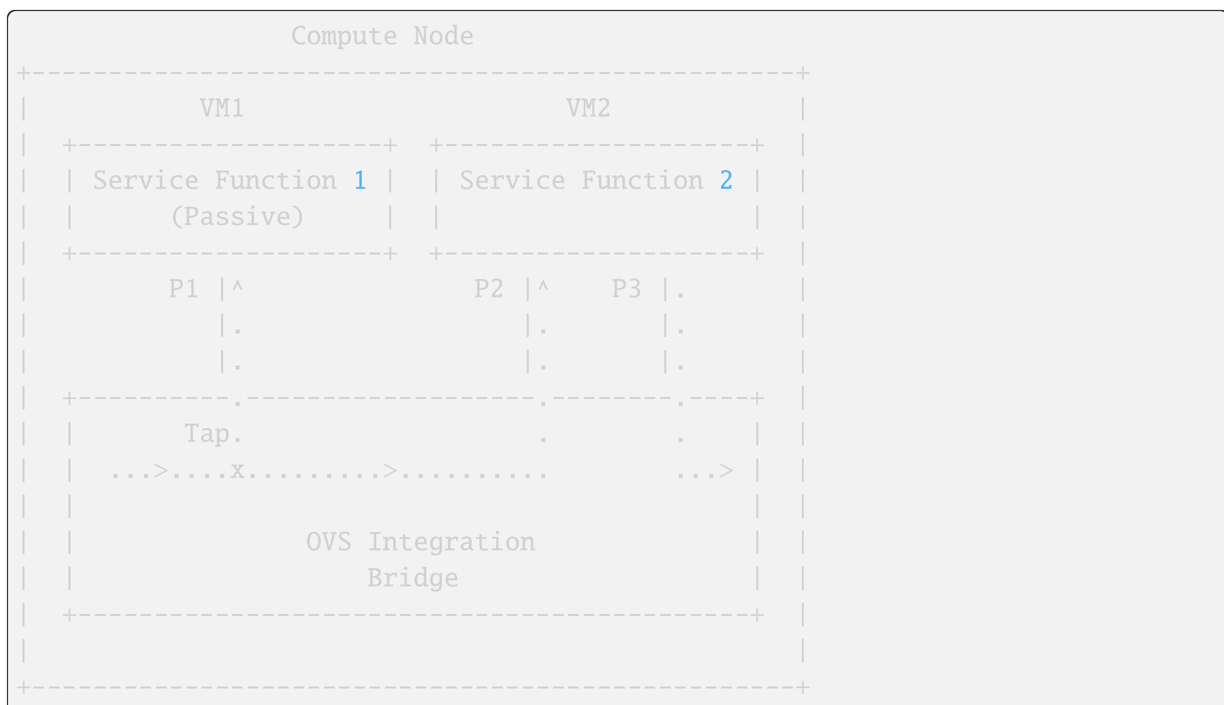
The Port Pair Group port-pair-group-parameter attribute allows service specific configuration to be applied to all Service Functions (Port Pairs) in a Port Pair Group.

The port-pair-group-parameter will be enhanced to add a tap-enabled field. The tap-enabled field will apply to all Service Functions in the Port Pair Group. This field is set to true to indicate that the data-plane switch behavior will be to send the packets to the ingress port of the SF and also forward these packets to the next hop SF. Each Port Pair in the Port Pair Group will act as a tap by passing packets to the passive SF and also forwarding these packets to the next downstream SF. This Port Pair will only send packets to the ingress port of the SF and not receive any packets from the egress port of the SF.

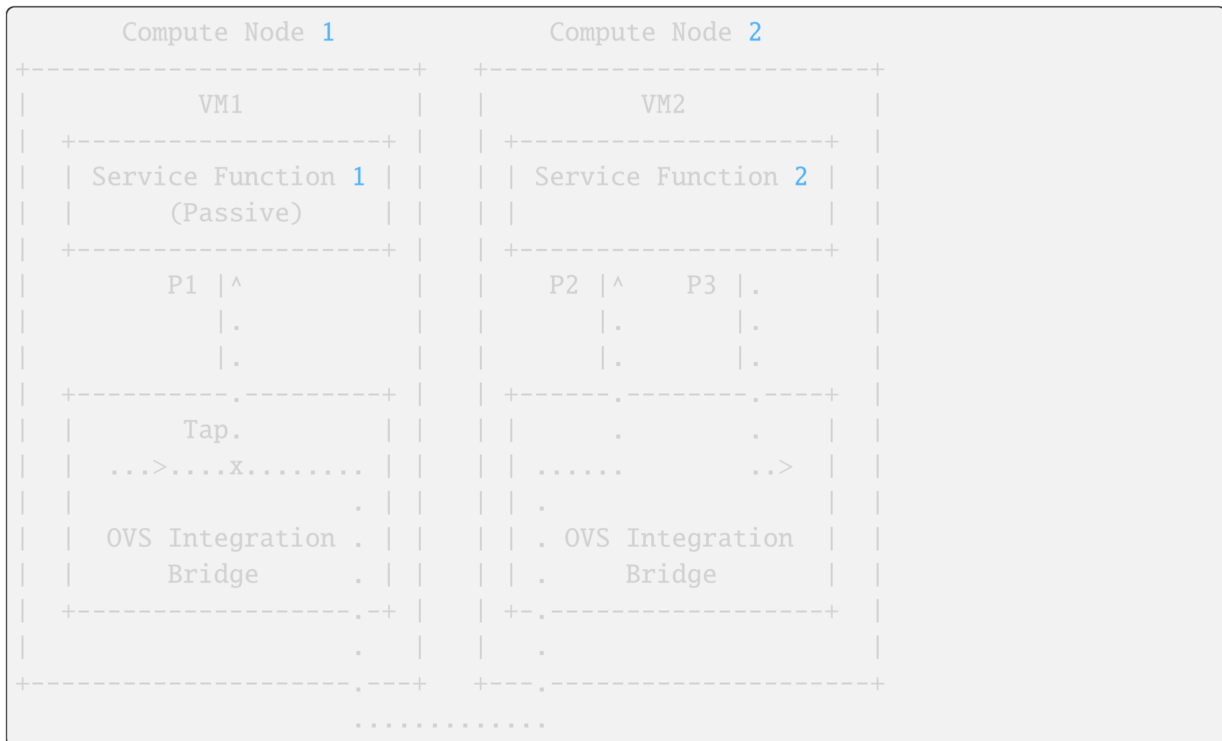
If tap-enabled is set to false or is not present then default behavior will occur. The tap may be applied at any hop (Port Pair Group) in a Port Chain. Every hop in a Port Chain may be configured as a tap.

OVS Driver Implementation

If a SF is configured as a tap the OVS Integration bridge will add a tap to replicate packets received from upstream SFs. One copy is sent to the ingress port (P1) of the passive Service Function (SF 1 on VM1). The other copy is sent to the ingress port (P2) of the next downstream Service Function (SF 2 on VM2).



The tap will work regardless of whether the next hop SF is hosted on the same Compute node as the tap Port Pair as shown above or on another Compute node as shown below.



Workflow & OVS working details for Tap SF

Tap SFs are deployed to monitor/analyze traffic of a network segment. These SFs receive copy of the packet coming out from egress port of default SFs or any logical ports (source/destination) of a service chain.

Steps for Tap Port Pair and Port Pair Group creation:

1. **Create Port**

   openstack sfc port create name p1 net1

2. **Create Port Pair**

   openstack sfc port pair create tap_pp ingress p1 egress p1

3. **Create Port Pair Group**

   openstack sfc port pair group create tap_ppg port-pair tap_pp tap-enabled=True

Apart from sending packet to next-hop SF, the egress port-chain flow in Local Switching Table sends a copy of packet to TAP_CLASSIFIER_TABLE using RESUBMIT action, which does further processing on the Tap destined packet.

Following tables are introduced to process Tap destined traffic:

1. TAP_CLASSIFIER_TABLE (Table 7) - This table classifies traffic based on source mac of SF egress port or any logical port and the IP header (MPLS or IP). VLAN tagging and MPLS encapsulation is done on the packet to send to Tap SF. Based on the location of Tap SF, if on same compute node, action is to resubmit to INGRESS_TABLE. If located on another compute node, action is to output packet to tunnel patch port.

2. TAP_TUNNEL_OUTPUT_TABLE (Table 25) - This table belongs to tunnel bridge or br-tun. This table contains the flows which floods Tap SF destined packets to the tunnel ports.

Alternatives

None

Data model impact

Add tap-enabled to the Port Pair Group parameter. The tap-enabled field is set to true to enable the tap feature. The tap-enabled field is set to false to disable the tap feature.

REST API impact

Add tap-enabled: true to the port-pair-group-parameter.

Security impact

None

Notifications impact

None

Other end user impact

None

Performance Impact

None

Other deployer impact

None.

Developer impact

None.

Implementation

Assignee(s)

- Cathy Zhang (cathy.h.zhang@huawei.com)
- Louis Fourie (louis.fourie@huawei.com)
- Farhad Sunavala (farhad.sunavala@huawei.com)
- Vikash Kumar (vikash.kumar@oneconvergence.com)

Work Items

1. Extend API port-pair-group-parameter to support tap-enabled field.
2. Extend networking-sfc OVS driver to support tap-enabled field.
3. Add unit and functional tests.
4. Update documentation.

Dependencies

None

Testing

Unit tests and functional tests will be added.

Documentation Impact

None

References

None

5.2.7 Non-Transparent Service Functions for Port Chains

URL of the launchpad blueprint:

https://blueprints.launchpad.net/networking-sfc/+spec/sfc-non-transparent-sf

This specification describes the support for non-transparent Service Functions in SFC Port Chains.

Problem Description

Service Functions (SF) that do not support SFC encapsulation, such as NSH, require an SFC Proxy to re-classify a packet that is returned from the egress port of the SF. The SFC Proxy uses the N-tuple values of a packet header to re-classify a packet. The packet N-tuple consists of the following:

- Source IP address
- Destination IP address
- Source TCP/UDP port
- Destination TCP/UDP port
- IP Protocol

However, if the SF is non-transparent (it modifies a part of the N-tuple of a packet), then re-classification cannot be done correctly. See https://datatracker.ietf.org/doc/draft-song-sfc-legacy-sf-mapping/

Proposed Changes

This is an enhancement to the SFC proxy so that it is configured with the N-tuple translation rules of the SF. In other words how the SF translates the ingress Port N-tuple to the egress Port N-tuple of a packet:

SF Ingress port N-tuple => SF Egress port N-Tuple

The SFC Proxy can then adjust for the SF translation rules by using this N-tuple mapping. The SFC Proxy applies the N-tuple mapping to packets received from the egress port of the SF before the re-classification function.

The Port Pair Group port-pair-group-parameter attribute allows service specific configuration to be applied to all Service Functions (Port Pairs) in a Port Pair Group.

The port-pair-group-parameter will be enhanced to add an n-tuple-map. This is an array of ingress-egress N-tuple value pairs: {ingress-N-tuple-value, egress-N-tuple-value} that are the same as the actual translation done by the SF itself.

An example of the CLI format is shown below:

n_tuple_map=source_ip_prefix_ingress=10.0.0.9&
source_ip_prefix_egress=10.0.0.12& protocol_ingress=icmp& protocol_egress=tcp

The SFC Proxy in the OVS Integration Bridge will apply the n-tuple-map to the N-tuple of packets received from the egress port of the SF before they are passed to the re-classification function so that the re-classification rules are matched correctly.

(continues on next page)

(continued from previous page)

Alternatives

None

Data model impact

Add n-tuple-map to the Port Pair Group port-pair-group-parameter attribute.

REST API impact

Add n-tuple-map: N-TUPLE-MAP to the port-pair-group-parameter.

Security impact

None

Notifications impact

None

Other end user impact

None

Performance Impact

None

Other deployer impact

None.

Developer impact

None.

Implementation

Assignee(s)

- Cathy Zhang (cathy.h.zhang@huawei.com)
- Louis Fourie (louis.fourie@huawei.com)

Work Items

1. Extend API port-pair-group-parameter to support n-tuple-map attribute.
2. Extend networking-sfc OVS driver to support n-tuple-map attribute.
3. Add unit and functional tests.
4. Update documentation.

Dependencies

None

Testing

Unit tests and functional tests will be added.

Documentation Impact

None

References

None

5.2.8 IETF SFC Encapsulation

This section explains SFC Encapsulation support in networking-sfc.

The link to Launchpad at [4] is an umbrella for SFC Encapsulation work with the following scope:

- MPLS correlation support (labels exposed to SFs)
- Service Graphs allowing port-chains to be linked together
- The IETF SFC Encapsulation protocol, NSH (exposed to SFs), support
- No NSH Metadata support

SFC Encapsulation is an architectural concept from IETF SFC, which states [1]:

The SFC Encapsulation provides, at a minimum, SFP identification, and is used by the SFC-aware functions, such as the SFF and SFC-aware SFs. The SFC encapsulation is not used for network packet forwarding. In addition to SFP identification, the SFC Encapsulation carries metadata including data-plane context information.

Metadata is a very important capability of SFC Encapsulation, but its out of scope for this umbrella of work in networking-sfc.

Correlation is the term used to correlate packets to chains, in essence it is the Service Function Path (SFP) information that is part of the SFC Encapsulation. Correlation can be MPLS or NSH (SFC Encapsulation).

To clarify, MPLS correlation cannot be strictly called SFC Encapsulation since it doesnt fully encapsulate the packets, amongst other limitations such as available space to carry metadata [1]. However, since it can be used for Service Function Path identification, it is a good workaround to exercise the IETF SFC Encapsulation architectural concept in networking-sfc, when NSH is not desired.

Service Graphs is a concept mentioned in [1] but further defined and refined in [5] that builds on top of Reclassification and Branching (from [1]). Service Graphs make use of the full encapsulation of frames the SFC Encapsulation provides, and the Service Function Path information that is carried by it, to create dependencies between SFPs, making sure that theres no leakage of frames between paths. The figure below outlines the key elements in a Service Graph:

```
      Branch1       Join1
   pc1 --+--> pc2 ------>  pc4
         |           ^
         |           |
          --> pc3 ---

Branch1: pc1 = initial (source)
         pc2 = destination
         pc3 = destination
Join1:   pc2 = source
         pc3 = source
         pc4 = destination
```

Since Port Chains resemble Service Function Paths, with the `chain_id` attribute mapping to a Service Path Identifier (SPI), they are used as the SFPs for the Service Graph, and consequently Service Graphs in networking-sfc allow the creation of dependencies between Port Chains (alongside traffic classification criteria, just like a normal Port Chain, via Flow Classifier).

Terminology

- **Branching Point**: Or branch point, is a point in a Service Graph that leads to new SFPs.
- **Correlation**: Related to SFC Encapsulation, but focused on the fact that a Port Chain (an **SFP**) will be mapped to a unique identifier (the **SPI**) and that the hops of that chain will also have a unique index associated (the **SI**), with the forwarding of traffic based on those two parameters.
- **Destination Chain**: A Port Chain that branches from a previous chain (the **Source Chain**), i.e. a dependent chain. A Destination Chain may also be a **Source Chain**. For traffic to be accepted into a Destination Chain, it has to have come from the **Source Chains** that the Destination Chain depends on plus the Destination Chains own flow classifier (except logical source ports, which will be ignored as that would clash with the traffic coming out of respective Source Chains).
- **Initial Chain**: A Port Chain that is not a **Destination Chain**, but may be a **Source Chain** if its included in a Service Graph. In other words, this chain only matches on a Flow Classifier and takes into account the Logical Source Port defined by it (unlike **Destination Chains**).
- **Joining Point**: A point in a Service Graph that merges multiple incoming branches (**Source Chains**) into the same **Destination Chain**.
- **NSP**: Network Service Path (same as **SPI**).
- **NSI**: Network Service Index (same as **SI**).
- **SFP**: Service Function Path.
- **SI**: Service Index.
- **Source Chain**: The Port Chain that provides a branching point to Destination Chains. A Source Chain may also be an **Initial Chain** or a **Destination Chain**. Traffic that leaves a Source Chain, i.e. the egressing traffic from the last SF of the chain (and encapsulated for that particular chain) will be put into either one or no Destination Chains respective to this Source Chain, depending on whether the flow classifiers of the Destination Chains successfully match on the egressing traffic of the Source Chain.
- **SPI**: Service Path Identifier (numerically identifies an **SFP**).

Usage

In order to create Port Chains with Port Pairs that make use of the NSH correlation (i.e. the Network Service Header (NSH) is exposed to the SFs, so no SFC Proxy is logically instantiated by the networking-sfc backend), the Port Pairs `correlation` service function parameter can be used, by setting it to `nsh` (default is set to `None`):

`service_function_parameters: {correlation: 'nsh'}`

Alternatively, the MPLS correlation can be used as a workaround to NSH:

`service_function_parameters: {correlation: 'mpls'}`

Enabling the MPLS correlation doesnt fully encapsulate frames like NSH would, since the MPLS labels are inserted between the Ethernet header and the L3 protocol.

By default, port-chains always have their correlation set to `mpls`:

`chain_parameters: {correlation: 'mpls'}`

A Port Chain can have Port Pair Groups with MPLS-correlated Port Pairs or Port Pairs with no correlation. However, each Port Pair Group can only group Port Pairs that share the same correlation type (to process each hop and expose their feature set in a consistent and predictable way). The SFC OVS driver and agent are smart enough to only apply SFC Proxies to the hops that require so.

The MPLS correlation is only recommended when using SFC-proxied Port Pair Groups. In order to use NSH, the Port Chain correlation must be set to `nsh` (to clarify, SFC Proxies can also be used with NSH Port Chains, as long as the Port Pairs have no correlation set):

`chain_parameters: {correlation: 'nsh'}`

To create a Service Graph, first create the set of Port Chains that will compose the Service Graph. Then, create the Service Graph itself by referencing the Port Chains needed as a dictionary of source to (list of) destination chains, essentially describing each of the branching points of the chain. The following example, using the OpenStack Client, illustrates this (by creating a graph that starts from an initial chain `pc1` which forks into `pc2` and `pc3`, and then joins back into a single chain `pc4` (if thats what the user intended) using the MPLS correlation (if using NSH, the flows are equivalent but OpenFlow NSH actions and matches are used instead):

```
# we assume that the Neutron ports p0..p4 are already created and bound
$ openstack sfc port pair create --ingress p1 --egress p1  --service-function-
↪parameters correlation=mpls pp1
$ openstack sfc port pair create --ingress p2 --egress p2  --service-function-
↪parameters correlation=mpls pp2
$ openstack sfc port pair create --ingress p3 --egress p3  --service-function-
↪parameters correlation=mpls pp3
$ openstack sfc port pair create --ingress p4 --egress p4  --service-function-
↪parameters correlation=mpls pp4
$ openstack sfc port pair group create --port-pair pp1 ppg1
$ openstack sfc port pair group create --port-pair pp2 ppg2
$ openstack sfc port pair group create --port-pair pp3 ppg3
$ openstack sfc port pair group create --port-pair pp4 ppg4
$ openstack sfc flow classifier create --protocol udp --source-port 2001 --
↪logical-source-port p0 fc1
$ openstack sfc flow classifier create --protocol udp --source-port 2002 --
↪logical-source-port p0 fc2
$ openstack sfc flow classifier create --protocol udp --source-port 2003 --
↪logical-source-port p0 fc3
$ openstack sfc flow classifier create --protocol udp --source-port 2004 --
↪logical-source-port p0 fc4
$ openstack sfc port chain create --port-pair-group ppg1 --flow-classifier --
↪chain-parameters correlation=mpls fc1 pc1
$ openstack sfc port chain create --port-pair-group ppg2 --flow-classifier --
↪chain-parameters correlation=mpls fc2 pc2
$ openstack sfc port chain create --port-pair-group ppg3 --flow-classifier --
↪chain-parameters correlation=mpls fc3 pc3
$ openstack sfc port chain create --port-pair-group ppg4 --flow-classifier --
```

(continues on next page)

(continued from previous page)

```
↪chain-parameters correlation=mpls fc4 pc4
$ openstack sfc service graph create --branching-point pc1:pc2,pc3 --
↪branching-point pc2:pc4 --branching-point pc3:pc4 sg1
```

In the Python language, the dictionary of Port Chains provided above via the OpenStack Client would look like this:

```
{
    'port_chains': {
        'pc1': ['pc2', 'pc3'],
        'pc2': ['pc4'],
        'pc3': ['pc4']
    }
}
```

Note that, because pc2, pc3 and pc4 depend on other chains, their Flow Classifiers Logical Source Ports will be ignored.

To clarify what happens under the hood when using the Open vSwitch driver, lets look at the relevant flows that are generated for the above example:

Table 0:

```
priority=30,udp,tp_src=2001,in_port=10 actions=push_mpls:0x8847,set_field:511-
↪>mpls_label,set_mpls_ttl(255),group:1
priority=30,udp,tp_src=2002,reg0=0x1fe actions=push_mpls:0x8847,set_field:767-
↪>mpls_label,set_mpls_ttl(255),group:2
priority=30,udp,tp_src=2003,reg0=0x1fe actions=push_mpls:0x8847,set_
↪field:1023->mpls_label,set_mpls_ttl(255),group:3
priority=30,udp,tp_src=2004,reg0=0x2fe actions=push_mpls:0x8847,set_
↪field:1279->mpls_label,set_mpls_ttl(255),group:4
priority=30,udp,tp_src=2004,reg0=0x3fe actions=push_mpls:0x8847,set_
↪field:1279->mpls_label,set_mpls_ttl(255),group:4
priority=30,mpls,in_port=11,mpls_label=510 actions=load:0x1fe->NXM_NX_REG0[],
↪pop_mpls:0x0800,resubmit(,0)
priority=30,mpls,in_port=12,mpls_label=766 actions=load:0x2fe->NXM_NX_REG0[],
↪pop_mpls:0x0800,resubmit(,0)
priority=30,mpls,in_port=13,mpls_label=1022 actions=load:0x3fe->NXM_NX_REG0[],
↪pop_mpls:0x0800,resubmit(,0)
priority=30,mpls,in_port=14,mpls_label=1278 actions=pop_mpls:0x0800,NORMAL
```

Table 5: (usual flows for sending to table 10 or across tunnel, without proxying)

Table 10: (usual flows to make traffic ingress into the Service Functions, shown below):

```
priority=1,mpls,dl_vlan=1,dl_dst=fa:16:3e:97:91:a2,mpls_label=511 actions=pop_
↪vlan,output:11
priority=1,mpls,dl_vlan=1,dl_dst=fa:16:3e:87:2a:ad,mpls_label=767 actions=pop_
↪vlan,output:12
priority=1,mpls,dl_vlan=1,dl_dst=fa:16:3e:77:59:f1,mpls_label=1023␣
↪actions=pop_vlan,output:13
```

(continues on next page)

(continued from previous page)

```
priority=1,mpls,dl_vlan=1,dl_dst=fa:16:3e:34:07:f5,mpls_label=1279␣
↪actions=pop_vlan,output:14
```

Groups Table: (usual flows for load-balancing and re-writing the destination MAC addresses)

Considering that the OF port 10 is p0, 11 is p1, and so on with 14 being p4, there are three important things to notice from the Service Graphs flows above:

- At the end of the Source Chains (pc1, pc2 and pc3), instead of the typical flow (in table 0) that would remove the MPLS shim (with `pop_mpls`) and then use the NORMAL action, the chains SFP information is written to a register (e.g. `actions=load:0x1fe->NXM_NX_REG0[]`) and the packet is sent back to the same table to be matched by a Destination Chain.
- At the beginning of the Destination Chains (pc2, pc3 and pc4), instead of the typical flow (in table 0) that would match solely on the Flow Classifier (specifically the ingress OF port that comes from the Logical Source Port together with the actual traffic classification definition), a specific SFP information register value will be matched on (e.g. `reg0=0x1fe`) together with the traffic classification definition from the Flow Classifier but not OF ingress port will be used (i.e. Logical Source Port ignored).
- For the case of Joining Points, where a chain is Destination to multiple Source Chains, there will be one flow matching on the register value per Source Chain, the only difference in the entire flow being the value of that register (reflecting each of the Source Chains SFP infos). Two flows can be seen above in table 0, matching on traffic meant for pc4.

Implementation

PPG/SF Correlation

At the API side, both MPLS and NSH correlations are defined as possible options (values) to the `correlation` key in the `service_function_parameters` field of the `port_pair` resource. Furthermore, Port Pair Groups must include Port Pairs of the same correlation type.

The parameter is saved in the database in the same way as any other port-pair parameter, inside the `sfc_service_function_params` table (example for NSH):

```
keyword='correlation'
value='nsh'
pair_id=PORT_PAIR_UUID
```

The NSH correlation parameter will eventually be fed to the enabled backend, such as Open vSwitch. Through the OVS SFC driver and agent, the vswitches on the multiple nodes where networking-sfc is deployed will be configured with the set of flows that allow classification, encapsulation, decapsulation and forwarding of MPLS tagged or untagged packets. Applying the IETF SFC view to this, Open vSwitch switches thus implement the logical elements of Classifier, Service Function Forwarder (SFF) and SFC Proxy (stateless) [1].

In networking-sfc, the OVS driver talks to the agents on the multiple compute nodes by sending flow rule messages to them across the RPC channels.

In flow rules, correlation parameters of both port-chains and port-pairs are specified using the `pc_corr` and `pp_corr` flow rule keys, respectively. Moreover, a `pp_corr` key is also specified in each of the hops of the `next_hops` flow rule key.

Remember: a port-pair-group contains port-pairs that all share the same correlation type, so the comparison between `pc_corr` and each of the `pp_corr` of the next hops will yield the same result.

`pc_corr` is the correlation mechanism (SFC Encapsulation) to be used for the entire port-chain. The values may be `None`, `'mpls'`, or `'nsh'`.

`pp_corr` is the correlation mechanism supported by an individual SF. The values may be `'None'`, `'mpls'`, or `'nsh'`.

The backend driver compares `pc_corr` and `pp_corr` to determine if SFC Proxy is needed for a SF that is not capable of processing the SFC Encapsulation mechanism. For example, if `pc_corr` is `'mpls'` and `pp_corr` is `None`, then SFC Proxy is needed.

The following is an example of an sf_node flow rule (taken from one of the SFC OVS agents unit tests):

```
'nsi': 255,
'ingress': '6331a00d-779b-462b-b0e4-6a65aa3164ef',
'next_hops': [{
    'local_endpoint': '10.0.0.1',
    'ingress': '8768d2b3-746d-4868-ae0e-e81861c2b4e6',
    'weight': 1,
    'net_uuid': '8768d2b3-746d-4868-ae0e-e81861c2b4e7',
    'network_type': 'vxlan',
    'segment_id': 33,
    'gw_mac': '00:01:02:03:06:09',
    'cidr': '10.0.0.0/8',
    'mac_address': '12:34:56:78:cf:23',
    'pp_corr': 'nsh'
}],
'del_fcs': [],
'group_refcnt': 1,
'node_type': 'sf_node',
'egress': '29e38fb2-a643-43b1-baa8-a86596461cd5',
'next_group_id': 1,
'nsp': 256,
'add_fcs': [{
    'source_port_range_min': 100,
    'destination_ip_prefix': u'10.200.0.0/16',
    'protocol': u'tcp',
    'l7_parameters': {},
    'source_port_range_max': 100,
    'source_ip_prefix': '10.100.0.0/16',
    'destination_port_range_min': 100,
    'ethertype': 'IPv4',
    'destination_port_range_max': 100,
}],
'pc_corr': 'nsh',
'pp_corr': 'nsh',
'id': uuidutils.generate_uuid()
```

It can be seen that `'nsh'` appears three times in the flow rule, twice in the root (specifying the correlation of port-chain and port-pair of the current hop) and once inside the single hop of `next_hops`, regarding its port-pair.

The three appearances will dictate how flows (both matches and actions) will be added by the OVS agent.

Lets take a look at the possible scenarios:

	Curr Hop pp_corr	Next Hop pp_corr	Action
1	NSH/MPLS	NSH/MPLS	Egress from SF: match on NSH/MPLS to determine next hop Ingress to next SF: send NSH/MPLS to SF
2	NSH/MPLS	None	Egress from SF: match on NSH/MPLS to determine next hop Ingress to next SF: pop NSH/MPLS first
3	None	NSH/MPLS	Egress from SF: reclassify packet and add new NSH/MPLS Ingress to next SF: send NSH/MPLS to SF
4	None	None	Egress from SF: reclassify packet and add new NSH/MPLS Ingress to next SF: pop NSH/MPLS first

An important point to make is that correlations cannot be mixed, i.e. if the Port Chain uses the MPLS correlation, then its PPGs cannot include Port Pairs using the NSH correlation, and vice-versa. So, on the table above, consider either NSH or MPLS for any given row, but not both.

The following further explains each of the possibilities from the table above. To simplify, the NSH correlation is considered (MPLS is equivalent here).

1. **pp_corr=nsh and every next_hops pp_corr=nsh**

The ingress of this sf_node will not remove the NSH. When egressing from this sf_node, OVS will not attempt to match on the flow_classifier defined in `add_fcs`, but rather the expected NSH after the SF is done processing the packet (the NSI is supposed to be decremented by 1 by the SF). When preparing the packet to go to the next hop, no attempt at inserting NSH will be done, since the packet already has the correct labels.

2. **pp_corr=nsh and every next_hops pp_corr=None**

The ingress of this sf_node will not remove the NSH. When egressing from this sf_node, OVS will not attempt to match on the flow_classifier defined in `add_fcs`, but rather the expected NSH after the SF is done processing the packet (the NSI is supposed to be decremented by 1 by the SF). When preparing the packet to go to the next hop, no attempt at inserting NSH will be done, since the packet already has the correct labels. The next hops own flow rule (not the one shown above) will have an action to first remove the NSH and then forward to the SF.

3. **pp_corr=None and every next_hops pp_corr=nsh**

The ingress of this sf_node will first remove the NSH and then forward to the SF, as its actions. When egressing from this sf_node, OVS will match on the flow-classifier defined in `add_fcs`, effectively implementing an SFC Proxy and running networking-sfcs classic mode. When preparing the packet to go to the next hop, a new NSH needs to be inserted. This is done on Table 0, the same table where `add_fcs` was matched. Right before the packets are submitted to the Groups Table, they receive the expected NSH for the next hop. The reason why this cant be done on the `ACROSS_SUBNET_TABLE` like when the next_hops correlation is set to None, is the fact that the choice of labels would be ambiguous. If multiple port-chains share the same port-pair-group at a given hop, then encapsulating/adding NSH as one of `ACROSS_SUBNET_TABLE`s actions means that at least one of port-chains will be fed the wrong label and, consequently, leak into a different port-chain. This is due to the fact that, in `ACROSS_SUBNET_TABLE`, the flow matches only on the destination MAC address of the frame (and that isnt enough to know what chain the frame is part of). So, again, the encapsulation/adding of NSH will have to be done in Table 0

for this specific scenario where in the current hop the packets dont have labels but on the next hop they are expected to.

4. **pp_corr=None and every next_hops pp_corr=None**

This is classic networking-sfc. The ingress of this sf_node will first remove the NSH and then forward to the SF, as its actions. When egressing from this sf_node, OVS will match on the flow-classifier defined in `add_fcs` effectively implementing an SFC Proxy and running networking-sfcs classic mode. When preparing the packet to go to the next hop, a new NSH needs to be inserted, which is done at the `ACROSS_SUBNET_TABLE`, after a destination port-pair has been chosen with the help of the Groups Table.

Service Graphs

At the API side, Service Graphs are presented as a specific resource called `service_graph`. Besides the attributes `id`, `name`, `description` and `project_id`, this resource expects to have a dictionary called `port_chains` that maps source chains to (lists of) destination chains.

Service Graphs glue existing Port Chains, creating dependencies between them, in effect changing the criteria to get into each of the chains by not relying solely on the Flow Classifier anymore (except for the initial chain of the graph). Traffic entering a destination chain of a Service Graph is dependent on its source chain and its own flow classifiers.

In the database, Service Graphs are stored as 2 tables:

- `sfc_service_graphs`: This table stores the independent data of each of the Service Graph resources, specifically the name, description and project ID.
- `sfc_service_graph_chain_associations`: This table stores the actual associations between Service Graphs and Port Chains, stating which ones are source chains and which ones are destination chains. Besides the `service_graph_id` field (primary key, and foreign key to `sfc_service_graphs.id`), there are the `src_chain` and the `dst_chain` fields, each pointing to an ID of a Port Chain, both being foreign keys to `sfc_port_chains.id`.

So, to represent the branching points of the example graph provided in the Usage section above, the following entries would be stored in `sfc_service_graph_chain_associations`:

service_graph_id	src_chain	dst_chain
SG1 ID SG1 ID SG1 ID SG1 ID	PC1 ID PC1 ID PC2 ID PC3 ID	PC2 ID PC3 ID PC4 ID PC4 ID

Some of the validations that occur at the database/plugin level are:

- Port Chains cant be deleted if they are in use by a graph.
- Port Chains cant be updated (to include a different set of Port Pair Groups) if they are in use by a graph.
- Service Graphs cant have Port Chain loops or circular paths.
- A Port Chain cant be added twice as destination of the same source chain (that would essentially replicate packets).
- Port Chains cannot be part of more than one graph at any given time.
- Branching points have to support a correlation protocol (MPLS or NSH).
- The correlation protocol has to be the same for every included Port Chain.

- For a given branching point (destination chain), the traffic classification of each branch has to be different to prevent ambiguity.

At the OVS driver level, all of the logic takes place in the postcommit methods, `create_service_graph_postcommit` and `delete_service_graph_postcommit`. At present time, the dictionary of Port Chains that a Service Graph references cannot be updated and, as such, the drivers (not just OVS) dont have to support the update operation.

In essence, the OVS driver will look at the `port_chains` dictionary of the graph and generate flow rules for every branching point. Each branching point includes both the last path node (the last `sf_node`) of the respective source chain and each first path node (the `src_node`) of the respective destination chains. All of these flow rules are meant to replace the flows that the original flow rules (during creation of the Port Chains themselves) had requested the agent to create.

The flow rules for the source chains will include a special attribute called `branch_point`, set to the value of `True`. This indicates to the agent that this path nodes (expected to be the last `sf_node` of that chain) NSP and NSI should be saved so that the destination chains can match on them while doing the normal traffic classification (via their own Flow Classifiers). Example:

```
'branch_point': True
```

The flow rules for the destination chains will include a special attribute called `branch_info`, a dictionary with two keys: `matches` and `on_add`. Example:

```
'branch_info': {
    'matches': set([(2, 254), (3, 254)]),
    'on_add': True
}
```

`matches` contains a set of tuples with the NSP and NSI (`(<nsp>, <nsi>)`) to be matched by the particular destination chain. `on_add` simply specifies whether the `matches` should be used when adding the flow or otherwise when removing the flow - in very much the same fashion as `add_fcs/del_fcs` for the Flow Classifiers, except that here its either adding or removing the NSP/NSI matches and never replacing/updating them.

For source chains `branch_point` there is no need to have an `on_add` since the OpenFlow matches will not change depending on whether we are removing or adding this branch point. Only the actions will change (for relevant flows in Table 0).

At the OVS agent level, `branch_point` and `branch_info` are interpreted in order to generate the appropriate set of flows, replacing the ones originally created by the constituent Port Chains (to clarify, only the flows at the branching points).

`'branch_point': True` will tell the agent to replace the egress flow from the last `sf_node`, in Table 0, with a new one whose actions will be to: * copy the NSP and NSI from the MPLS label or NSH into a register: `reg0`; * remove the MPLS label or NSH; * send the traffic back to Table 0, now without MPLS/NSH but with `reg0` set. Example of this flow (using MPLS correlation):

```
table=0,priority=30,mpls,in_port=8,mpls_label=509 actions=load:0x1fd->NXM_NX_
↪REG0[],pop_mpls:0x0800,resubmit(,0)
```

When `branch_info` is set, with `'on_add': True` and `'matches': set([(1, 253)])`, the agent will replace the egress flow from the `src_node` of the destination chain that is specified in the flow rule, in Table 0, with a different set of matches from a typical `src_node`: * it will still match on what the Flow Classifiers specify; * but the logical source port match is ignored (there is not in_port=X); * most

importantly, it will match on a specified value of `reg0` (NSP/NSI). Example of this flow (using MPLS correlation):

```
table=0,priority=30,udp,reg0=0x1fd actions=push_mpls:0x8847,set_field:767->
↪mpls_label,set_mpls_ttl(255),group:3
```

With `'on_add': False`, the agent will replace the above flow with the original flow for the `src_node` of that Port Chain, matching only on the Flow Classifiers fields.

Known Limitations

- Service Graphs is not compatible with Symmetric Port Chains at the moment. Furthermore, Service Graphs are unidirectional;
- The MPLS correlation protocol does not provide full frame encapsulation, so the SFC Encapsulation NSH protocol should be used instead;
- Every Port Chain has to have a different set of Flow Classifiers, even if the logical source ports are different, even when they are attached to Service Graphs. This is necessary when deploying Port Chains that have Port Pairs with no correlation protocol (to prevent per-hop classification ambiguity), but is a limitation otherwise and hasnt been addressed yet;
- SI/NSI is only available at the Open vSwitch driver level, meaning that the networking-sfc API cant consistently manage and persist all of the SFP information (only SPI/NSP) independently of the driver. SI/NSI and SPI/NSP are used by the logical Service Function Forwarders (SFF) that the drivers are expected to control.

References

[1] https://datatracker.ietf.org/doc/rfc7665/?include_text=1

[2] http://i.imgur.com/rxzNNUZ.png

[3] http://i.imgur.com/nzgatKB.png

[4] https://bugs.launchpad.net/networking-sfc/+bug/1587486

[5] https://datatracker.ietf.org/doc/draft-ietf-sfc-nsh/?include_text=1

5.2.9 Exclusive Port-Pair Group for Non-Transparent Service Functions

URL of the launchpad blueprint:

https://blueprints.launchpad.net/networking-sfc/+spec/sfc-proxy-port-correlation

This specification describes the support for non-transparent Service Functions in SFC Port Chains using a SFC Port Pair Group that is used exclusively by one Port Chain. Non-transparent Service Functions modify the N-tuple header fields of a packet.

Problem Description

Most legacy Service Functions (SF) do not support SFC encapsulation, such as NSH, and therefore require an SFC Proxy to re-classify a packet that is returned from the egress port of the SF. The SFC Proxy uses the N-tuple values of a packet header to re-classify a packet. The packet N-tuple consists of the following:

- Source IP address
- Destination IP address
- Source TCP/UDP port
- Destination TCP/UDP port
- IP Protocol

However, if the SF is non-transparent (it modifies a part of the N-tuple of a packet), then re-classification cannot be done correctly. See https://datatracker.ietf.org/doc/draft-song-sfc-legacy-sf-mapping/

In addition the SF may dynamically change the mapping of the N-tuple values as the SF operations progress. A mechanism that uses a static N-tuple mapping to adjust for N-tuple changes cannot be employed.

Proposed Changes

This is an enhancement to the SFC proxy so that it can handle the dynamic changes to N-tuple translation rules of the SF.

A solution to the non-transparent SF is to use a SF VM that has multiple instances and assign the port-pairs for each SF instance to a separate Port Chain.

This can be done by adding these ports to a SFC Proxy Port Pair Group which operates as a Port Pair Correlation Map instead of a normal Load Distribution function. The Proxy Port Pair Group is configured with multiple Port Pairs that are attached to the SF Instances of a specific non-transparent SF type, such as a Firewall SF. This Port Pair Group is configured to operate as a Port Pair Correlation Map.

Each non-transparent SF instance is attached to a single Port Pair. These SF instances may either run on a VM or on a container within a VM. If an SF instance runs within a container, the container sub-port ([1][2]) is used as the ingress and/or egress port of the Port Pair.

Each Port Chain is mapped to one of these port-pairs. Packets for a Port Chain arriving at the OVS Integration bridge are steered to the ingress port of the Port Pair assigned to that Port Chain. Packets received back from the SF on its egress port are then mapped back to the corresponding Port Chain. This mechanism avoids the need for the SFC Proxy to re-classify packets returned from the egress port of the non-transparent SF.

For example, in the figure below, packets on Port Chain A are steered to Port Pair 1 and sent to the ingress port of SF Instance 1. Packets from the egress port of SF Instance 1 are then mapped back to Port Chain A and are delivered to the next hop in the chain.

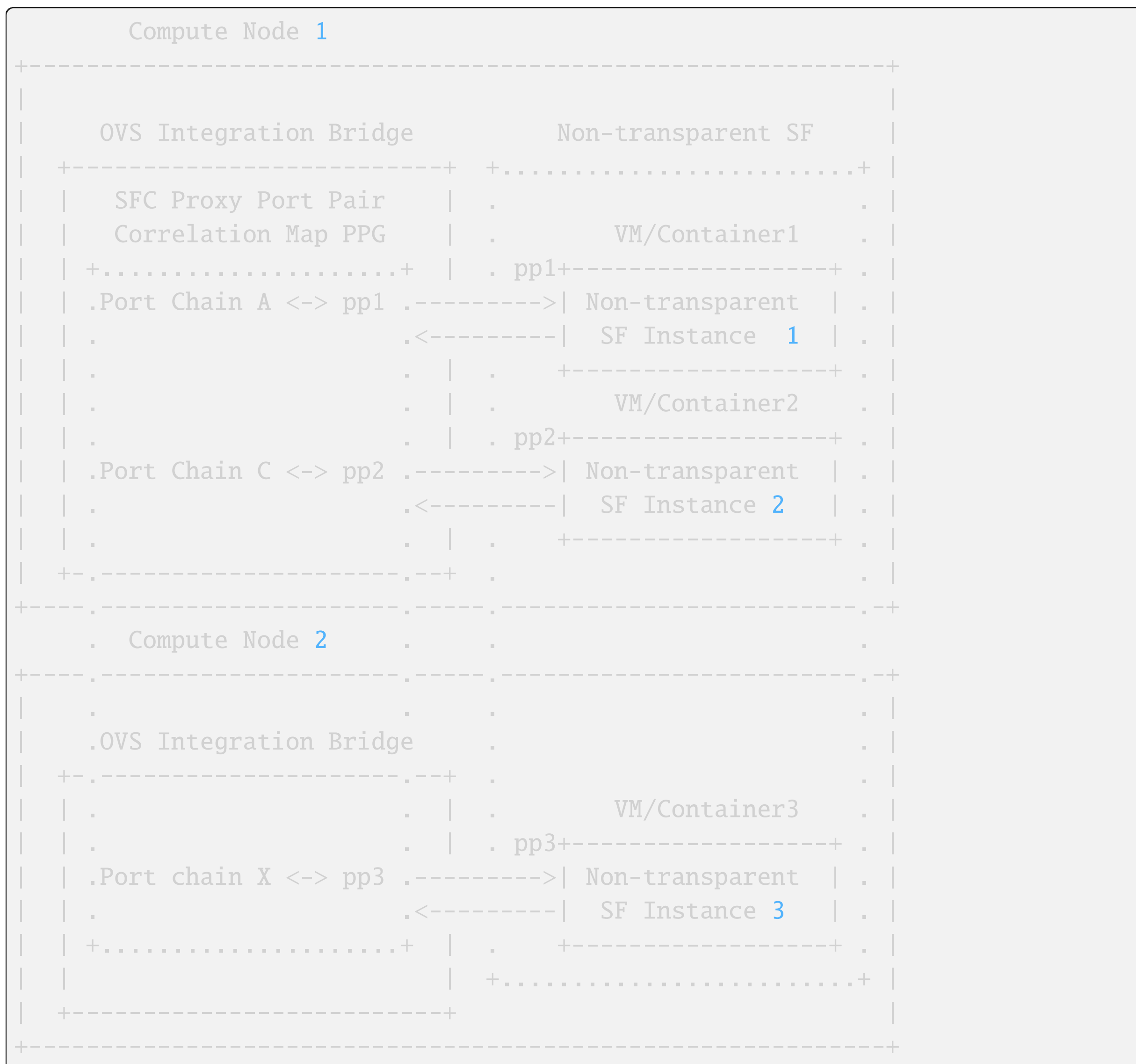
When a Port Chain is created (or updated) that uses a SFC Proxy PPG, the Port Chain is assigned to one of the Port Pairs in the PPG and the Port Pair is reserved for that Port Chain. If the Port Chain is deleted or the PPG is removed from the Port Chain, its Port Pair becomes available for use by another Port Chain.

The Port Pairs in the SFC Proxy Port Pair Group may be hosted on different Compute Nodes as shown in the diagram below.

If a Port Chain is created that uses a SFC Proxy Port Pair Group and all the Pairs in that PPG are in use by other Port Chains, an error Maximum number of Port Chains reached is returned.

This obviously requires that multiple instances of the non-transparent SF be deployed in either VMs or containers. The number of SF instances that must be deployed and configured as Port Pairs depends on the maximum number of Port Chains that are expected to use that particular SF. However, deploying multiple instances of a SF is easily done in modern data centers.

A Port Chain may include multiple SFC Proxy PPGs, each one for a different type of non-transparent SF. For example PPG1 may be a group of non-transparent Firewall SF instances and PPG2 may be a group of non-transparent HTTP Optimizer SF instances.



Alternatives

An alternative mechanism for non-transparent SFs is to mark PPG as exclusive so that it is assigned to one port chain only. This would require a PPG be created for each port chain. The advantage to this approach is that the PPG can be used for load balancing.

Data model impact

Add a proxy-correlation-map attribute to the Port Pair Group. This is a Boolean that will enable the Proxy Port Correlation. Add an exclusive attribute to the Port Pair Group. This is a Boolean that will enable exclusive use of a Port Pair Group by one Port Chain.

REST API impact

Add proxy-correlation-map: true to the Port Pair Group. Add exclusive: true to the Port Pair Group.

Security impact

None

Notifications impact

None

Other end user impact

None

Performance Impact

None

Other deployer impact

None.

Developer impact

None.

Implementation

Assignee(s)

- Cathy Zhang (cathy.h.zhang@huawei.com)
- Louis Fourie (louis.fourie@huawei.com)

Work Items

1. Extend API port-pair-group-parameter to support proxy-correlation-map and the exclusive attributes. 2. Extend networking-sfc OVS driver to support proxy-correlation-map and exclusive attributes. 3. Add unit and functional tests. 4. Update documentation.

Dependencies

None

Testing

Unit tests and functional tests will be added.

Documentation Impact

None

References

[1] Neutron Trunk-port https://wiki.openstack.org/wiki/Neutron/TrunkPort

[2] VLAN aware VMs https://review.openstack.org/#/c/243786/11/specs/mitaka/vlan-aware-vms.rst