ODL SFC

Release master

OpenDaylight Project

Mar 19, 2020

CONTENTS

This documentation provides critical information needed to help you write ODL Applications/Projects that can co-exist with other ODL Projects.

Contents:

CHAPTER

ONE

SERVICE FUNCTION CHAINING DEVELOPER GUIDE

1.1 OpenDaylight Service Function Chaining (SFC) Overview

OpenDaylight Service Function Chaining (SFC) provides the ability to define an ordered list of a network services (e.g. firewalls, load balancers). These service are then "stitched" together in the network to create a service chain. This project provides the infrastructure (chaining logic, APIs) needed for ODL to provision a service chain in the network and an end-user application for defining such chains.

- ACE - Access Control Entry
- ACL - Access Control List
- SCF - Service Classifier Function
- SF - Service Function
- SFC - Service Function Chain
- SFF - Service Function Forwarder
- SFG - Service Function Group
- SFP - Service Function Path
- RSP - Rendered Service Path
- NSH - Network Service Header

1.2 SFC Classifier Control and Date plane Developer guide

1.2.1 Overview

Description of classifier can be found in: https://datatracker.ietf.org/doc/draft-ietf-sfc-architecture/

Classifier manages everything from starting the packet listener to creation (and removal) of appropriate ip(6)tables rules and marking received packets accordingly. Its functionality is **available only on Linux** as it leverages **NetfilterQueue**, which provides access to packets matched by an **iptables** rule. Classifier requires **root privileges** to be able to operate.

So far it is capable of processing ACL for MAC addresses, ports, IPv4 and IPv6. Supported protocols are TCP and UDP.

1.2.2 Classifier Architecture

Python code located in the project repository sfc-py/common/classifier.py.

Note: classifier assumes that Rendered Service Path (RSP) **already exists** in ODL when an ACL referencing it is obtained

1. sfc_agent receives an ACL and passes it for processing to the classifier
2. the RSP (its SFF locator) referenced by ACL is requested from ODL
3. if the RSP exists in the ODL then ACL based iptables rules for it are applied

After this process is over, every packet successfully matched to an iptables rule (i.e. successfully classified) will be NSH encapsulated and forwarded to a related SFF, which knows how to traverse the RSP.

Rules are created using appropriate iptables command. If the Access Control Entry (ACE) rule is MAC address related both iptables and IPv6 tables rules are issued. If ACE rule is IPv4 address related, only iptables rules are issued, same for IPv6.

Note: iptables **raw** table contains all created rules

Information regarding already registered RSP(s) are stored in an internal data-store, which is represented as a dictionary:

```
{rsp_id: {'name': <rsp_name>,
          'chains': {'chain_name': (<ipv>,),
                     ...
                     },
          'sff': {'ip': <ip>,
                  'port': <port>,
                  'starting-index': <starting-index>,
                  'transport-type': <transport-type>
                  },
          },
...
}
```

- `name`: name of the RSP
- `chains`: dictionary of iptables chains related to the RSP with information about IP version for which the chain exists
- `SFF`: SFF forwarding parameters
  - `ip`: SFF IP address
  - `port`: SFF port
  - `starting-index`: index given to packet at first RSP hop
  - `transport-type`: encapsulation protocol

1.2.3 Key APIs and Interfaces

This features exposes API to configure classifier (corresponds to service-function-classifier.yang)

1.2.4 API Reference Documentation

See: sfc-model/src/main/yang/service-function-classifier.yang

1.3 SFC-OVS Plug-in

1.3.1 Overview

SFC-OVS provides integration of SFC with Open vSwitch (OVS) devices. Integration is realized through mapping of SFC objects (like SF, SFF, Classifier, etc.) to OVS objects (like Bridge, TerminationPoint=Port/Interface). The mapping takes care of automatic instantiation (setup) of corresponding object whenever its counterpart is created. For example, when a new SFF is created, the SFC-OVS plug-in will create a new OVS bridge and when a new OVS Bridge is created, the SFC-OVS plug-in will create a new SFF.

1.3.2 SFC-OVS Architecture

SFC-OVS uses the OVSDB MD-SAL Southbound API for getting/writing information from/to OVS devices. The core functionality consists of two types of mapping:

a. mapping from OVS to SFC
   - OVS Bridge is mapped to SFF
   - OVS TerminationPoints are mapped to SFF DataPlane locators

b. mapping from SFC to OVS
   - SFF is mapped to OVS Bridge
   - SFF DataPlane locators are mapped to OVS TerminationPoints

1.3.3 Key APIs and Interfaces

- SFF to OVS mapping API (methods to convert SFF object to OVS Bridge and OVS TerminationPoints)
- OVS to SFF mapping API (methods to convert OVS Bridge and OVS TerminationPoints to SFF object)

1.4 SFC Southbound REST Plug-in

1.4.1 Overview

The Southbound REST Plug-in is used to send configuration from datastore down to network devices supporting a REST API (i.e. they have a configured REST URI). It supports POST/PUT/DELETE operations, which are triggered accordingly by changes in the SFC data stores.

- Access Control List (ACL)
- Service Classifier Function (SCF)

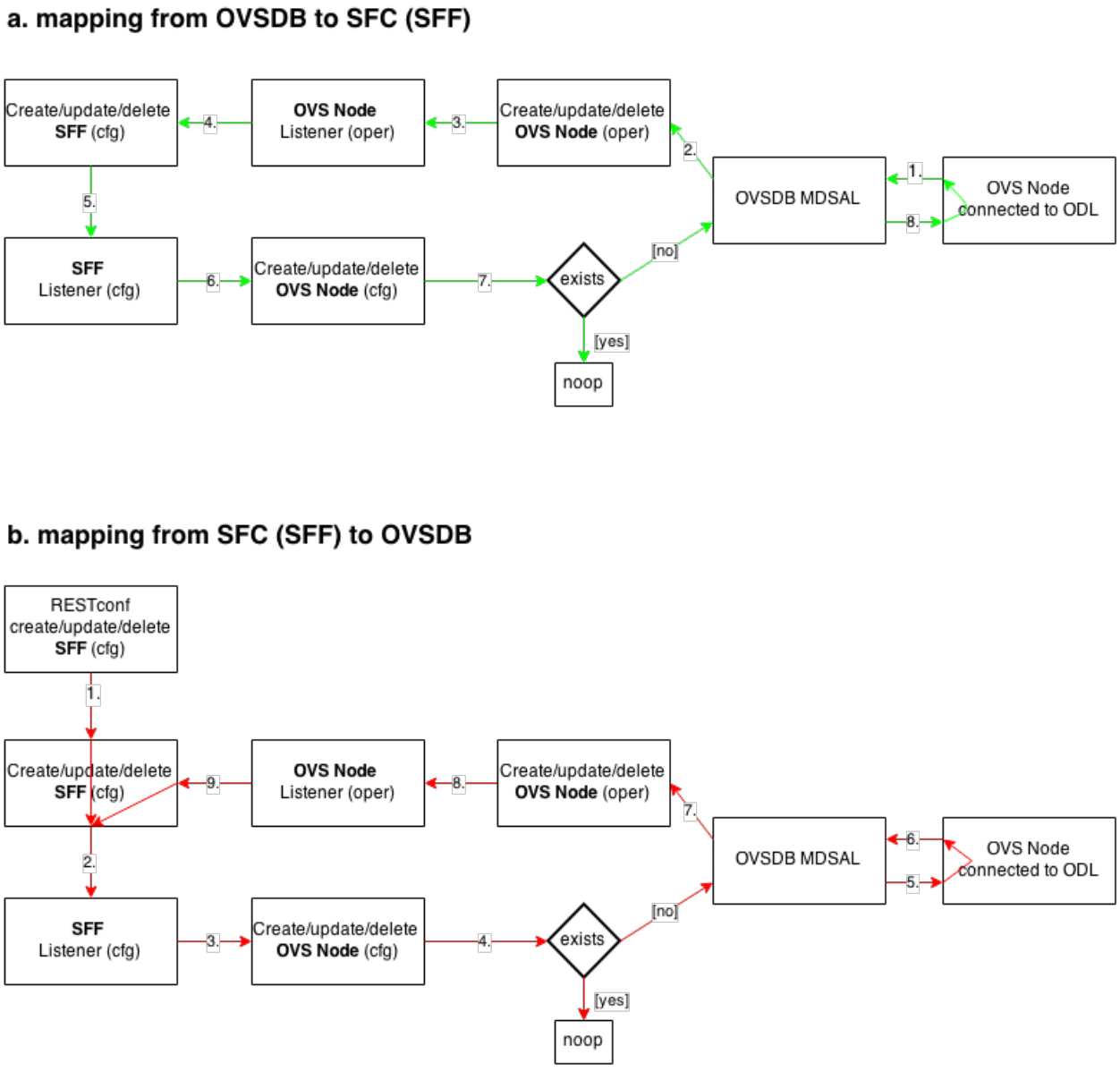


Fig. 1: SFC <—> OVS mapping flow diagram

- Service Function (SF)
- Service Function Group (SFG)
- Service Function Schedule Type (SFST)
- Service Function Forwarder (SFF)
- Rendered Service Path (RSP)

1.4.2 Southbound REST Plug-in Architecture

1. **listeners** - used to listen on changes in the SFC data stores
2. **JSON exporters** - used to export JSON-encoded data from binding-aware data store objects
3. **tasks** - used to collect REST URIs of network devices and to send JSON-encoded data down to these devices

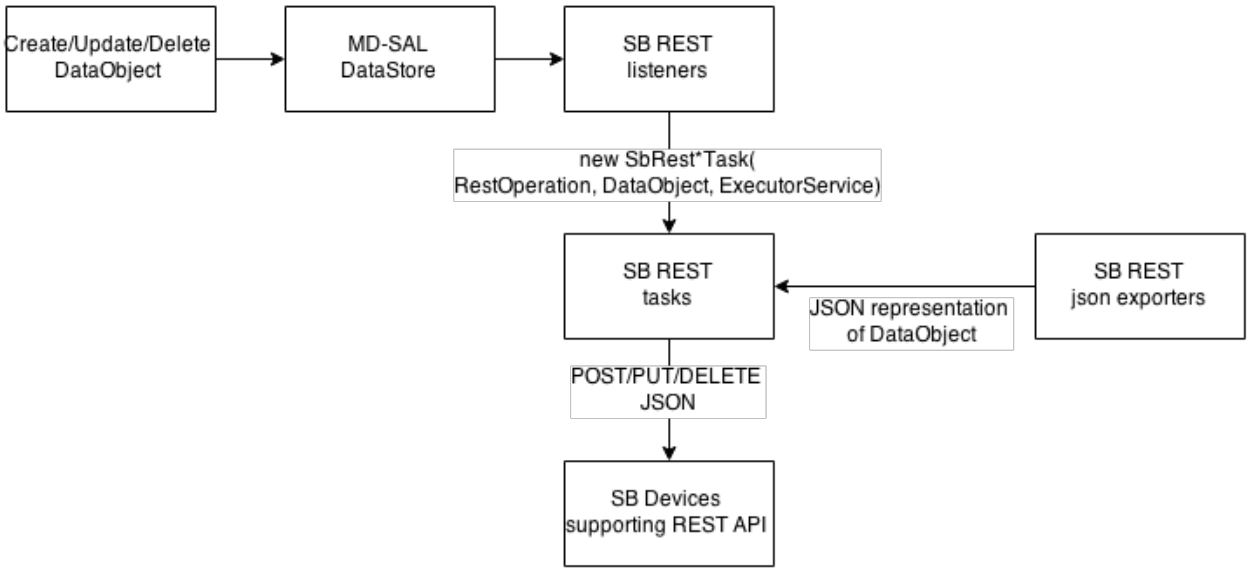


Fig. 2: Southbound REST Plug-in Architecture diagram

1.4.3 Key APIs and Interfaces

The plug-in provides Southbound REST API designated to listening REST devices. It supports POST/PUT/DELETE operations. The operation (with corresponding JSON-encoded data) is sent to unique REST URL belonging to certain data type.

- Access Control List (ACL): `http://<host>:<port>/config/ietf-acl:access-lists/access-list/`
- Service Function (SF): `http://<host>:<port>/config/service-function:service-functions/service-function/`
- Service Function Group (SFG): `http://<host>:<port>/config/service-function:service-function-groups/service-function-group/`
- Service Function Schedule Type (SFST): `http://<host>:<port>/config/service-function-scheduler-type:service-function-scheduler-types/service-function-scheduler-type/`

- Service Function Forwarder (SFF): `http://<host>:<port>/config/service-function-forwarder:service-function-forwarders/service-function-forwarder/`
- Rendered Service Path (RSP): `http://<host>:<port>/operational/rendered-service-path:rendered-service-paths/rendered-service-path/`

Therefore, network devices willing to receive REST messages must listen on these REST URLs.

Note: Service Classifier Function (SCF) URL does not exist, because SCF is considered as one of the network devices willing to receive REST messages. However, there is a listener hooked on the SCF data store, which is triggering POST/PUT/DELETE operations of ACL object, because ACL is referenced in `service-function-classifier.yang`

1.5 Service Function Load Balancing Developer Guide

1.5.1 Overview

SFC Load-Balancing feature implements load balancing of Service Functions, rather than a one-to-one mapping between Service Function Forwarder and Service Function.

1.5.2 Load Balancing Architecture

Service Function Groups (SFG) can replace Service Functions (SF) in the Rendered Path model. A Service Path can only be defined using SFGs or SFs, but not a combination of both.

Relevant objects in the YANG model are as follows:

1. Service-Function-Group-Algorithm:

```
Service-Function-Group-Algorithms {
    Service-Function-Group-Algorithm {
        String name
        String type
    }
}
```

```
Available types: ALL, SELECT, INDIRECT, FAST_FAILURE
```

2. Service-Function-Group:

```
Service-Function-Groups {
    Service-Function-Group {
        String name
        String serviceFunctionGroupAlgorithmName
        String type
        String groupId
        Service-Function-Group-Element {
            String service-function-name
            int index
        }
    }
}
```

3. ServiceFunctionHop: holds a reference to a name of SFG (or SF)

1.5.3 Key APIs and Interfaces

This feature enhances the existing SFC API.

REST API commands include: * For Service Function Group (SFG): read existing SFG, write new SFG, delete existing SFG, add Service Function (SF) to SFG, and delete SF from SFG * For Service Function Group Algorithm (SFG-Alg): read, write, delete

Bundle providing the REST API: sfc-sb-rest * Service Function Groups and Algorithms are defined in: sfc-sfg and sfc-sfg-alg * Relevant JAVA API: SfcProviderServiceFunctionGroupAPI, SfcProviderServiceFunctionGroupAlgAPI

1.6 Service Function Scheduling Algorithms

1.6.1 Overview

When creating the Rendered Service Path (RSP), the earlier release of SFC chose the first available service function from a list of service function names. Now a new API is introduced to allow developers to develop their own schedule algorithms when creating the RSP. There are four scheduling algorithms (Random, Round Robin, Load Balance and Shortest Path) are provided as examples for the API definition. This guide gives a simple introduction of how to develop service function scheduling algorithms based on the current extensible framework.

1.6.2 Architecture

The following figure illustrates the service function selection framework and algorithms.

The YANG Model defines the Service Function Scheduling Algorithm type identities and how they are stored in the MD-SAL data store for the scheduling algorithms.

The MD-SAL data store stores all informations for the scheduling algorithms, including their types, names, and status.

The API provides some basic APIs to manage the informations stored in the MD-SAL data store, like putting new items into it, getting all scheduling algorithms, etc.

The RESTCONF API provides APIs to manage the informations stored in the MD-SAL data store through RESTful calls.

The Service Function Chain Renderer gets the enabled scheduling algorithm type, and schedules the service functions with scheduling algorithm implementation.

1.6.3 Key APIs and Interfaces

While developing a new Service Function Scheduling Algorithm, a new class should be added and it should extend the base schedule class SfcServiceFunctionSchedulerAPI. And the new class should implement the abstract function:

```
public List<String> scheduleServiceFuntions(ServiceFunctionChain chain, int
serviceIndex).
```

- ``**ServiceFunctionChain chain**``: the chain which will be rendered
- ``**int serviceIndex**``: the initial service index for this rendered service path
- ``**List<String>**``: a list of service function names which scheduled by the Service Function Scheduling Algorithm.

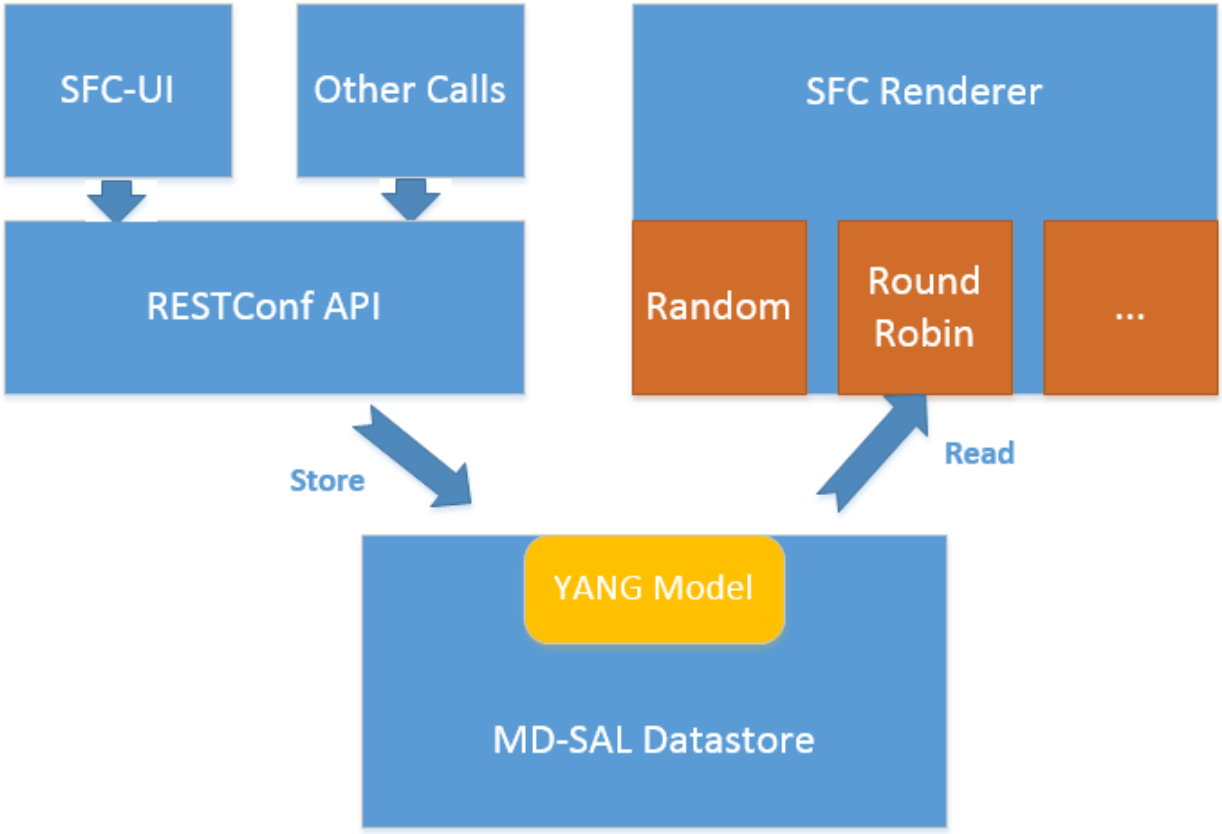


Fig. 3: SF Scheduling Algorithm framework Architecture

1.6.4 API Reference Documentation

Please refer the API docs generated in the mdsal-apidocs.

1.7 SFC Proof of Transit Developer Guide

1.7.1 Overview

SFC Proof of Transit implements the in-situ OAM (iOAM) Proof of Transit verification for SFCs and other paths. The implementation is broadly divided into the North-bound (NB) and the South-bound (SB) side of the application. The NB side is primarily charged with augmenting the RSP with user-inputs for enabling the PoT on the RSP, while the SB side is dedicated to auto-generated SFC PoT parameters, periodic refresh of these parameters and delivering the parameters to the NETCONF and iOAM capable nodes (eg. VPP instances).

1.7.2 Architecture

The following diagram gives the high level overview of the different parts.

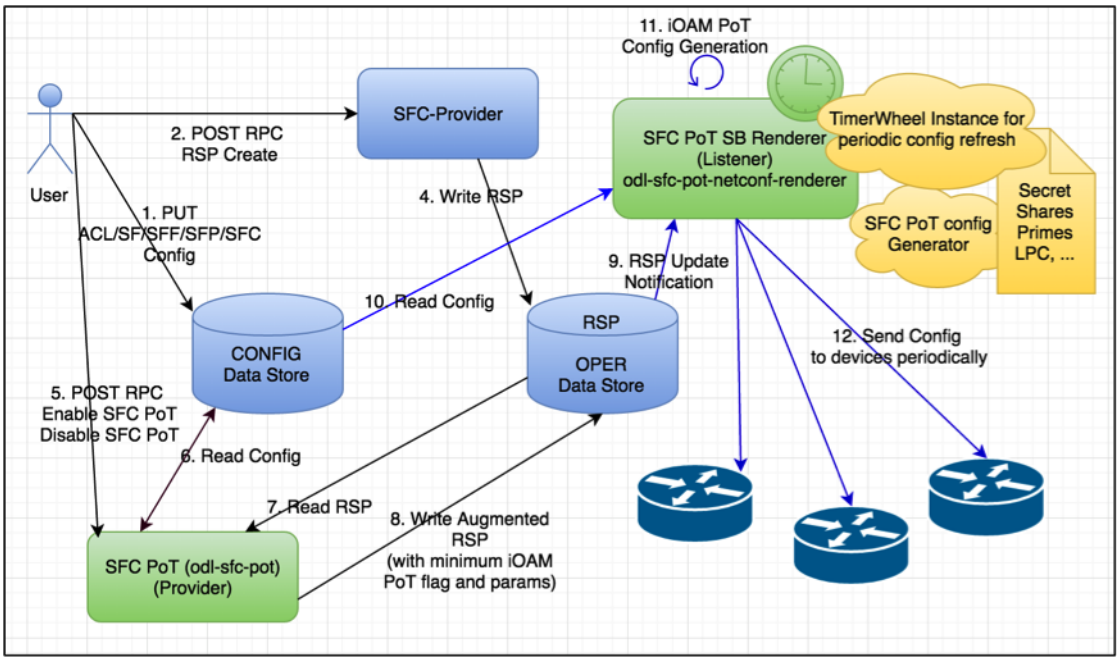


Fig. 4: SFC Proof of Transit Internal Architecture

The Proof of Transit feature is enabled by two sub-features:

1. ODL SFC PoT: `feature:install odl-sfc-pot`
2. ODL SFC PoT NETCONF Renderer: `feature:install odl-sfc-pot-netconf-renderer`

1.7.3 Details

The following classes and handlers are involved.

1. The class (SfcPotRpc) sets up RPC handlers for enabling the feature.
2. There are new RPC handlers for two new RPCs (EnableSfcIoamPotRenderedPath and DisableSfcIoamPotRenderedPath) and effected via SfcPotRspProcessor class.
3. When a user configures via a POST RPC call to enable Proof of Transit on a particular SFC (via the Rendered Service Path), the configuration drives the creation of necessary augmentations to the RSP (to modify the RSP) to effect the Proof of Transit configurations.
4. The augmentation meta-data added to the RSP are defined in the sfc-ioam-nb-pot.yang file.

   Note: There are no auto generated configuration parameters added to the RSP to avoid RSP bloat.

5. Adding SFC Proof of Transit meta-data to the RSP is done in the SfcPotRspProcessor class.
6. Once the RSP is updated, the RSP data listeners in the SB renderer modules (odl-sfc-pot-netconf-renderer) will listen to the RSP changes and send out configurations to the necessary network nodes that are part of the SFC.
7. The configurations are handled mainly in the SfcPotAPI, SfcPotConfigGenerator, SfcPotPolyAPI, SfcPotPolyClass and SfcPotPolyClassAPI classes.
8. There is a sfc-ioam-sb-pot.yang file that shows the format of the iOAM PoT configuration data sent to each node of the SFC.
9. A timer is started based on the "ioam-pot-refresh-period" value in the SB renderer module that handles configuration refresh periodically.
10. The SB and timer handling are done in the odl-sfc-pot-netconf-renderer module. Note: This is NOT done in the NB odl-sfc-pot module to avoid periodic updates to the RSP itself.
11. ODL creates a new profile of a set of keys and secrets at a constant rate and updates an internal data store with the configuration. The controller labels the configurations per RSP as "even" or "odd" – and the controller cycles between "even" and "odd" labeled profiles. The rate at which these profiles are communicated to the nodes is configurable and in future, could be automatic based on profile usage. Once the profile has been successfully communicated to all nodes (all Netconf transactions completed), the controller sends an "enable pot-profile" request to the ingress node.
12. The nodes are to maintain two profiles (an even and an odd pot-profile). One profile is currently active and in use, and one profile is about to get used. A flag in the packet is indicating whether the odd or even pot-profile is to be used by a node. This is to ensure that during profile change we're not disrupting the service. I.e. if the "odd" profile is active, the controller can communicate the "even" profile to all nodes and only if all the nodes have received it, the controller will tell the ingress node to switch to the "even" profile. Given that the indicator travels within the packet, all nodes will switch to the "even" profile. The "even" profile gets active on all nodes – and nodes are ready to receive a new "odd" profile.
13. HashedTimerWheel implementation is used to support the periodic configuration refresh. The default refresh is 5 seconds to start with.
14. Depending on the last updated profile, the odd or the even profile is updated in the fresh timer pop and the configurations are sent down appropriately.
15. SfcPotTimerQueue, SfcPotTimerWheel, SfcPotTimerTask, SfcPotTimerData and SfcPotTimerThread are the classes that handle the Proof of Transit protocol profile refresh implementation.

16. The RSP data store is NOT being changed periodically and the timer and configuration refresh modules are present in the SB renderer module handler and hence there are are no scale or RSP churn issues affecting the design.

The following diagram gives the overall sequence diagram of the interactions between the different classes.

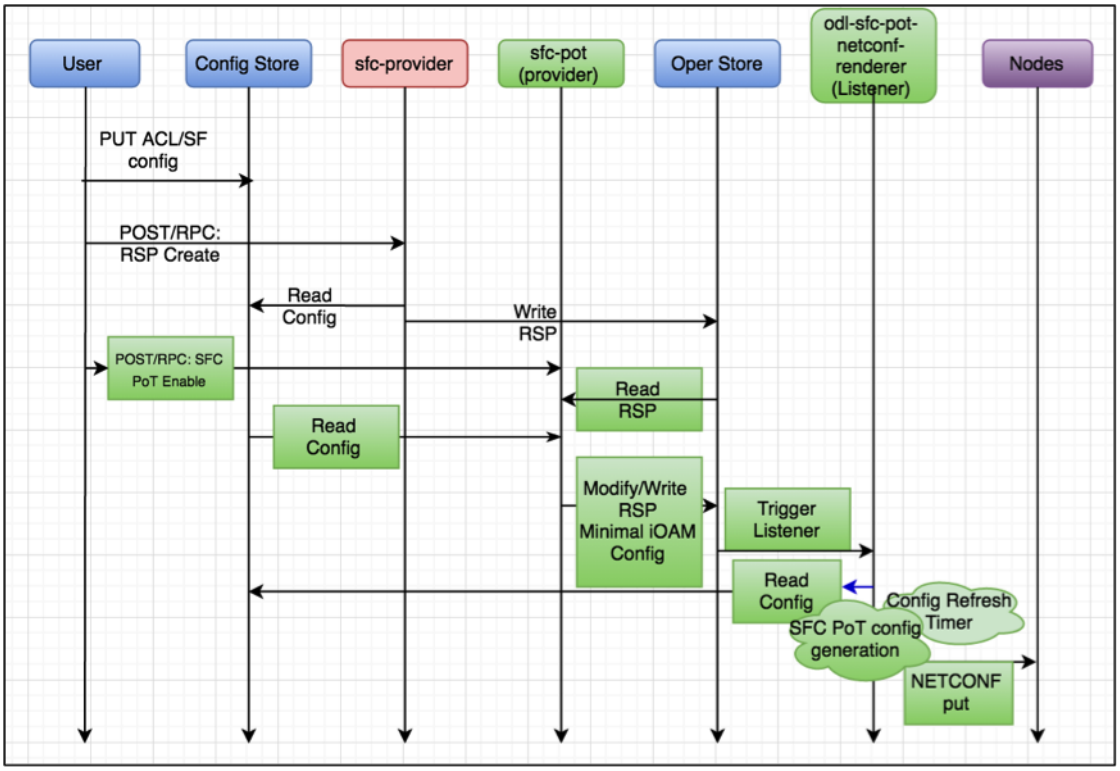


Fig. 5: SFC Proof of Transit Sequence Diagram

1.8 Logical Service Function Forwarder

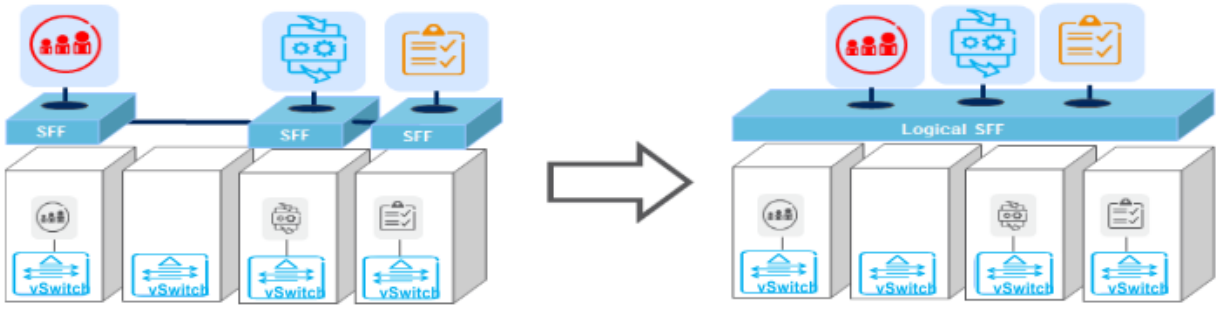
1.8.1 Overview

Rationale

When the current SFC is deployed in a cloud environment, it is assumed that each switch connected to a Service Function is configured as a Service Function Forwarder and each Service Function is connected to its Service Function Forwarder depending on the Compute Node where the Virtual Machine is located. This solution allows the basic cloud use cases to be fulfilled, as for example, the ones required in OPNFV Brahmaputra, however, some advanced use cases, like the transparent migration of VMs can not be implemented. The Logical Service Function Forwarder enables the following advanced use cases:

1. Service Function mobility without service disruption
2. Service Functions load balancing and failover

As shown in the picture below, the Logical Service Function Forwarder concept extends the current SFC northbound API to provide an abstraction of the underlying Data Center infrastructure. The Data Center underlaying network can be abstracted by a single SFF. This single SFF uses the logical port UUID as data plane locator to connect SFs globally and in a location-transparent manner. SFC makes use of Genius project to track the location of the SF's logical ports.



The SFC internally distributes the necessary flow state over the relevant switches based on the internal Data Center topology and the deployment of SFs.

1.8.2 Changes in data model

The Logical Service Function Forwarder concept extends the current SFC northbound API to provide an abstraction of the underlying Data Center infrastructure.

The Logical SFF simplifies the configuration of the current SFC data model by reducing the number of parameters to be be configured in every SFF, since the controller will discover those parameters by interacting with the services offered by the Genius project.

The following picture shows the Logical SFF data model. The model gets simplified as most of the configuration parameters of the current SFC data model are discovered in runtime. The complete YANG model can be found here logical SFF model.

There are other minor changes in the data model; the SFC encapsulation type has been added or moved in the following files:

- RSP data model
- SFP data model
- Service Locator data model

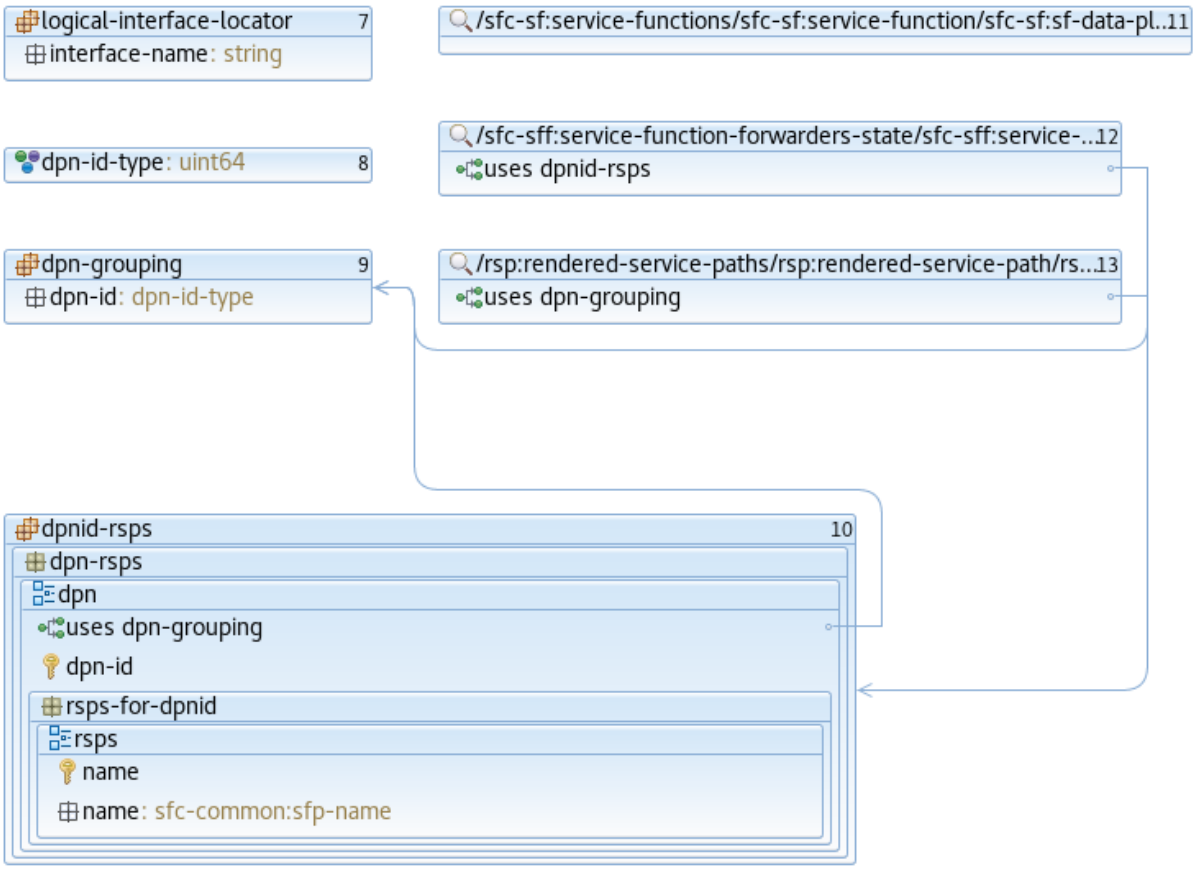
1.8.3 Interaction with Genius

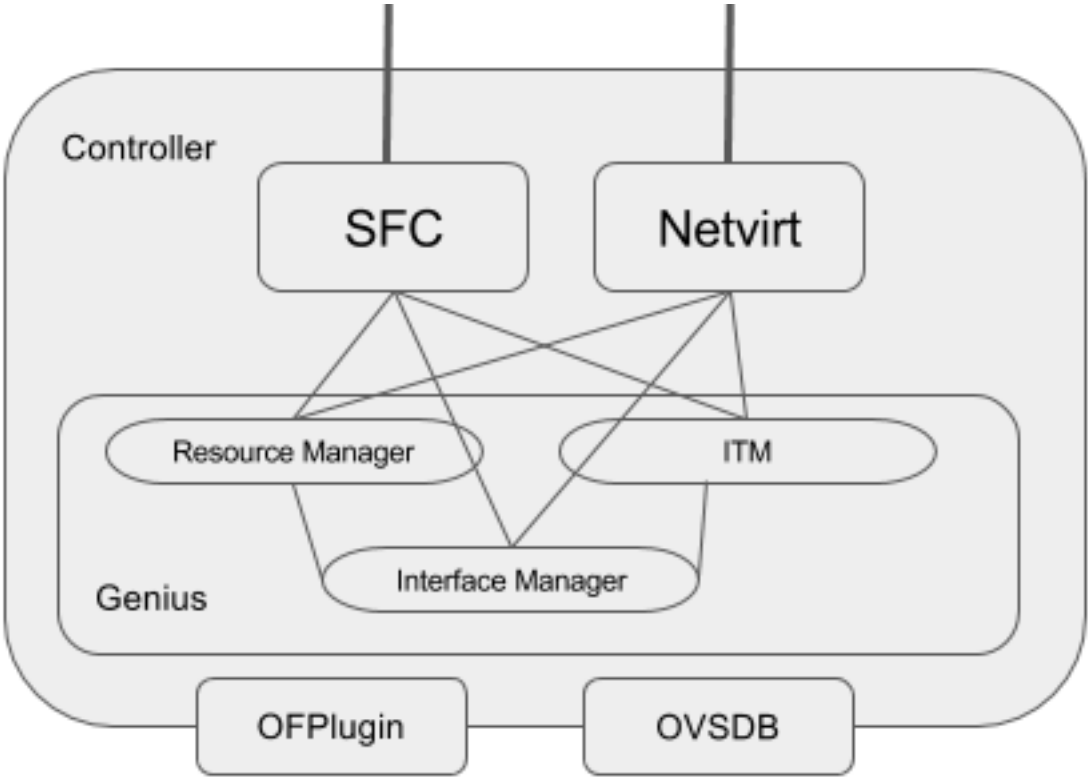
Feature *sfc-genius* functionally enables SFC integration with Genius. This allows configuring a Logical SFF and SFs attached to this Logical SFF via logical interfaces (i.e. neutron ports) that are registered with Genius.

As shown in the following picture, SFC will interact with Genius project's services to provide the Logical SFF functionality.

The following are the main Genius' services used by SFC:

1. Interaction with Interface Tunnel Manager (ITM)
2. Interaction with the Interface Manager
3. Interaction with Resource Manager


logical-interface-locator 7
interface-name: string
/sfc-sf:service-functions/sfc-sf:service-function/sfc-sf:sf-data-pl..11
dpn-id-type: uint64 8
/sfc-sff:service-function-forwarders-state/sfc-sff:service-...12
uses dpnid-rsps
dpn-grouping 9
dpn-id: dpn-id-type
/rsp:rendered-service-paths/rsp:rendered-service-path/rs...13
uses dpn-grouping
dpnid-rsps 10
dpn-rsps
dpn
uses dpn-grouping
dpn-id
rsps-for-dpnid
rsps
name
name: sfc-common:sfp-name



SFC Service registration with Genius

Genius handles the coexistence of different network services. As such, SFC service is registered with Genius performing the following actions:

SFC Service Binding As soon as a Service Function associated to the Logical SFF is involved in a Rendered Service Path, SFC service is bound to its logical interface via Genius Interface Manager. This has the effect of forwarding every incoming packet from the Service Function to the SFC pipeline of the attached switch, as long as it is not consumed by a different bound service with higher priority.

SFC Service Terminating Action As soon as SFC service is bound to the interface of a Service Function for the first time on a specific switch, a terminating service action is configured on that switch via Genius Interface Tunnel Manager. This has the effect of forwarding every incoming packet from a different switch to the SFC pipeline as long as the traffic is VXLAN encapsulated on VNI 0.

The following sequence diagrams depict how the overall process takes place:

For more information on how Genius allows different services to coexist, see the Genius User Guide.

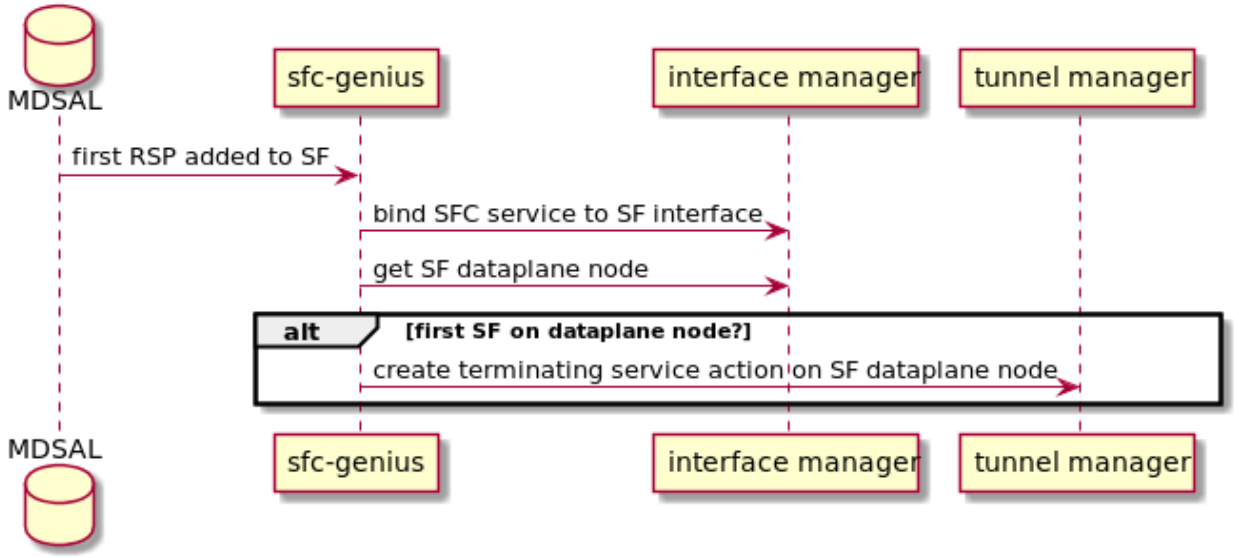


Fig. 6: SFC genius module interaction with Genius at RSP creation.

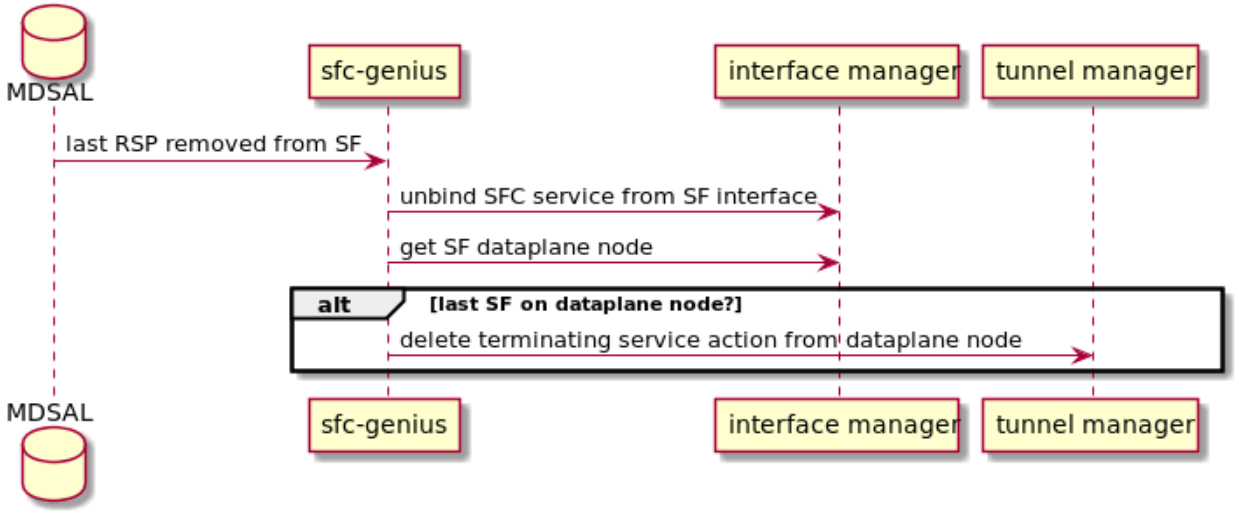


Fig. 7: SFC genius module interaction with Genius at RSP removal.

Path Rendering

During path rendering, Genius is queried to obtain needed information, such as:

- Location of a logical interface on the data-plane.
- Tunnel interface for a specific pair of source and destination switches.
- Egress OpenFlow actions to output packets to a specific interface.

See *RSP Rendering* section for more information.

VM migration

Upon VM migration, it's logical interface is first unregistered and then registered with Genius, possibly at a new physical location. *sfc-genius* reacts to this by re-rendering all the RSPs on which the associated SF participates, if any.

The following picture illustrates the process:

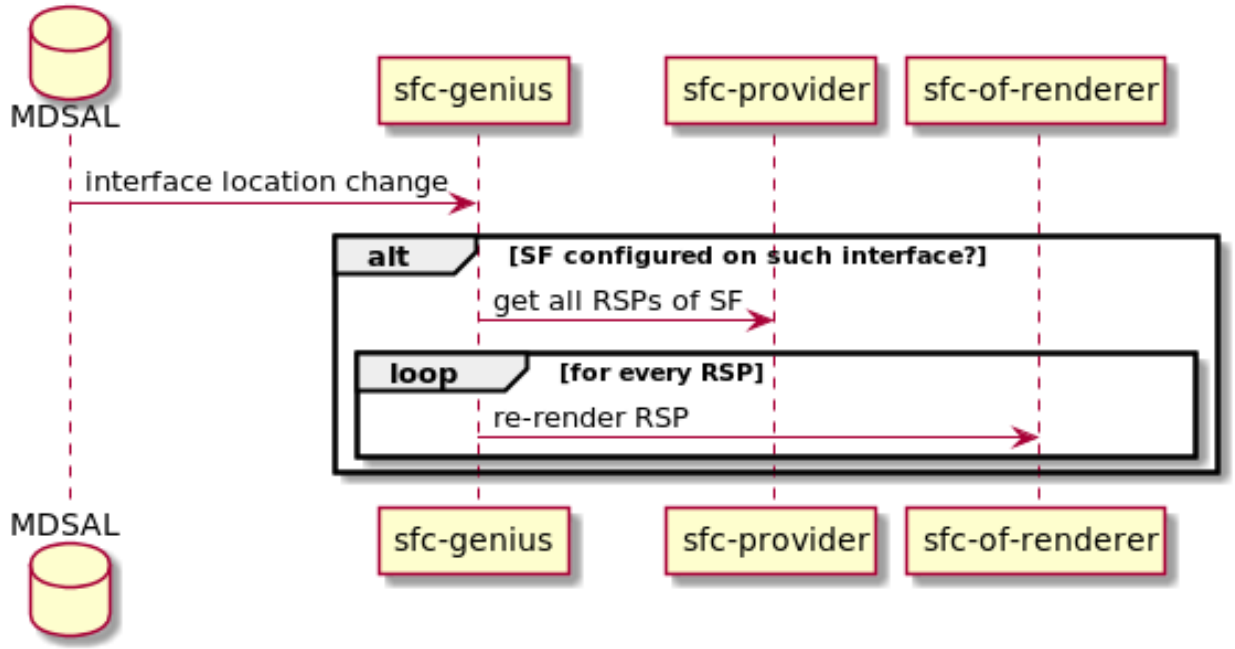


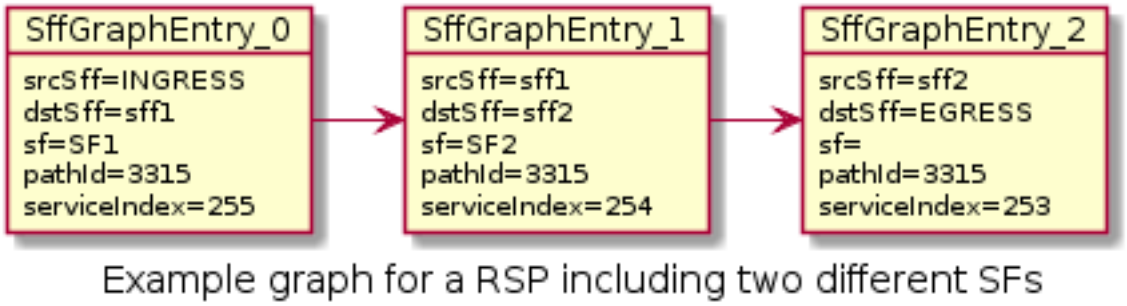
Fig. 8: SFC genius module at VM migration.

1.8.4 RSP Rendering changes for paths using the Logical SFF

1. **Construction of the auxiliary rendering graph**

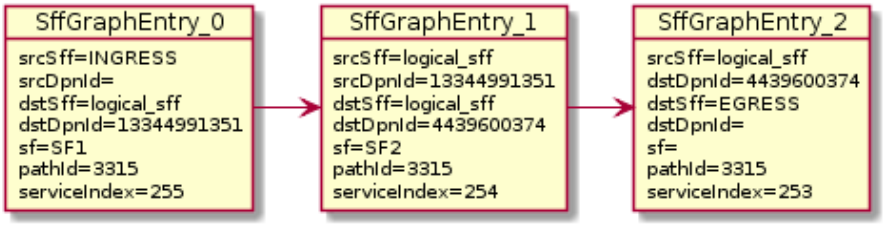
   When starting the rendering of a RSP, the SFC renderer builds an auxiliary graph with information about the required hops for traffic traversing the path. RSP processing is achieved by iteratively evaluating each of the entries in the graph, writing the required flows in the proper switch for each hop.

   It is important to note that the graph includes both traffic ingress (i.e. traffic entering into the first SF) and traffic egress (i.e. traffic leaving the chain from the last SF) as hops. Therefore, the number of entries in the graph equals the number of SFs in the chain plus one.



Example graph for a RSP including two different SFs

The process of rendering a chain when the switches involved are part of the Logical SFF also starts with the construction of the hop graph. The difference is that when the SFs used in the chain are using a logical interface, the SFC renderer will also retrieve from Genius the DPIDs for the switches, storing them in the graph. In this context, those switches are the ones in the compute nodes each SF is hosted on at the time the chain is rendered.



Example graph for a RSP including two different SFs that are configured using a Logical SFF.
It can be observed that the service functions SF1, SF2 are hosted in different compute nodes

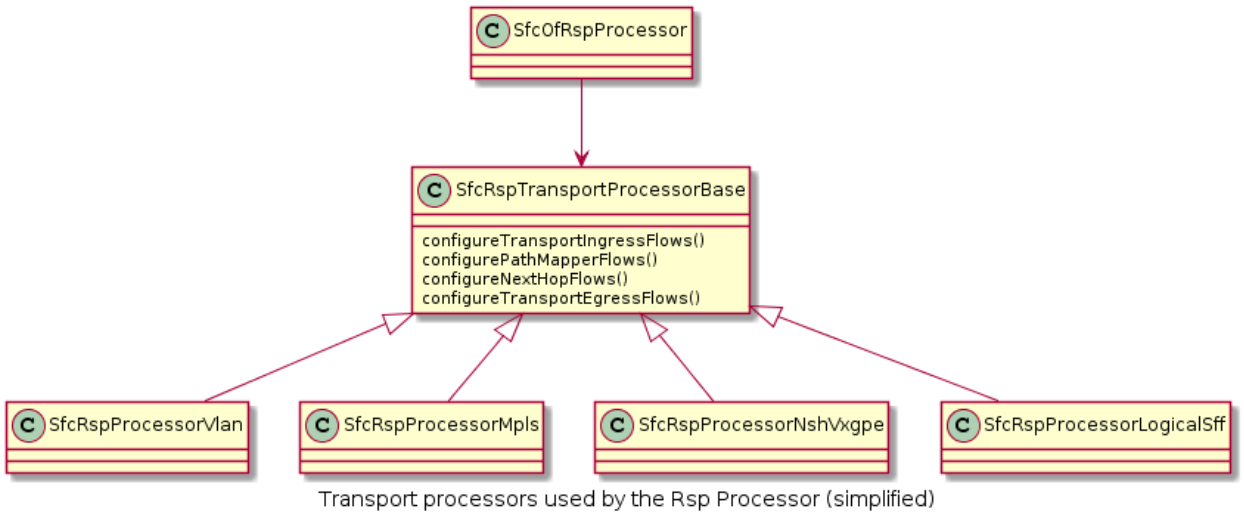
2. **New transport processor**

   Transport processors are classes which calculate and write the correct flows for a chain. Each transport processor specializes on writing the flows for a given combination of transport type and SFC encapsulation.

   A specific transport processor has been created for paths using a Logical SFF. A particularity of this transport processor is that its use is not only determined by the transport / SFC encapsulation combination, but also because the chain is using a Logical SFF. The actual condition evaluated for selecting the Logical SFF transport processor is that the SFs in the chain are using logical interface locators, and that the DPIDs for those locators can be successfully retrieved from Genius.

   The main differences between the Logical SFF transport processor and other processors are the following:

   - Instead of srcSff, dstSff fields in the hops graph (which are all equal in a path using a Logical SFF), the Logical SFF transport processor uses previously stored srcDpnId, dstDpnId fields in order to know whether an actual hop between compute nodes must be performed or not (it is possible that two consecutive SFs are collocated in the same compute node).
   - When a hop between switches really has to be performed, it relies on Genius for getting the actions to perform that hop. The retrieval of those actions involve two steps:
     - First, Genius' Overlay Tunnel Manager module is used in order to retrieve the target interface for a jump between the source and the destination DPIDs.
     - Then, egress instructions for that interface are retrieved from Genius's Interface Manager.
   - There are no next hop rules between compute nodes, only egress instructions (the transport zone tunnels have all the required routing information).
   - Next hop information towards SFs uses mac adresses which are also retrieved from the Genius datastore.



- The Logical SFF transport processor performs NSH decapsulation in the last switch of the chain.

3. **Post-rendering update of the operational data model**

   When the rendering of a chain finishes successfully, the Logical SFF Transport Processor perform two operational datastore modifications in order to provide some relevant runtime information about the chain. The exposed information is the following:

   - Rendered Service Path state: when the chain uses a Logical SFF, DPIDs for the switches in the compute nodes on which the SFs participating in the chain are hosted are added to the hop information.
   - SFF state: A new list of all RSPs which use each DPID is has been added. It is updated on each RSP addition / deletion.

1.8.5 Classifier impacts

This section explains the changes made to the SFC classifier, enabling it to be attached to Logical SFFs.

Refer to the following image to better understand the concept, and the required steps to implement the feature.

As stated in the *SFC User Guide*, the classifier needs to be provisioned using logical interfaces as attachment points.

When that happens, MDSAL will trigger an event in the odl-sfc-scf-openflow feature (i.e. the sfc-classifier), which is responsible for installing the classifier flows in the classifier switches.

The first step of the process, is to bind the interfaces to classify in Genius, in order for the desired traffic (originating from the VMs having the provisioned attachment-points) to enter the SFC pipeline. This will make traffic reach table 82 (SFC classifier table), coming from table 0 (table managed by Genius, shared by all applications).

The next step, is deciding which flows to install in the SFC classifier table. A table-miss flow will be installed, having a MatchAny clause, whose action is to jump to Genius's egress dispatcher table. This enables traffic intended for other applications to still be processed.

The flow that allows the SFC pipeline to continue is added next, having higher match priority than the table-miss flow. This flow has two responsabilities:

1. **Push the NSH header, along with its metadata (required within the SFC pipeline)**

   Features the specified ACL matches as match criteria, and push NSH along with its metadata into the Action list.

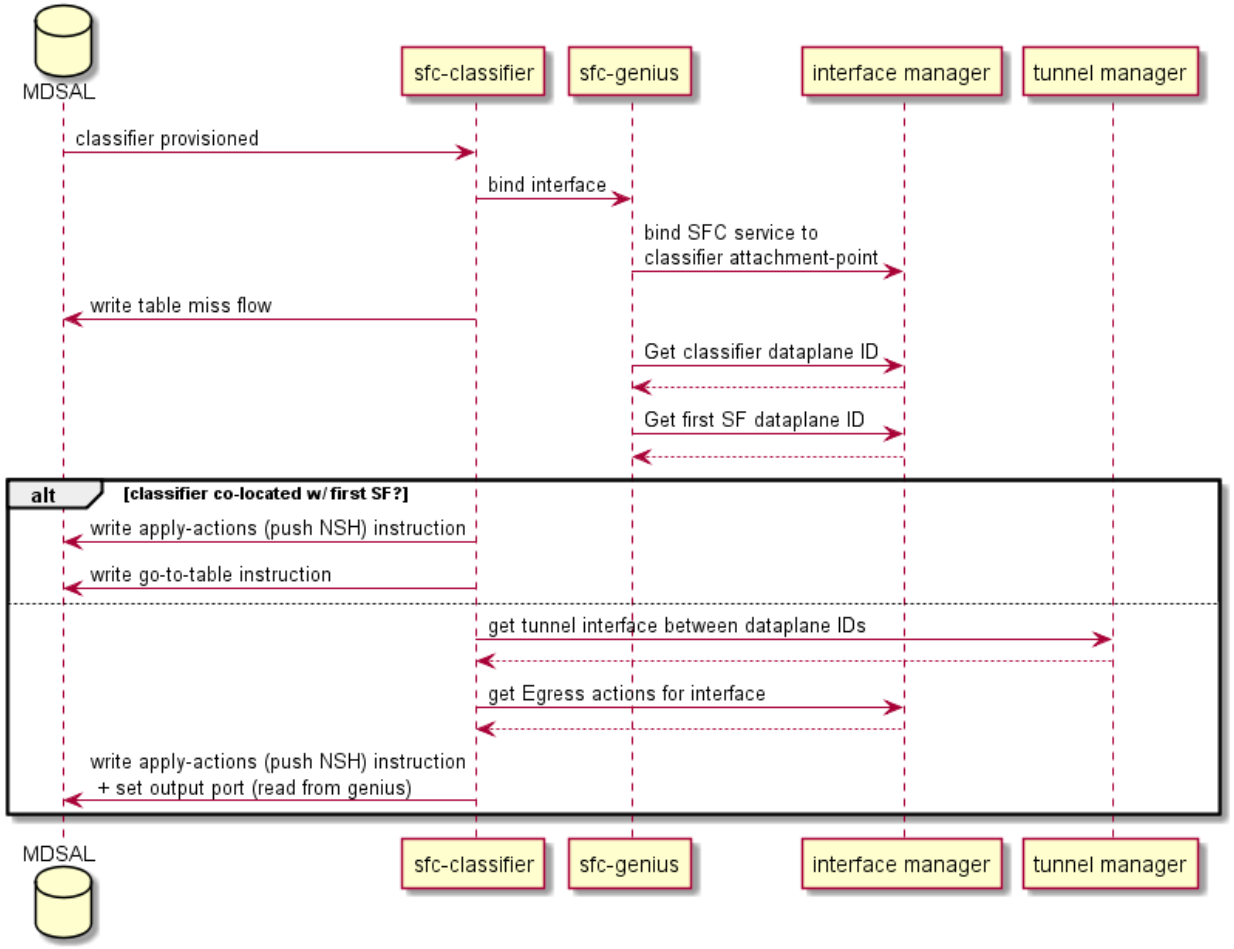


Fig. 9: SFC classifier integration with Genius.

2. **Advance the SFC pipeline**

   Forward the traffic to the first Service Function in the RSP. This steers packets into the SFC domain, and how it is done depends on whether the classifier is co-located with the first service function in the specified RSP.

   Should the classifier be co-located (i.e. in the same compute node), a new instruction is appended to the flow, telling all matches to jump to the transport ingress table.

   If not, Genius's tunnel manager service is queried to get the tunnel interface connecting the classifier node with the compute node where the first Service Function is located, and finally, Genius's interface manager service is queried asking for instructions on how to reach that tunnel interface.

   These actions are then appended to the Action list already containing push NSH and push NSH metadata Actions, and written in an Apply-Actions Instruction into the datastore.

CHAPTER

TWO

SERVICE FUNCTION CHAINING USER GUIDE

2.1 OpenDaylight Service Function Chaining (SFC) Overview

OpenDaylight Service Function Chaining (SFC) provides the ability to define an ordered list of network services (e.g. firewalls, load balancers). These services are then "stitched" together in the network to create a service chain. This project provides the infrastructure (chaining logic, APIs) needed for ODL to provision a service chain in the network and an end-user application for defining such chains.

- ACE - Access Control Entry
- ACL - Access Control List
- SCF - Service Classifier Function
- SF - Service Function
- SFC - Service Function Chain
- SFF - Service Function Forwarder
- SFG - Service Function Group
- SFP - Service Function Path
- RSP - Rendered Service Path
- NSH - Network Service Header

2.2 SFC User Interface

2.2.1 Overview

The SFC User interface comes with a Command Line Interface (CLI): it provides several Karaf console commands to show the SFC model (SF, SFFs, etc.) provisioned in the datastore.

2.2.2 SFC Web Interface (SFC-UI)

Architecture

SFC-UI operates purely by using RESTCONF.

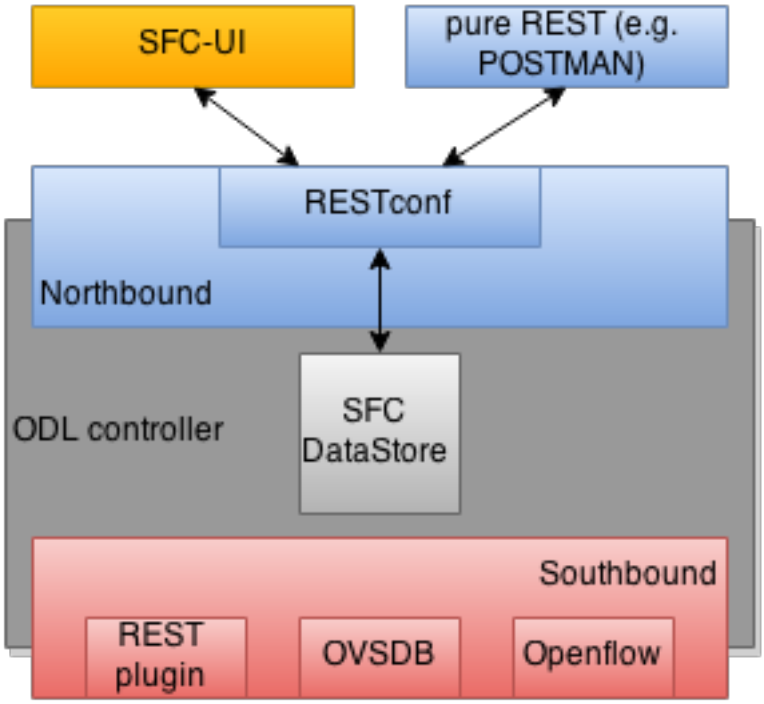


Fig. 1: SFC-UI integration into ODL

How to access

1. Run ODL distribution (run karaf)
2. In Karaf console execute: `feature:install odl-sfc-ui`
3. Visit SFC-UI on: `http://<odl_ip_address>:8181/sfc/index.html`

2.2.3 SFC Command Line Interface (SFC-CLI)

Overview

The Karaf Container offers a complete Unix-like console that allows managing the container. This console can be extended with custom commands to manage the features deployed on it. This feature will add some basic commands to show the provisioned SFC entities.

How to use it

The SFC-CLI implements commands to show some of the provisioned SFC entities: Service Functions, Service Function Forwarders, Service Function Chains, Service Function Paths, Service Function Classifiers, Service Nodes and Service Function Types:

- List one/all provisioned Service Functions:

```
sfc:sf-list [--name <name>]
```

- List one/all provisioned Service Function Forwarders:

```
sfc:sff-list [--name <name>]
```

- List one/all provisioned Service Function Chains:

```
sfc:sfc-list [--name <name>]
```

- List one/all provisioned Service Function Paths:

```
sfc:sfp-list [--name <name>]
```

- List one/all provisioned Service Function Classifiers:

```
sfc:sc-list [--name <name>]
```

- List one/all provisioned Service Nodes:

```
sfc:sn-list [--name <name>]
```

- List one/all provisioned Service Function Types:

```
sfc:sft-list [--name <name>]
```

2.3 SFC Southbound REST Plug-in

2.3.1 Overview

The Southbound REST Plug-in is used to send configuration from datastore down to network devices supporting a REST API (i.e. they have a configured REST URI). It supports POST/PUT/DELETE operations, which are triggered accordingly by changes in the SFC data stores.

- Access Control List (ACL)
- Service Classifier Function (SCF)
- Service Function (SF)
- Service Function Group (SFG)
- Service Function Schedule Type (SFST)
- Service Function Forwarder (SFF)
- Rendered Service Path (RSP)

2.3.2 Southbound REST Plug-in Architecture

From the user perspective, the REST plug-in is another SFC Southbound plug-in used to communicate with network devices.

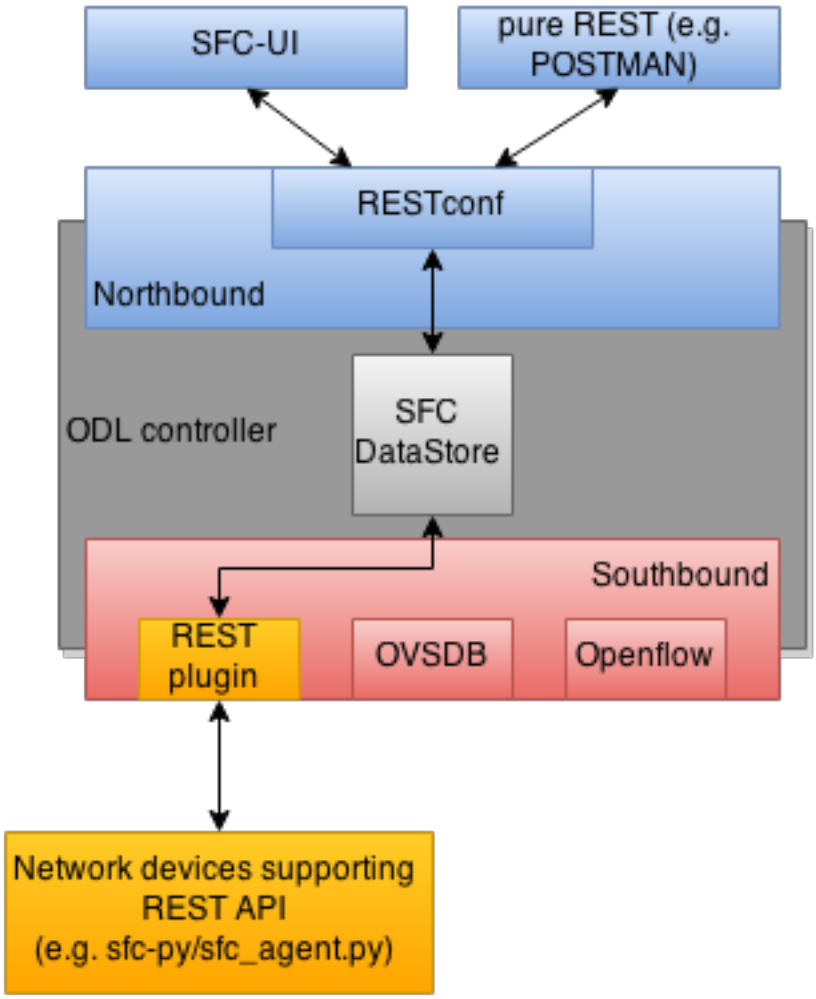


Fig. 2: Southbound REST Plug-in integration into ODL

2.3.3 Configuring Southbound REST Plugin

1. Run ODL distribution (run karaf)
2. In Karaf console execute: `feature:install odl-sfc-sb-rest`
3. Configure REST URIs for SF/SFF through SFC User Interface or RESTCONF (required configuration steps can be found in the tutorial stated bellow)

2.3.4 Tutorial

Comprehensive tutorial on how to use the Southbound REST Plug-in and how to control network devices with it can be found on: https://wiki.opendaylight.org/view/Service_Function_Chaining:Main#SFC_103

2.4 SFC-OVS integration

2.4.1 Overview

SFC-OVS provides integration of SFC with Open vSwitch (OVS) devices. Integration is realized through mapping of SFC objects (like SF, SFF, Classifier, etc.) to OVS objects (like Bridge, TerminationPoint=Port/Interface). The mapping takes care of automatic instantiation (setup) of corresponding object whenever its counterpart is created. For example, when a new SFF is created, the SFC-OVS plug-in will create a new OVS bridge.

The feature is intended for SFC users willing to use Open vSwitch as an underlying network infrastructure for deploying RSPs (Rendered Service Paths).

2.4.2 SFC-OVS Architecture

SFC-OVS uses the OVSDB MD-SAL Southbound API for getting/writing information from/to OVS devices. From the user perspective SFC-OVS acts as a layer between SFC datastore and OVSDB.

2.4.3 Configuring SFC-OVS

1. Run ODL distribution (run karaf)
2. In Karaf console execute: `feature:install odl-sfc-ovs`
3. Configure Open vSwitch to use ODL as a manager, using following command: `ovs-vsctl set-manager tcp:<odl_ip_address>:6640`

2.4.4 Tutorials

Verifying mapping from SFF to OVS

Overview

This tutorial shows the usual workflow during creation of an OVS Bridge with use of the SFC APIs.

Prerequisites

- Open vSwitch installed (ovs-vsctl command available in shell)
- SFC-OVS feature configured as stated above

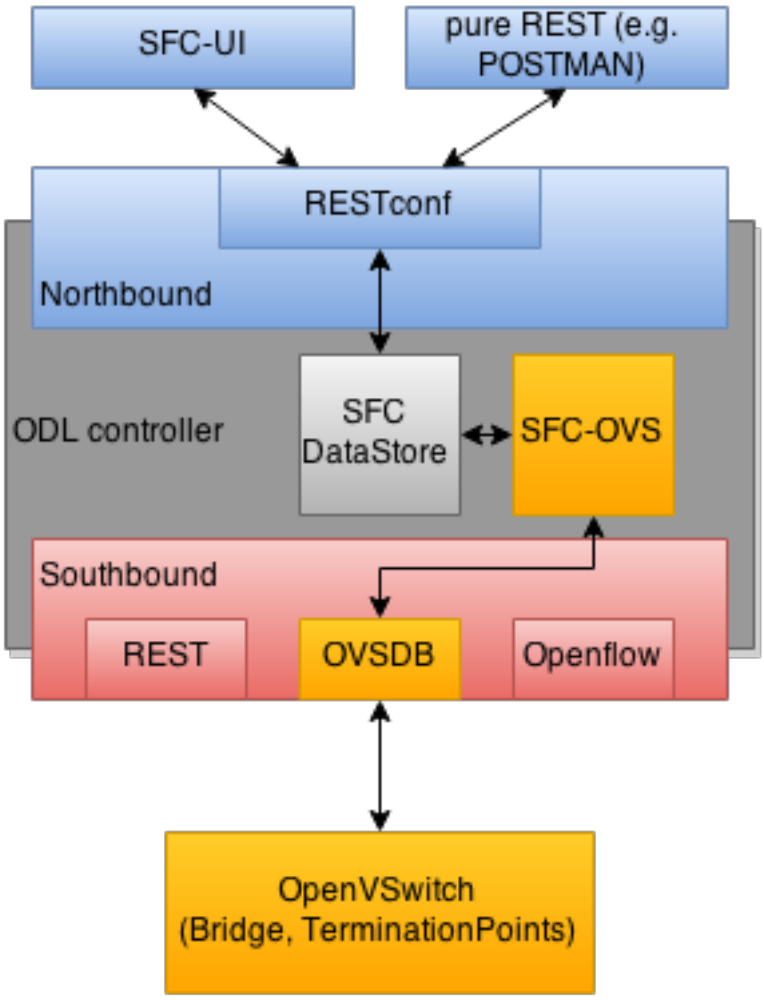


Fig. 3: SFC-OVS integration into ODL

Instructions

1. In a shell execute: `ovs-vsctl set-manager tcp:<odl_ip_address>:6640`
2. Send POST request to URL: `http://<odl_ip_address>:8181/restconf/operations/service-function-forwarder-ovs:create-ovs-bridge` Use Basic auth with credentials: "admin", "admin" and set `Content-Type: application/json`. The content of POST request should be following:

```
{
    "input":
    {
        "name": "br-test",
        "ovs-node": {
            "ip": "<Open_vSwitch_ip_address>"
        }
    }
}
```

Open_vSwitch_ip_address is the IP address of the machine where Open vSwitch is installed.

Verification

In a shell execute: `ovs-vsctl show`. There should be a Bridge with the name *br-test* and one port/interface called *br-test*.

Also, the corresponding SFF for this OVS Bridge should be configured, which can be verified through the SFC User Interface or RESTCONF as follows.

a. Visit the SFC User Interface: `http://<odl_ip_address>:8181/sfc/index.html#/sfc/serviceforwarder`
b. Use pure RESTCONF and send a GET request to URL: `http://<odl_ip_address>:8181/restconf/config/service-function-forwarder:service-function-forwarders`

There should be an SFF, whose name will be ending with *br1* and the SFF should contain two DataPlane locators: *br1* and *testPort*.

2.5 SFC Classifier User Guide

2.5.1 Overview

Description of classifier can be found in: https://datatracker.ietf.org/doc/draft-ietf-sfc-architecture/

There are two types of classifier:

1. OpenFlow Classifier
2. Iptables Classifier

2.5.2 OpenFlow Classifier

OpenFlow Classifier implements the classification criteria based on OpenFlow rules deployed into an OpenFlow switch. An Open vSwitch will take the role of a classifier and performs various encapsulations such NSH, VLAN, MPLS, etc. In the existing implementation, classifier can support NSH encapsulation. Matching information is based on ACL for MAC addresses, ports, protocol, IPv4 and IPv6. Supported protocols are TCP, UDP and SCTP. Actions information in the OF rules, shall be forwarding of the encapsulated packets with specific information related to the RSP.

Classifier Architecture

The OVSDB Southbound interface is used to create an instance of a bridge in a specific location (via IP address). This bridge contains the OpenFlow rules that perform the classification of the packets and react accordingly. The OpenFlow Southbound interface is used to translate the ACL information into OF rules within the Open vSwitch.

Note: in order to create the instance of the bridge that takes the role of a classifier, an "empty" SFF must be created.

Configuring Classifier

1. An empty SFF must be created in order to host the ACL that contains the classification information.
2. SFF data plane locator must be configured
3. Classifier interface must be manually added to SFF bridge.

Administering or Managing Classifier

Classification information is based on MAC addresses, protocol, ports and IP. ACL gathers this information and is assigned to an RSP which turns to be a specific path for a Service Chain.

2.5.3 Iptables Classifier

Classifier manages everything from starting the packet listener to creation (and removal) of appropriate ip(6)tables rules and marking received packets accordingly. Its functionality is **available only on Linux** as it leverdges **NetfilterQueue**, which provides access to packets matched by an **iptables** rule. Classifier requires **root privileges** to be able to operate.

So far it is capable of processing ACL for MAC addresses, ports, IPv4 and IPv6. Supported protocols are TCP and UDP.

Classifier Architecture

Python code located in the project repository sfc-py/common/classifier.py.

Note: classifier assumes that Rendered Service Path (RSP) **already exists** in ODL when an ACL referencing it is obtained

---

1. sfc_agent receives an ACL and passes it for processing to the classifier
2. the RSP (its SFF locator) referenced by ACL is requested from ODL

3. if the RSP exists in the ODL then ACL based iptables rules for it are applied

After this process is over, every packet successfully matched to an iptables rule (i.e. successfully classified) will be NSH encapsulated and forwarded to a related SFF, which knows how to traverse the RSP.

Rules are created using appropriate iptables command. If the Access Control Entry (ACE) rule is MAC address related both iptables and IPv6 tables rules re issued. If ACE rule is IPv4 address related, only iptables rules are issued, same for IPv6.

Note: iptables **raw** table contains all created rules

Configuring Classifier

Classfier does't need any configuration.
Its only requirement is that the **second (2) Netfilter Queue** is not used by any other process and is **avalilable for the classifier**.

Administering or Managing Classifier

Classifier runs alongside sfc_agent, therefore the command for starting it locally is:

```
sudo python3.4 sfc-py/sfc_agent.py --rest --odl-ip-port localhost:8181
--auto-sff-name --nfq-class
```

2.6 SFC OpenFlow Renderer User Guide

2.6.1 Overview

The Service Function Chaining (SFC) OpenFlow Renderer (SFC OF Renderer) implements Service Chaining on OpenFlow switches. It listens for the creation of a Rendered Service Path (RSP) in the operational data store, and once received it programs Service Function Forwarders (SFF) that are hosted on OpenFlow capable switches to forward packets through the service chain. Currently the only tested OpenFlow capable switch is OVS 2.9.

Common acronyms used in the following sections:

- SF - Service Function
- SFF - Service Function Forwarder
- SFC - Service Function Chain
- SFP - Service Function Path
- RSP - Rendered Service Path

2.6.2 SFC OpenFlow Renderer Architecture

The SFC OF Renderer is invoked after a RSP is created in the operational data store using an MD-SAL listener called `SfcOfRspDataListener`. Upon SFC OF Renderer initialization, the `SfcOfRspDataListener` registers itself to listen for RSP changes. When invoked, the `SfcOfRspDataListener` processes the RSP and calls the `SfcOfFlowProgrammerImpl` to create the necessary flows in the Service Function Forwarders configured in the RSP. Refer to the following diagram for more details.

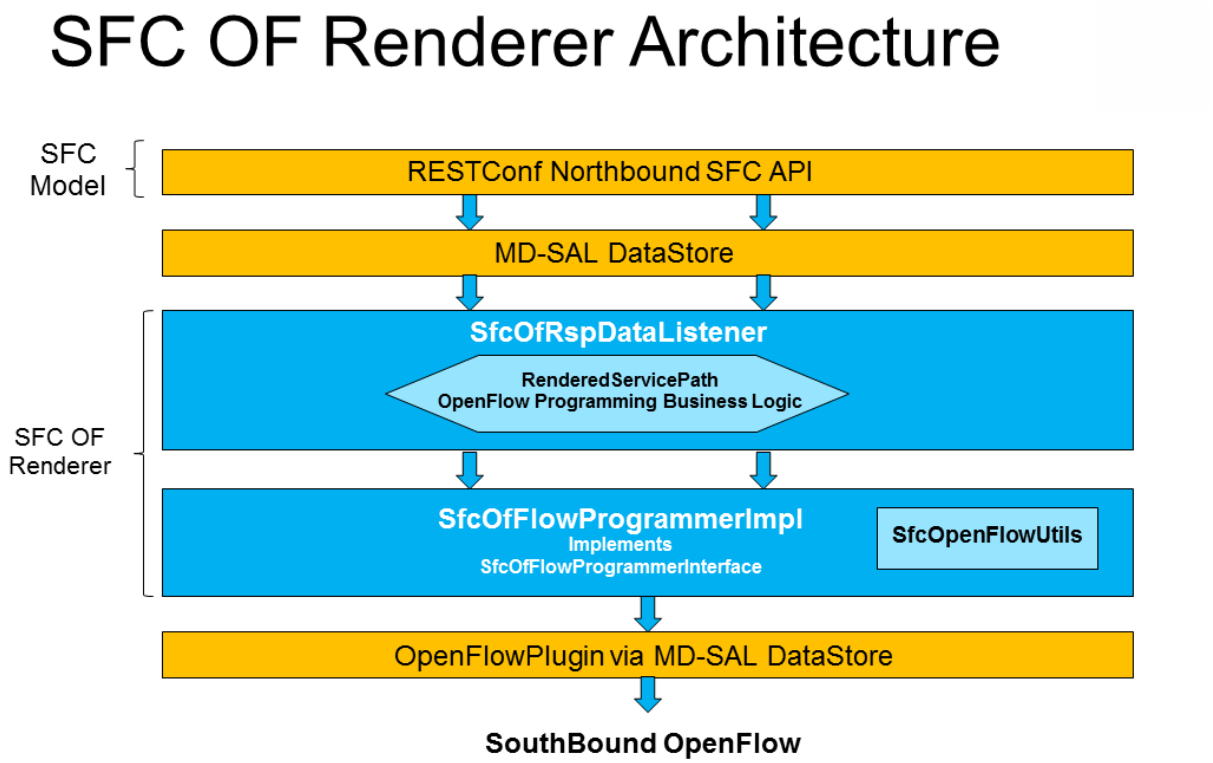


Fig. 4: SFC OpenFlow Renderer High Level Architecture

2.6.3 SFC OpenFlow Switch Flow pipeline

The SFC OpenFlow Renderer uses the following tables for its Flow pipeline:

- Table 0, Classifier
- Table 1, Transport Ingress
- Table 2, Path Mapper
- Table 3, Path Mapper ACL
- Table 4, Next Hop
- Table 10, Transport Egress

The OpenFlow Table Pipeline is intended to be generic to work for all of the different encapsulations supported by SFC.

All of the tables are explained in detail in the following section.

The SFFs (SFF1 and SFF2), SFs (SF1), and topology used for the flow tables in the following sections are as described in the following diagram.

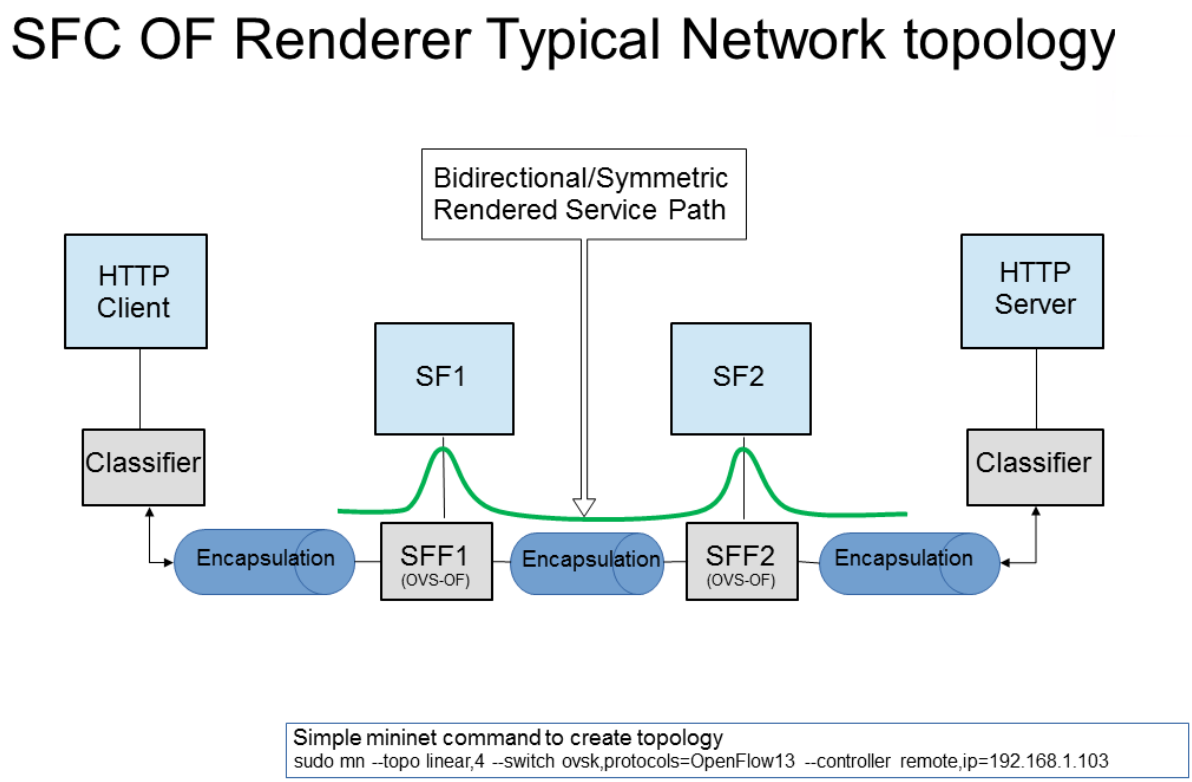


Fig. 5: SFC OpenFlow Renderer Typical Network Topology

Classifier Table detailed

It is possible for the SFF to also act as a classifier. This table maps subscriber traffic to RSPs, and is explained in detail in the classifier documentation.

If the SFF is not a classifier, then this table will just have a simple Goto Table 1 flow.

Transport Ingress Table detailed

The Transport Ingress table has an entry per expected tunnel transport type to be received in a particular SFF, as established in the SFC configuration.

Here are two example on SFF1: one where the RSP ingress tunnel is MPLS assuming VLAN is used for the SFF-SF, and the other where the RSP ingress tunnel is either Eth+NSH or just NSH with no ethernet.

Priority	Match	Action
256	EtherType==0x8847 (MPLS unicast)	Goto Table 2
256	EtherType==0x8100 (VLAN)	Goto Table 2
250	EtherType==0x894f (Eth+NSH)	Goto Table 2
250	PacketType==0x894f (NSH no Eth)	Goto Table 2
5	Match Any	Drop

Table: Table Transport Ingress

Path Mapper Table detailed

The Path Mapper table has an entry per expected tunnel transport info to be received in a particular SFF, as established in the SFC configuration. The tunnel transport info is used to determine the RSP Path ID, and is stored in the OpenFlow Metadata. This table is not used for NSH, since the RSP Path ID is stored in the NSH header.

For SF nodes that do not support NSH tunneling, the IP header DSCP field is used to store the RSP Path Id. The RSP Path Id is written to the DSCP field in the Transport Egress table for those packets sent to an SF.

Here is an example on SFF1, assuming the following details:

- VLAN ID 1000 is used for the SFF-SF
- The RSP Path 1 tunnel uses MPLS label 100 for ingress and 101 for egress
- The RSP Path 2 (symmetric downlink path) uses MPLS label 101 for ingress and 100 for egress

Priority	Match	Action
256	MPLS Label==100	RSP Path=1, Pop MPLS, Goto Table 4
256	MPLS Label==101	RSP Path=2, Pop MPLS, Goto Table 4
256	VLAN ID==1000, IP DSCP==1	RSP Path=1, Pop VLAN, Goto Table 4
256	VLAN ID==1000, IP DSCP==2	RSP Path=2, Pop VLAN, Goto Table 4
5	Match Any	Goto Table 3

Table: Table Path Mapper

Path Mapper ACL Table detailed

This table is only populated when PacketIn packets are received from the switch for TcpProxy type SFs. These flows are created with an inactivity timer of 60 seconds and will be automatically deleted upon expiration.

Next Hop Table detailed

The Next Hop table uses the RSP Path Id and appropriate packet fields to determine where to send the packet next. For NSH, only the NSP (Network Services Path, RSP ID) and NSI (Network Services Index, next hop) fields from the NSH header are needed to determine the VXLAN tunnel destination IP. For VLAN or MPLS, then the source MAC address is used to determine the destination MAC address.

Here are two examples on SFF1, assuming SFF1 is connected to SFF2. RSP Paths 1 and 2 are symmetric VLAN paths. RSP Paths 3 and 4 are symmetric NSH paths. RSP Path 1 ingress packets come from external to SFC, for which we don't have the source MAC address (MacSrc).

Priority	Match	Action
256	RSP Path==1, MacSrc==SF1	MacDst=SFF2, Goto Table 10
256	RSP Path==2, MacSrc==SF1	Goto Table 10
256	RSP Path==2, MacSrc==SFF2	MacDst=SF1, Goto Table 10
246	RSP Path==1	MacDst=SF1, Goto Table 10
550	dl_type=0x894f, nsh_spi=3,nsh_si=255 (NSH, SFF Ingress RSP 3, hop 1)	load:0xa000002→ NXM_NX_TUN_IPV4_DST[], Goto Table 10
550	dl_type=0x894f nsh_spi=3,nsh_si=254 (NSH, SFF Ingress from SF, RSP 3, hop 2)	load:0xa00000a→ NXM_NX_TUN_IPV4_DST[], Goto Table 10
550	dl_type=0x894f, nsh_spi=4,nsh_si=254 (NSH, SFF1 Ingress from SFF2)	load:0xa00000a→ NXM_NX_TUN_IPV4_DST[], Goto Table 10
5	Match Any	Drop

Table: Table Next Hop

Transport Egress Table detailed

The Transport Egress table prepares egress tunnel information and sends the packets out.

Here are two examples on SFF1. RSP Paths 1 and 2 are symmetric MPLS paths that use VLAN for the SFF-SF. RSP Paths 3 and 4 are symmetric NSH paths. Since it is assumed that switches used for NSH will only have one VXLAN port, the NSH packets are just sent back where they came from.

Priority	Match	Action
256	RSP Path==1, MacDst==SF1	Push VLAN ID 1000, Port=SF1
256	RSP Path==1, MacDst==SFF2	Push MPLS Label 101, Port=SFF2
256	RSP Path==2, MacDst==SF1	Push VLAN ID 1000, Port=SF1
246	RSP Path==2	Push MPLS Label 100, Port=Ingress
256	in_port=1,dl_type=0x894f nsh_spi=0x3,nsh_si=255 (NSH, SFF Ingress RSP 3)	IN_PORT
256	in_port=1,dl_type=0x894f, nsh_spi=0x3,nsh_si=254 (NSH,SFF Ingress from SF,RSP 3)	IN_PORT
256 | in_port=1,dl_type=0x894f, nsh_spi=0x4,nsh_si=254 (NSH, SFF1 Ingress from SFF2)		IN_PORT
5	Match Any	Drop

Table: Table Transport Egress

2.6.4 Administering SFC OF Renderer

To use the SFC OpenFlow Renderer Karaf, at least the following Karaf features must be installed.

- odl-openflowplugin-nxm-extensions
- odl-openflowplugin-flow-services
- odl-sfc-provider
- odl-sfc-model
- odl-sfc-openflow-renderer
- odl-sfc-ui (optional)

Since OpenDaylight Karaf features internally install dependent features all of the above features can be installed by simply installing the ''odl-sfc-openflow-renderer" feature.

The following command can be used to view all of the currently installed Karaf features:

```
opendaylight-user@root>feature:list -i
```

Or, pipe the command to a grep to see a subset of the currently installed Karaf features:

```
opendaylight-user@root>feature:list -i | grep sfc
```

To install a particular feature, use the Karaf `feature:install` command.

2.6.5 SFC OF Renderer Tutorial

Overview

In this tutorial, the VXLAN-GPE NSH encapsulations will be shown. The following Network Topology diagram is a logical view of the SFFs and SFs involved in creating the Service Chains.

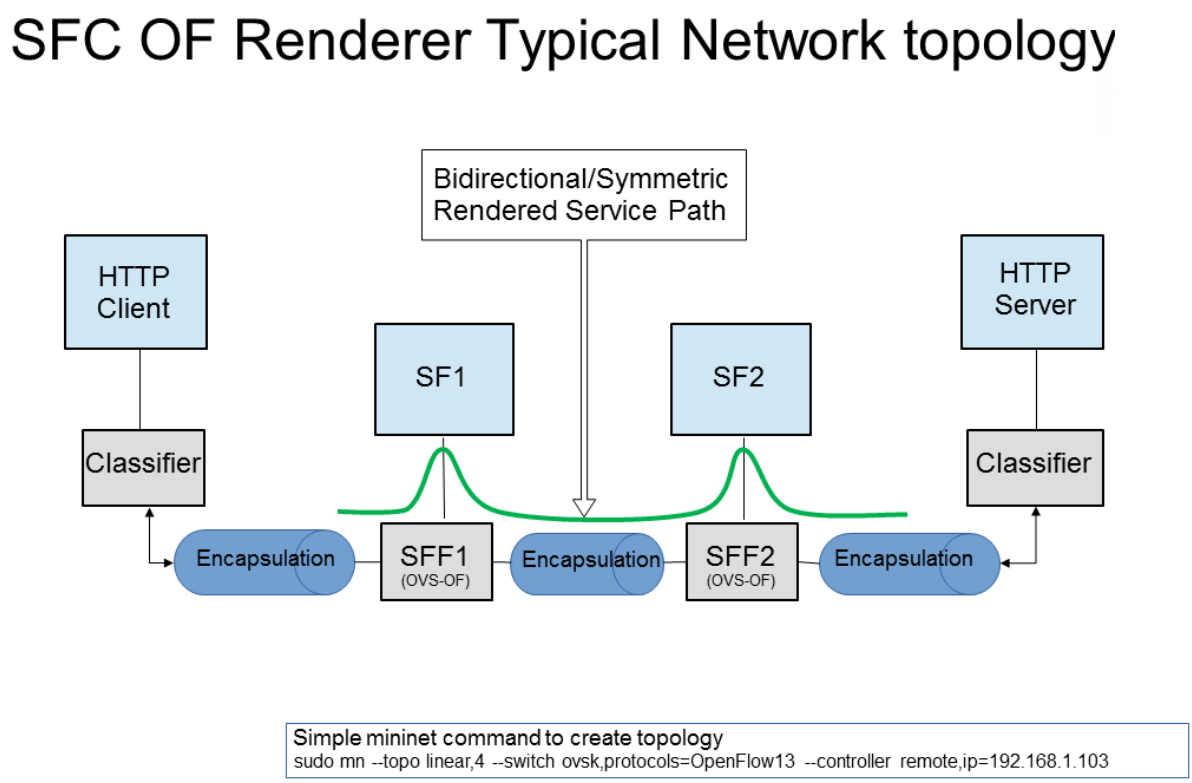


Fig. 6: SFC OpenFlow Renderer Typical Network Topology

Prerequisites

To use this example, SFF OpenFlow switches must be created and connected as illustrated above. Additionally, the SFs must be created and connected.

Note that RSP symmetry depends on the Service Function Path symmetric field, if present. If not, the RSP will be symmetric if any of the SFs involved in the chain has the bidirectional field set to true.

Target Environment

The target environment is not important, but this use-case was created and tested on Linux.

Instructions

The steps to use this tutorial are as follows. The referenced configuration in the steps is listed in the following sections.

There are numerous ways to send the configuration. In the following configuration chapters, the appropriate `curl` command is shown for each configuration to be sent, including the URL.

Steps to configure the SFC OF Renderer tutorial:

1. Send the `SF` RESTCONF configuration
2. Send the `SFF` RESTCONF configuration
3. Send the `SFC` RESTCONF configuration
4. Send the `SFP` RESTCONF configuration
5. The `RSP` will be created internally when the `SFP` is created.

Once the configuration has been successfully created, query the Rendered Service Paths with either the SFC UI or via RESTCONF. Notice that the RSP is symmetrical, so the following 2 RSPs will be created:

- sfc-path1-Path-<RSP-ID>
- sfc-path1-Path-<RSP-ID>-Reverse

At this point the Service Chains have been created, and the OpenFlow Switches are programmed to steer traffic through the Service Chain. Traffic can now be injected from a client into the Service Chain. To debug problems, the OpenFlow tables can be dumped with the following commands, assuming SFF1 is called `s1` and SFF2 is called `s2`.

```
sudo ovs-ofctl -O OpenFlow13  dump-flows s1
```

```
sudo ovs-ofctl -O OpenFlow13  dump-flows s2
```

In all the following configuration sections, replace the `${JSON}` string with the appropriate JSON configuration. Also, change the `localhost` destination in the URL accordingly.

SFC OF Renderer NSH Tutorial

The following configuration sections show how to create the different elements using NSH encapsulation.

NSH Service Function configuration

The Service Function configuration can be sent with the following command:

```
curl -i -H "Content-Type: application/json" -H "Cache-Control: no-cache"
--data '${JSON}' -X PUT --user
admin:admin http://localhost:8181/restconf/config/service-function:service-functions/
```

SF configuration JSON.

```
{
 "service-functions": {
   "service-function": [
     {
       "name": "sf1",
       "type": "http-header-enrichment",
       "ip-mgmt-address": "10.0.0.2",
       "sf-data-plane-locator": [
         {
           "name": "sf1dpl",
           "ip": "10.0.0.10",
           "port": 4789,
           "transport": "service-locator:vxlan-gpe",
           "service-function-forwarder": "sff1"
         }
       ]
     },
     {
       "name": "sf2",
       "type": "firewall",
       "ip-mgmt-address": "10.0.0.3",
       "sf-data-plane-locator": [
         {
           "name": "sf2dpl",
            "ip": "10.0.0.20",
            "port": 4789,
            "transport": "service-locator:vxlan-gpe",
           "service-function-forwarder": "sff2"
         }
       ]
     }
   ]
 }
}
```

NSH Service Function Forwarder configuration

The Service Function Forwarder configuration can be sent with the following command:

```
curl -i -H "Content-Type: application/json" -H "Cache-Control: no-cache" --data '$
↪{JSON}' -X PUT --user admin:admin http://localhost:8181/restconf/config/service-
↪function-forwarder:service-function-forwarders/
```

SFF configuration JSON.

```
{
 "service-function-forwarders": {
   "service-function-forwarder": [
     {
       "name": "sff1",
       "service-node": "openflow:2",
       "sff-data-plane-locator": [
         {
           "name": "sff1dpl",
           "data-plane-locator":
           {
               "ip": "10.0.0.1",
               "port": 4789,
               "transport": "service-locator:vxlan-gpe"
           }
         }
       ],
       "service-function-dictionary": [
         {
           "name": "sf1",
           "sff-sf-data-plane-locator":
           {
               "sf-dpl-name": "sf1dpl",
               "sff-dpl-name": "sff1dpl"
           }
         }
       ]
     },
     {
       "name": "sff2",
       "service-node": "openflow:3",
       "sff-data-plane-locator": [
         {
           "name": "sff2dpl",
           "data-plane-locator":
           {
               "ip": "10.0.0.2",
               "port": 4789,
               "transport": "service-locator:vxlan-gpe"
           }
         }
       ],
       "service-function-dictionary": [
         {
           "name": "sf2",
           "sff-sf-data-plane-locator":
           {
               "sf-dpl-name": "sf2dpl",
               "sff-dpl-name": "sff2dpl"
           }
         }
```

(continues on next page)

(continued from previous page)

```
        ]
      }
    ]
  }
}
```

NSH Service Function Chain configuration

The Service Function Chain configuration can be sent with the following command:

```
curl -i -H "Content-Type: application/json" -H "Cache-Control: no-cache"
--data '${JSON}' -X PUT --user
admin:admin http://localhost:8181/restconf/config/service-function-chain:service-
↪function-chains/
```

SFC configuration JSON.

```
{
 "service-function-chains": {
   "service-function-chain": [
     {
       "name": "sfc-chain1",
       "sfc-service-function": [
         {
           "name": "hdr-enrich-abstract1",
           "type": "http-header-enrichment"
         },
         {
           "name": "firewall-abstract1",
           "type": "firewall"
         }
       ]
     }
   ]
 }
}
```

NSH Service Function Path configuration

The Service Function Path configuration can be sent with the following command:

```
curl -i -H "Content-Type: application/json" -H "Cache-Control: no-cache" --data '$
↪{JSON}' -X PUT --user admin:admin http://localhost:8181/restconf/config/service-
↪function-path:service-function-paths/
```

SFP configuration JSON.

```
{
  "service-function-paths": {
    "service-function-path": [
      {
```

(continues on next page)

(continued from previous page)

```
        "name": "sfc-path1",
        "service-chain-name": "sfc-chain1",
        "transport-type": "service-locator:vxlan-gpe",
        "symmetric": true
      }
    ]
  }
}
```

NSH Rendered Service Path Query

The following command can be used to query all of the created Rendered Service Paths:

```
curl -H "Content-Type: application/json" -H "Cache-Control: no-cache" -X GET --user␣
↪admin:admin http://localhost:8181/restconf/operational/rendered-service-
↪path:rendered-service-paths/
```

SFC OF Renderer MPLS Tutorial

The following configuration sections show how to create the different elements using MPLS encapsulation.

MPLS Service Function configuration

The Service Function configuration can be sent with the following command:

```
curl -i -H "Content-Type: application/json" -H "Cache-Control: no-cache"
--data '${JSON}' -X PUT --user
admin:admin http://localhost:8181/restconf/config/service-function:service-functions/
```

SF configuration JSON.

```
{
 "service-functions": {
   "service-function": [
     {
       "name": "sf1",
       "type": "http-header-enrichment",
       "ip-mgmt-address": "10.0.0.2",
       "sf-data-plane-locator": [
         {
           "name": "sf1-sff1",
           "mac": "00:00:08:01:02:01",
           "vlan-id": 1000,
           "transport": "service-locator:mac",
           "service-function-forwarder": "sff1"
         }
       ]
     },
     {
```

(continues on next page)

(continued from previous page)

```
        "name": "sf2",
        "type": "firewall",
        "ip-mgmt-address": "10.0.0.3",
        "sf-data-plane-locator": [
          {
            "name": "sf2-sff2",
            "mac": "00:00:08:01:03:01",
            "vlan-id": 2000,
            "transport": "service-locator:mac",
            "service-function-forwarder": "sff2"
          }
        ]
      }
    ]
  }
}
```

MPLS Service Function Forwarder configuration

The Service Function Forwarder configuration can be sent with the following command:

```
curl -i -H "Content-Type: application/json" -H "Cache-Control: no-cache" --data '$
↪{JSON}' -X PUT --user admin:admin http://localhost:8181/restconf/config/service-
↪function-forwarder:service-function-forwarders/
```

SFF configuration JSON.

```
{
 "service-function-forwarders": {
   "service-function-forwarder": [
     {
       "name": "sff1",
       "service-node": "openflow:2",
       "sff-data-plane-locator": [
         {
           "name": "ulSff1Ingress",
           "data-plane-locator":
           {
               "mpls-label": 100,
               "transport": "service-locator:mpls"
           },
           "service-function-forwarder-ofs:ofs-port":
           {
               "mac": "11:11:11:11:11:11",
               "port-id" : "1"
           }
         },
         {
           "name": "ulSff1ToSff2",
           "data-plane-locator":
           {
               "mpls-label": 101,
               "transport": "service-locator:mpls"
           },
```

(continues on next page)

(continued from previous page)

```
            "service-function-forwarder-ofs:ofs-port":
            {
                "mac": "33:33:33:33:33:33",
                "port-id" : "2"
            }
          },
          {
            "name": "toSf1",
            "data-plane-locator":
            {
                "mac": "22:22:22:22:22:22",
                "vlan-id": 1000,
                "transport": "service-locator:mac",
            },
            "service-function-forwarder-ofs:ofs-port":
            {
                "mac": "33:33:33:33:33:33",
                "port-id" : "3"
            }
          }
        ],
        "service-function-dictionary": [
          {
            "name": "sf1",
            "sff-sf-data-plane-locator":
            {
                "sf-dpl-name": "sf1-sff1",
                "sff-dpl-name": "toSf1"
            }
          }
        ]
      },
      {
        "name": "sff2",
        "service-node": "openflow:3",
        "sff-data-plane-locator": [
          {
            "name": "ulSff2Ingress",
            "data-plane-locator":
            {
                "mpls-label": 101,
                "transport": "service-locator:mpls"
            },
            "service-function-forwarder-ofs:ofs-port":
            {
                "mac": "44:44:44:44:44:44",
                "port-id" : "1"
            }
          },
          {
            "name": "ulSff2Egress",
            "data-plane-locator":
            {
                "mpls-label": 102,
                "transport": "service-locator:mpls"
            },
            "service-function-forwarder-ofs:ofs-port":
```

(continues on next page)

(continued from previous page)

```
            {
                "mac": "66:66:66:66:66:66",
                "port-id" : "2"
            }
          },
          {
            "name": "toSf2",
            "data-plane-locator":
            {
                "mac": "55:55:55:55:55:55",
                "vlan-id": 2000,
                "transport": "service-locator:mac"
            },
            "service-function-forwarder-ofs:ofs-port":
            {
                "port-id" : "3"
            }
          }
        ],
        "service-function-dictionary": [
          {
            "name": "sf2",
            "sff-sf-data-plane-locator":
            {
                "sf-dpl-name": "sf2-sff2",
                "sff-dpl-name": "toSf2"

            },
            "service-function-forwarder-ofs:ofs-port":
            {
                "port-id" : "3"
            }
          }
        ]
      }
    ]
  }
}
```

MPLS Service Function Chain configuration

The Service Function Chain configuration can be sent with the following command:

```
curl -i -H "Content-Type: application/json" -H "Cache-Control: no-cache"
 --data '${JSON}' -X PUT --user admin:admin
 http://localhost:8181/restconf/config/service-function-chain:service-function-chains/
```

SFC configuration JSON.

```
{
 "service-function-chains": {
   "service-function-chain": [
     {
       "name": "sfc-chain1",
```

(continues on next page)

(continued from previous page)

```
        "sfc-service-function": [
          {
            "name": "hdr-enrich-abstract1",
            "type": "http-header-enrichment"
          },
          {
            "name": "firewall-abstract1",
            "type": "firewall"
          }
        ]
      }
    ]
  }
}
```

MPLS Service Function Path configuration

The Service Function Path configuration can be sent with the following command. This will internally trigger the Rendered Service Paths to be created.

```
curl -i -H "Content-Type: application/json" -H "Cache-Control: no-cache"
--data '${JSON}' -X PUT --user admin:admin
 http://localhost:8181/restconf/config/service-function-path:service-function-paths/
```

SFP configuration JSON.

```
{
  "service-function-paths": {
    "service-function-path": [
      {
        "name": "sfc-path1",
        "service-chain-name": "sfc-chain1",
        "transport-type": "service-locator:mpls",
        "symmetric": true
      }
    ]
  }
}
```

The following command can be used to query all of the Rendered Service Paths that were created when the Service Function Path was created:

```
curl -H "Content-Type: application/json" -H "Cache-Control: no-cache" -X GET
--user admin:admin http://localhost:8181/restconf/operational/rendered-service-
↪path:rendered-service-paths/
```

2.7 SFC IOS XE Renderer User Guide

2.7.1 Overview

The early Service Function Chaining (SFC) renderer for IOS-XE devices (SFC IOS-XE renderer) implements Service Chaining functionality on IOS-XE capable switches. It listens for the creation of a Rendered Service Path (RSP) and sets up Service Function Forwarders (SFF) that are hosted on IOS-XE switches to steer traffic through the service chain.

Common acronyms used in the following sections:

- SF - Service Function
- SFF - Service Function Forwarder
- SFC - Service Function Chain
- SP - Service Path
- SFP - Service Function Path
- RSP - Rendered Service Path
- LSF - Local Service Forwarder
- RSF - Remote Service Forwarder

2.7.2 SFC IOS-XE Renderer Architecture

When the SFC IOS-XE renderer is initialized, all required listeners are registered to handle incoming data. It involves CSR/IOS-XE `NodeListener` which stores data about all configurable devices including their mountpoints (used here as databrokers), `ServiceFunctionListener`, `ServiceForwarderListener` (see mapping) and `RenderedPathListener` used to listen for RSP changes. When the SFC IOS-XE renderer is invoked, `RenderedPathListener` calls the `IosXeRspProcessor` which processes the RSP change and creates all necessary Service Paths and Remote Service Forwarders (if necessary) on IOS-XE devices.

2.7.3 Service Path details

Each Service Path is defined by index (represented by NSP) and contains service path entries. Each entry has appropriate service index (NSI) and definition of next hop. Next hop can be Service Function, different Service Function Forwarder or definition of end of chain - terminate. After terminating, the packet is sent to destination. If a SFF is defined as a next hop, it has to be present on device in the form of Remote Service Forwarder. RSFs are also created during RSP processing.

Example of Service Path:

```
service-chain service-path 200
   service-index 255 service-function firewall-1
   service-index 254 service-function dpi-1
   service-index 253 terminate
```

2.7.4 Mapping to IOS-XE SFC entities

Renderer contains mappers for SFs and SFFs. IOS-XE capable device is using its own definition of Service Functions and Service Function Forwarders according to appropriate .yang file. `ServiceFunctionListener` serves as a listener for SF changes. If SF appears in datastore, listener extracts its management ip address and looks into cached IOS-XE nodes. If some of available nodes match, Service function is mapped in `IosXeServiceFunctionMapper` to be understandable by IOS-XE device and it's written into device's config. `ServiceForwarderListener` is used in a similar way. All SFFs with suitable management ip address it mapped in `IosXeServiceForwarderMapper`. Remapped SFFs are configured as a Local Service Forwarders. It is not possible to directly create Remote Service Forwarder using IOS-XE renderer. RSF is created only during RSP processing.

2.7.5 Administering SFC IOS-XE renderer

To use the SFC IOS-XE Renderer Karaf, at least the following Karaf features must be installed:

- odl-aaa-shiro
- odl-sfc-model
- odl-sfc-provider
- odl-restconf
- odl-netconf-topology
- odl-sfc-ios-xe-renderer

2.7.6 SFC IOS-XE renderer Tutorial

Overview

This tutorial is a simple example how to create Service Path on IOS-XE capable device using IOS-XE renderer

Preconditions

To connect to IOS-XE device, it is necessary to use several modified yang models and override device's ones. All .yang files are in the `Yang/netconf` folder in the `sfc-ios-xe-renderer module` in the SFC project. These files have to be copied to the `cache/schema` directory, before Karaf is started. After that, custom capabilities have to be sent to network-topology:

- PUT ./config/network-topology:network-topology/topology/topology-netconf/node/<device-name>

```
<node xmlns="urn:TBD:params:xml:ns:yang:network-topology">
  <node-id>device-name</node-id>
  <host xmlns="urn:opendaylight:netconf-node-topology">device-ip</host>
  <port xmlns="urn:opendaylight:netconf-node-topology">2022</port>
  <username xmlns="urn:opendaylight:netconf-node-topology">login</username>
  <password xmlns="urn:opendaylight:netconf-node-topology">password</password>
  <tcp-only xmlns="urn:opendaylight:netconf-node-topology">false</tcp-only>
  <keepalive-delay xmlns="urn:opendaylight:netconf-node-topology">0</keepalive-
↪delay>
  <yang-module-capabilities xmlns="urn:opendaylight:netconf-node-topology">
     <override>true</override>
     <capability xmlns="urn:opendaylight:netconf-node-topology">
        urn:ietf:params:xml:ns:yang:ietf-inet-types?module=ietf-inet-types&amp;
↪revision=2013-07-15
```

(continues on next page)

(continued from previous page)

```
      </capability>
      <capability xmlns="urn:opendaylight:netconf-node-topology">
         urn:ietf:params:xml:ns:yang:ietf-yang-types?module=ietf-yang-types&amp;
↪revision=2013-07-15
      </capability>
      <capability xmlns="urn:opendaylight:netconf-node-topology">
         urn:ios?module=ned&amp;revision=2016-03-08
      </capability>
      <capability xmlns="urn:opendaylight:netconf-node-topology">
         http://tail-f.com/yang/common?module=tailf-common&amp;revision=2015-05-22
      </capability>
      <capability xmlns="urn:opendaylight:netconf-node-topology">
         http://tail-f.com/yang/common?module=tailf-meta-extensions&amp;
↪revision=2013-11-07
      </capability>
      <capability xmlns="urn:opendaylight:netconf-node-topology">
         http://tail-f.com/yang/common?module=tailf-cli-extensions&amp;
↪revision=2015-03-19
      </capability>
    </yang-module-capabilities>
</node>
```

Note: The device name in the URL and in the XML must match.

Instructions

When the IOS-XE renderer is installed, all NETCONF nodes in topology-netconf are processed and all capable nodes with accessible mountpoints are cached. The first step is to create LSF on node.

`Service Function Forwarder configuration`

- PUT ./config/service-function-forwarder:service-function-forwarders

```
{
    "service-function-forwarders": {
        "service-function-forwarder": [
            {
                "name": "CSR1Kv-2",
                "ip-mgmt-address": "172.25.73.23",
                "sff-data-plane-locator": [
                    {
                        "name": "CSR1Kv-2-dpl",
                        "data-plane-locator": {
                            "transport": "service-locator:vxlan-gpe",
                            "port": 6633,
                            "ip": "10.99.150.10"
                        }
                    }
                ]
            }
        ]
    }
}
```

If the IOS-XE node with appropriate management IP exists, this configuration is mapped and LSF is created on the device. The same approach is used for Service Functions.

- PUT ./config/service-function:service-functions

```
{
    "service-functions": {
        "service-function": [
            {
                "name": "Firewall",
                "ip-mgmt-address": "172.25.73.23",
                "type": "firewall",
                "sf-data-plane-locator": [
                    {
                        "name": "firewall-dpl",
                        "port": 6633,
                        "ip": "12.1.1.2",
                        "transport": "service-locator:gre",
                        "service-function-forwarder": "CSR1Kv-2"
                    }
                ]
            },
            {
                "name": "Dpi",
                "ip-mgmt-address": "172.25.73.23",
                "type":"dpi",
                "sf-data-plane-locator": [
                    {
                        "name": "dpi-dpl",
                        "port": 6633,
                        "ip": "12.1.1.1",
                        "transport": "service-locator:gre",
                        "service-function-forwarder": "CSR1Kv-2"
                    }
                ]
            },
            {
                "name": "Qos",
                "ip-mgmt-address": "172.25.73.23",
                "type":"qos",
                "sf-data-plane-locator": [
                    {
                        "name": "qos-dpl",
                        "port": 6633,
                        "ip": "12.1.1.4",
                        "transport": "service-locator:gre",
                        "service-function-forwarder": "CSR1Kv-2"
                    }
                ]
            }
        ]
    }
}
```

All these SFs are configured on the same device as the LSF. The next step is to prepare Service Function Chain.

- PUT ./config/service-function-chain:service-function-chains/

```
{
    "service-function-chains": {
        "service-function-chain": [
            {
                "name": "CSR3XSF",
                "sfc-service-function": [
                    {
                        "name": "Firewall",
                        "type": "firewall"
                    },
                    {
                        "name": "Dpi",
                        "type": "dpi"
                    },
                    {
                        "name": "Qos",
                        "type": "qos"
                    }
                ]
            }
        ]
    }
}
```

Service Function Path:

- PUT ./config/service-function-path:service-function-paths/

```
{
    "service-function-paths": {
        "service-function-path": [
            {
                "name": "CSR3XSF-Path",
                "service-chain-name": "CSR3XSF",
                "starting-index": 255,
                "symmetric": "true"
            }
        ]
    }
}
```

Without a classifier, there is possibility to POST RSP directly.

- POST ./operations/rendered-service-path:create-rendered-path

```
{
  "input": {
      "name": "CSR3XSF-Path-RSP",
      "parent-service-function-path": "CSR3XSF-Path"
  }
}
```

The resulting configuration:

```
!
service-chain service-function-forwarder local
  ip address 10.99.150.10
!
```

(continues on next page)

(continued from previous page)

```
service-chain service-function firewall
ip address 12.1.1.2
  encapsulation gre enhanced divert
!
service-chain service-function dpi
ip address 12.1.1.1
  encapsulation gre enhanced divert
!
service-chain service-function qos
ip address 12.1.1.4
  encapsulation gre enhanced divert
!
service-chain service-path 1
  service-index 255 service-function firewall
  service-index 254 service-function dpi
  service-index 253 service-function qos
  service-index 252 terminate
!
service-chain service-path 2
  service-index 255 service-function qos
  service-index 254 service-function dpi
  service-index 253 service-function firewall
  service-index 252 terminate
!
```

Service Path 1 is direct, Service Path 2 is reversed. Path numbers may vary.

2.8 Service Function Scheduling Algorithms

2.8.1 Overview

When creating the Rendered Service Path, the origin SFC controller chose the first available service function from a list of service function names. This may result in many issues such as overloaded service functions and a longer service path as SFC has no means to understand the status of service functions and network topology. The service function selection framework supports at least four algorithms (Random, Round Robin, Load Balancing and Shortest Path) to select the most appropriate service function when instantiating the Rendered Service Path. In addition, it is an extensible framework that allows 3rd party selection algorithm to be plugged in.

2.8.2 Architecture

The following figure illustrates the service function selection framework and algorithms.

A user has three different ways to select one service function selection algorithm:

1. Integrated RESTCONF Calls. OpenStack and/or other administration system could provide plugins to call the APIs to select one scheduling algorithm.
2. Command line tools. Command line tools such as curl or browser plugins such as POSTMAN (for Google Chrome) and RESTClient (for Mozilla Firefox) could select schedule algorithm by making RESTCONF calls.
3. SFC-UI. Now the SFC-UI provides an option for choosing a selection algorithm when creating a Rendered Service Path.

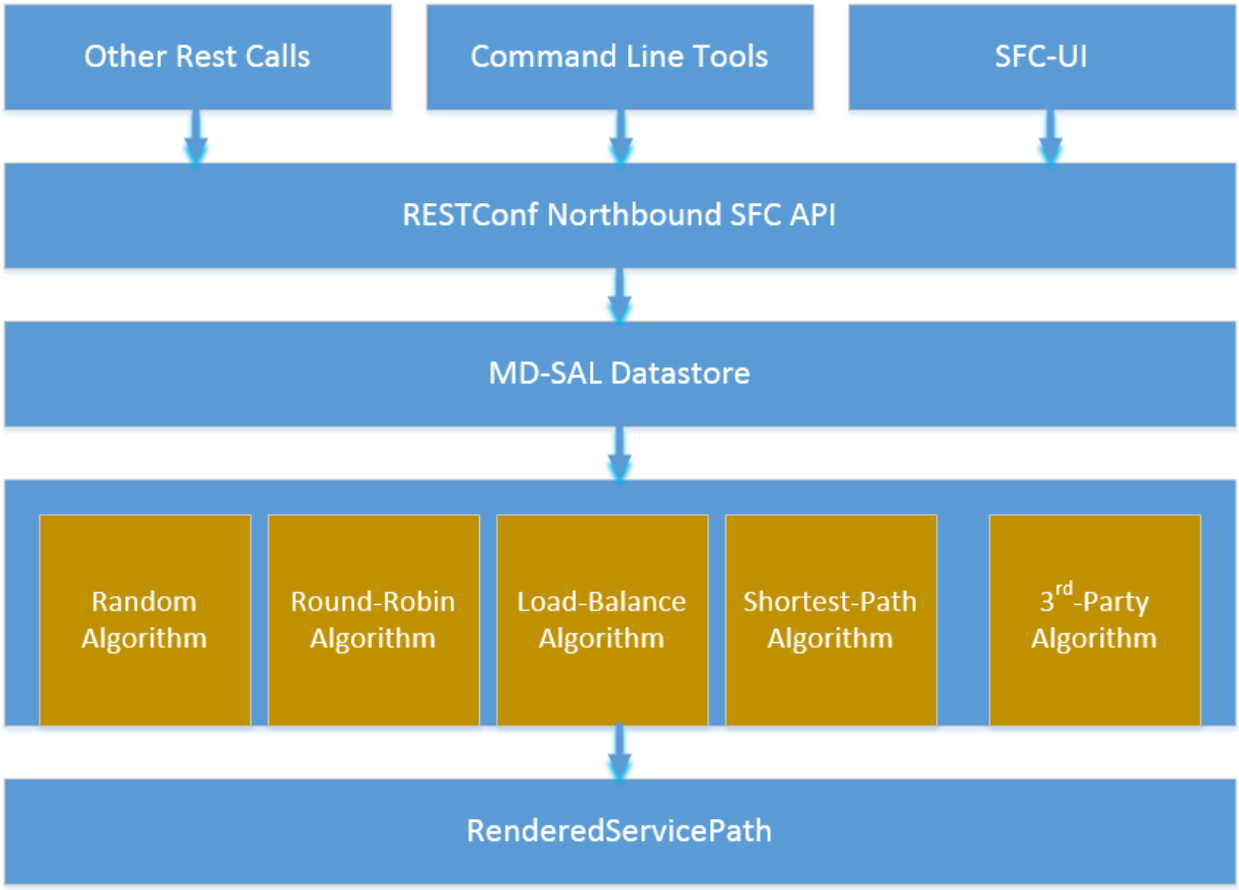


Fig. 7: SF Selection Architecture

The RESTCONF northbound SFC API provides GUI/RESTCONF interactions for choosing the service function selection algorithm. MD-SAL data store provides all supported service function selection algorithms, and provides APIs to enable one of the provided service function selection algorithms. Once a service function selection algorithm is enabled, the service function selection algorithm will work when creating a Rendered Service Path.

2.8.3 Select SFs with Scheduler

Administrator could use both the following ways to select one of the selection algorithm when creating a Rendered Service Path.

- Command line tools. Command line tools includes Linux commands curl or even browser plugins such as POSTMAN(for Google Chrome) or RESTClient(for Mozilla Firefox). In this case, the following JSON content is needed at the moment: Service_function_schudule_type.json

```
{
  "service-function-scheduler-types": {
    "service-function-scheduler-type": [
      {
        "name": "random",
        "type": "service-function-scheduler-type:random",
        "enabled": false
      },
      {
        "name": "roundrobin",
        "type": "service-function-scheduler-type:round-robin",
        "enabled": true
      },
      {
        "name": "loadbalance",
        "type": "service-function-scheduler-type:load-balance",
        "enabled": false
      },
      {
        "name": "shortestpath",
        "type": "service-function-scheduler-type:shortest-path",
        "enabled": false
      }
    ]
  }
}
```

  If using the Linux curl command, it could be:

```
curl -i -H "Content-Type: application/json" -H "Cache-Control: no-cache"
--data '$${Service_function_schudule_type.json}' -X PUT
--user admin:admin http://localhost:8181/restconf/config/service-function-
↪scheduler-type:service-function-scheduler-types/
```

Here is also a snapshot for using the RESTClient plugin:

- SFC-UI.SFC-UI provides a drop down menu for service function selection algorithm. Here is a snapshot for the user interaction from SFC-UI when creating a Rendered Service Path.

Note: Some service function selection algorithms in the drop list are not implemented yet. Only the first three algorithms are committed at the moment.

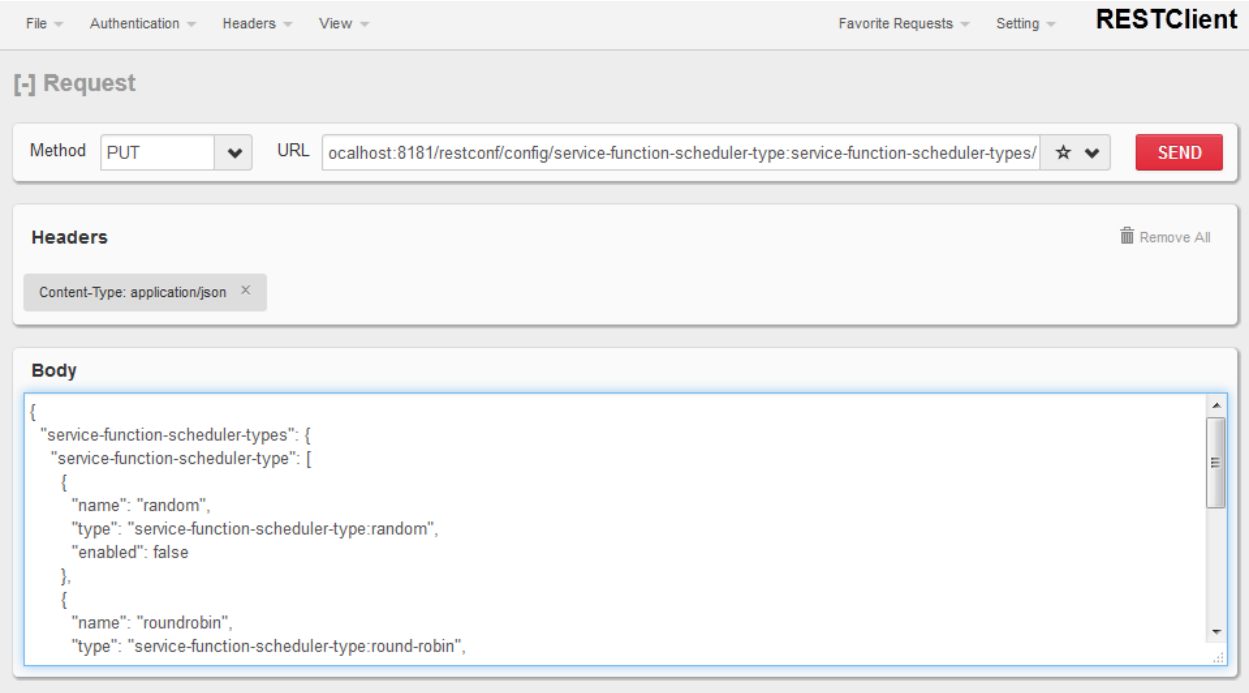


Fig. 8: Mozilla Firefox RESTClient

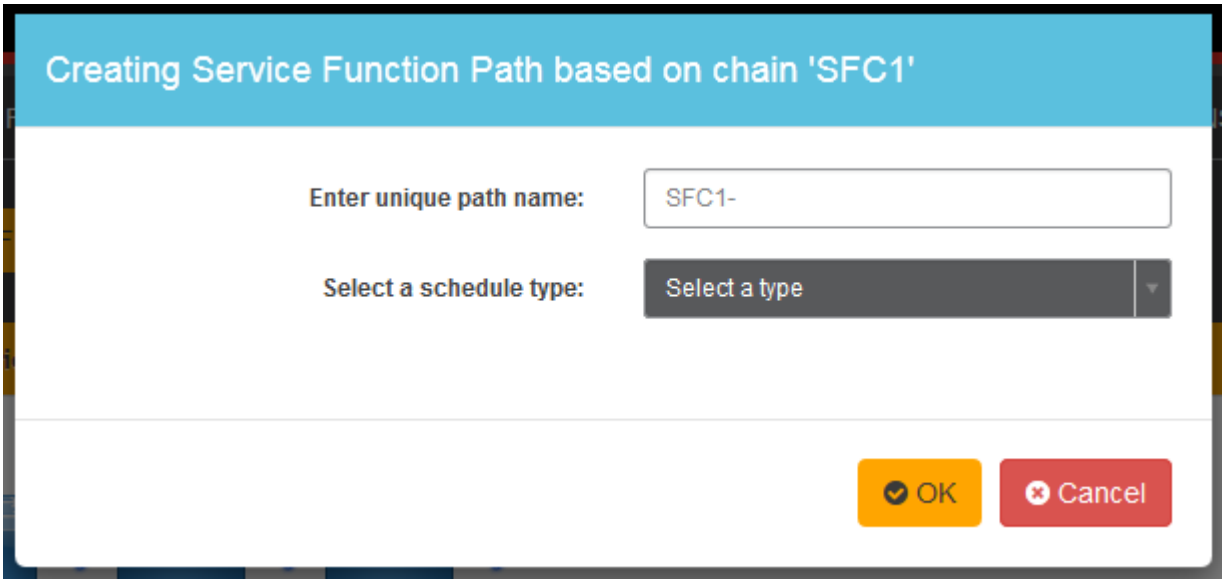


Fig. 9: Karaf Web UI

Random

Select Service Function from the name list randomly.

Overview

The Random algorithm is used to select one Service Function from the name list which it gets from the Service Function Type randomly.

Prerequisites

- Service Function information are stored in datastore.
- Either no algorithm or the Random algorithm is selected.

Target Environment

The Random algorithm will work either no algorithm type is selected or the Random algorithm is selected.

Instructions

Once the plugins are installed into Karaf successfully, a user can use his favorite method to select the Random scheduling algorithm type. There are no special instructions for using the Random algorithm.

Round Robin

Select Service Function from the name list in Round Robin manner.

Overview

The Round Robin algorithm is used to select one Service Function from the name list which it gets from the Service Function Type in a Round Robin manner, this will balance workloads to all Service Functions. However, this method cannot help all Service Functions load the same workload because it's flow-based Round Robin.

Prerequisites

- Service Function information are stored in datastore.
- Round Robin algorithm is selected

Target Environment

The Round Robin algorithm will work one the Round Robin algorithm is selected.

Instructions

Once the plugins are installed into Karaf successfully, a user can use his favorite method to select the Round Robin scheduling algorithm type. There are no special instructions for using the Round Robin algorithm.

Load Balance Algorithm

Select appropriate Service Function by actual CPU utilization.

Overview

The Load Balance Algorithm is used to select appropriate Service Function by actual CPU utilization of service functions. The CPU utilization of service function obtained from monitoring information reported via NETCONF.

Prerequisites

- CPU-utilization for Service Function.
- NETCONF server.
- NETCONF client.
- Each VM has a NETCONF server and it could work with NETCONF client well.

Instructions

Set up VMs as Service Functions. enable NETCONF server in VMs. Ensure that you specify them separately. For example:

a. Set up 4 VMs include 2 SFs' type are Firewall, Others are Napt44. Name them as firewall-1, firewall-2, napt44-1, napt44-2 as Service Function. The four VMs can run either the same server or different servers.

b. Install NETCONF server on every VM and enable it. More information on NETCONF can be found on the OpenDaylight wiki here: https://wiki.opendaylight.org/view/OpenDaylight_Controller:Config:Examples:Netconf:Manual_netopeer_installation

c. Get Monitoring data from NETCONF server. These monitoring data should be get from the NETCONF server which is running in VMs. The following static XML data is an example:

static XML data like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<service-function-description-monitor-report>
  <SF-description>
    <number-of-dataports>2</number-of-dataports>
    <capabilities>
      <supported-packet-rate>5</supported-packet-rate>
      <supported-bandwidth>10</supported-bandwidth>
```

(continues on next page)

(continued from previous page)

```
      <supported-ACL-number>2000</supported-ACL-number>
      <RIB-size>200</RIB-size>
      <FIB-size>100</FIB-size>
      <ports-bandwidth>
        <port-bandwidth>
          <port-id>1</port-id>
          <ipaddress>10.0.0.1</ipaddress>
          <macaddress>00:1e:67:a2:5f:f4</macaddress>
          <supported-bandwidth>20</supported-bandwidth>
        </port-bandwidth>
        <port-bandwidth>
          <port-id>2</port-id>
          <ipaddress>10.0.0.2</ipaddress>
          <macaddress>01:1e:67:a2:5f:f6</macaddress>
          <supported-bandwidth>10</supported-bandwidth>
        </port-bandwidth>
      </ports-bandwidth>
    </capabilities>
  </SF-description>
  <SF-monitoring-info>
    <liveness>true</liveness>
    <resource-utilization>
        <packet-rate-utilization>10</packet-rate-utilization>
        <bandwidth-utilization>15</bandwidth-utilization>
        <CPU-utilization>12</CPU-utilization>
        <memory-utilization>17</memory-utilization>
        <available-memory>8</available-memory>
        <RIB-utilization>20</RIB-utilization>
        <FIB-utilization>25</FIB-utilization>
        <power-utilization>30</power-utilization>
        <SF-ports-bandwidth-utilization>
          <port-bandwidth-utilization>
            <port-id>1</port-id>
            <bandwidth-utilization>20</bandwidth-utilization>
          </port-bandwidth-utilization>
          <port-bandwidth-utilization>
            <port-id>2</port-id>
            <bandwidth-utilization>30</bandwidth-utilization>
          </port-bandwidth-utilization>
        </SF-ports-bandwidth-utilization>
    </resource-utilization>
  </SF-monitoring-info>
</service-function-description-monitor-report>
```

a. Unzip SFC release tarball.

b. Run SFC: ${sfc}/bin/karaf. More information on Service Function Chaining can be found on the OpenDaylight SFC's wiki page: https://wiki.opendaylight.org/view/Service_Function_Chaining:Main

a. Deploy the SFC2 (firewall-abstract2napt44-abstract2) and click button to Create Rendered Service Path in SFC UI (http://localhost:8181/sfc/index.html).

b. Verify the Rendered Service Path to ensure the CPU utilization of the selected hop is the minimum one among all the service functions with same type. The correct RSP is firewall-1napt44-2

Shortest Path Algorithm

Select appropriate Service Function by Dijkstra's algorithm. Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a graph.

Overview

The Shortest Path Algorithm is used to select appropriate Service Function by actual topology.

Prerequisites

- Deployed topology (include SFFs, SFs and their links).
- Dijkstra's algorithm. More information on Dijkstra's algorithm can be found on the wiki here: http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

Instructions

a. Unzip SFC release tarball.

b. Run SFC: ${sfc}/bin/karaf.

c. Depoly SFFs and SFs. import the service-function-forwarders.json and service-functions.json in UI (http://localhost:8181/sfc/index.html#/sfc/config)

service-function-forwarders.json:

```
{
  "service-function-forwarders": {
    "service-function-forwarder": [
      {
        "name": "SFF-br1",
        "service-node": "OVSDB-test01",
        "rest-uri": "http://localhost:5001",
        "sff-data-plane-locator": [
          {
            "name": "eth0",
            "service-function-forwarder-ovs:ovs-bridge": {
              "uuid": "4c3778e4-840d-47f4-b45e-0988e514d26c",
              "bridge-name": "br-tun"
            },
            "data-plane-locator": {
              "port": 5000,
              "ip": "192.168.1.1",
              "transport": "service-locator:vxlan-gpe"
            }
          }
        ],
        "service-function-dictionary": [
          {
            "sff-sf-data-plane-locator": {
               "sf-dpl-name": "sf1dpl",
               "sff-dpl-name": "sff1dpl"
            },
            "name": "napt44-1",
```

(continues on next page)

(continued from previous page)

```
            "type": "napt44"
          },
          {
            "sff-sf-data-plane-locator": {
               "sf-dpl-name": "sf2dpl",
               "sff-dpl-name": "sff2dpl"
            },
            "name": "firewall-1",
            "type": "firewall"
          }
        ],
        "connected-sff-dictionary": [
          {
            "name": "SFF-br3"
          }
        ]
      },
      {
        "name": "SFF-br2",
        "service-node": "OVSDB-test01",
        "rest-uri": "http://localhost:5002",
        "sff-data-plane-locator": [
          {
            "name": "eth0",
            "service-function-forwarder-ovs:ovs-bridge": {
              "uuid": "fd4d849f-5140-48cd-bc60-6ad1f5fc0a1",
              "bridge-name": "br-tun"
            },
            "data-plane-locator": {
              "port": 5000,
              "ip": "192.168.1.2",
              "transport": "service-locator:vxlan-gpe"
            }
          }
        ],
        "service-function-dictionary": [
          {
            "sff-sf-data-plane-locator": {
               "sf-dpl-name": "sf1dpl",
               "sff-dpl-name": "sff1dpl"
            },
            "name": "napt44-2",
            "type": "napt44"
          },
          {
            "sff-sf-data-plane-locator": {
               "sf-dpl-name": "sf2dpl",
               "sff-dpl-name": "sff2dpl"
            },
            "name": "firewall-2",
            "type": "firewall"
          }
        ],
        "connected-sff-dictionary": [
          {
            "name": "SFF-br3"
          }
```

(continues on next page)

(continued from previous page)

```
        ]
      },
      {
        "name": "SFF-br3",
        "service-node": "OVSDB-test01",
        "rest-uri": "http://localhost:5005",
        "sff-data-plane-locator": [
          {
            "name": "eth0",
            "service-function-forwarder-ovs:ovs-bridge": {
              "uuid": "fd4d849f-5140-48cd-bc60-6ad1f5fc0a4",
              "bridge-name": "br-tun"
            },
            "data-plane-locator": {
              "port": 5000,
              "ip": "192.168.1.2",
              "transport": "service-locator:vxlan-gpe"
            }
          }
        ],
        "service-function-dictionary": [
          {
            "sff-sf-data-plane-locator": {
               "sf-dpl-name": "sf1dpl",
               "sff-dpl-name": "sff1dpl"
            },
            "name": "test-server",
            "type": "dpi"
          },
          {
            "sff-sf-data-plane-locator": {
               "sf-dpl-name": "sf2dpl",
               "sff-dpl-name": "sff2dpl"
            },
            "name": "test-client",
            "type": "dpi"
          }
        ],
        "connected-sff-dictionary": [
          {
            "name": "SFF-br1"
          },
          {
            "name": "SFF-br2"
          }
        ]
      }
    ]
  }
}
```

service-functions.json:

```
{
  "service-functions": {
    "service-function": [
      {
```

(continues on next page)

(continued from previous page)

```
        "rest-uri": "http://localhost:10001",
        "ip-mgmt-address": "10.3.1.103",
        "sf-data-plane-locator": [
          {
            "name": "preferred",
            "port": 10001,
            "ip": "10.3.1.103",
            "service-function-forwarder": "SFF-br1"
          }
        ],
        "name": "napt44-1",
        "type": "napt44"
      },
      {
        "rest-uri": "http://localhost:10002",
        "ip-mgmt-address": "10.3.1.103",
        "sf-data-plane-locator": [
          {
            "name": "master",
            "port": 10002,
            "ip": "10.3.1.103",
            "service-function-forwarder": "SFF-br2"
          }
        ],
        "name": "napt44-2",
        "type": "napt44"
      },
      {
        "rest-uri": "http://localhost:10003",
        "ip-mgmt-address": "10.3.1.103",
        "sf-data-plane-locator": [
          {
            "name": "1",
            "port": 10003,
            "ip": "10.3.1.102",
            "service-function-forwarder": "SFF-br1"
          }
        ],
        "name": "firewall-1",
        "type": "firewall"
      },
      {
        "rest-uri": "http://localhost:10004",
        "ip-mgmt-address": "10.3.1.103",
        "sf-data-plane-locator": [
          {
            "name": "2",
            "port": 10004,
            "ip": "10.3.1.101",
            "service-function-forwarder": "SFF-br2"
          }
        ],
        "name": "firewall-2",
        "type": "firewall"
      },
      {
        "rest-uri": "http://localhost:10005",
```

(continues on next page)

(continued from previous page)

```
        "ip-mgmt-address": "10.3.1.103",
        "sf-data-plane-locator": [
          {
            "name": "3",
            "port": 10005,
            "ip": "10.3.1.104",
            "service-function-forwarder": "SFF-br3"
          }
        ],
        "name": "test-server",
        "type": "dpi"
      },
      {
        "rest-uri": "http://localhost:10006",
        "ip-mgmt-address": "10.3.1.103",
        "sf-data-plane-locator": [
          {
            "name": "4",
            "port": 10006,
            "ip": "10.3.1.102",
            "service-function-forwarder": "SFF-br3"
          }
        ],
        "name": "test-client",
        "type": "dpi"
      }
    ]
  }
}
```

The deployed topology like this:

```
          +----+           +----+          +----+
          |sff1|+----------|sff3|----------+|sff2|
          +----+           +----+          +----+
            |                                |
     +--------------+                  +--------------+
     |              |                  |              |
+----------+   +--------+          +----------+   +--------+
|firewall-1|   |napt44-1|          |firewall-2|   |napt44-2|
+----------+   +--------+          +----------+   +--------+
```

- Deploy the SFC2(firewall-abstract2napt44-abstract2), select "Shortest Path" as schedule type and click button to Create Rendered Service Path in SFC UI (http://localhost:8181/sfc/index.html).
- Verify the Rendered Service Path to ensure the selected hops are linked in one SFF. The correct RSP is firewall-1napt44-1 or firewall-2napt44-2. The first SF type is Firewall in Service Function Chain. So the algorithm will select first Hop randomly among all the SFs type is Firewall. Assume the first selected SF is firewall-2. All the path from firewall-1 to SF which type is Napt44 are list:
  - Path1: firewall-2 → sff2 → napt44-2
  - Path2: firewall-2 → sff2 → sff3 → sff1 → napt44-1 The shortest path is Path1, so the selected next hop is napt44-2.

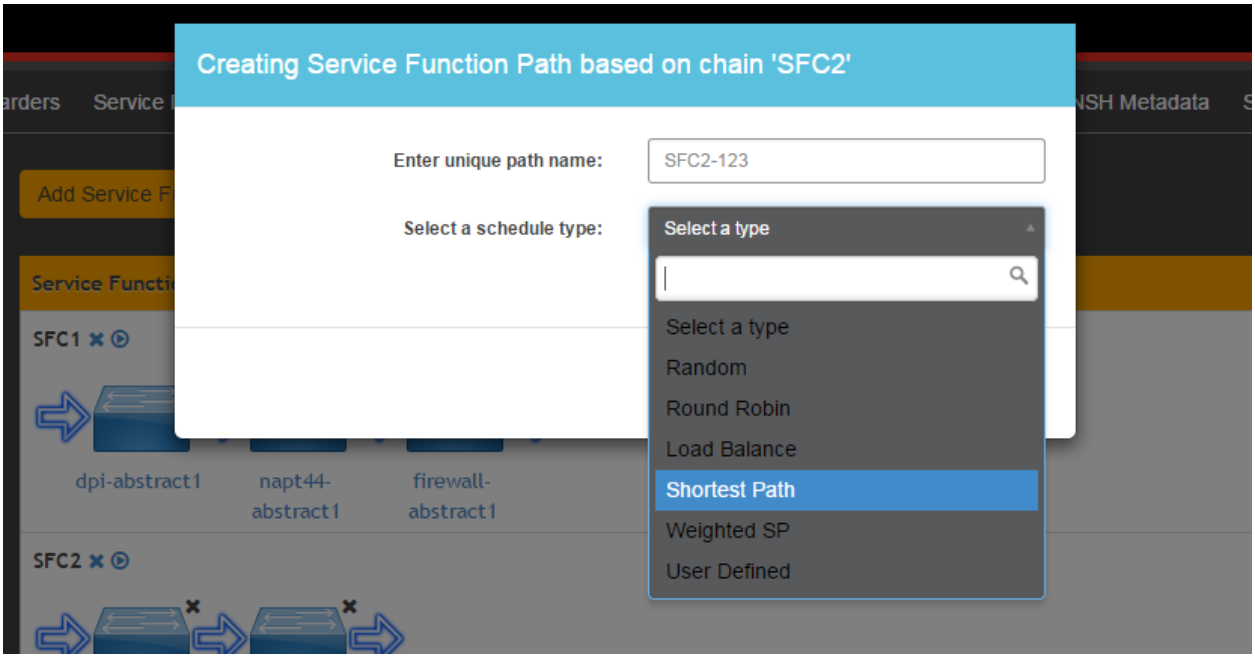


Fig. 10: select schedule type

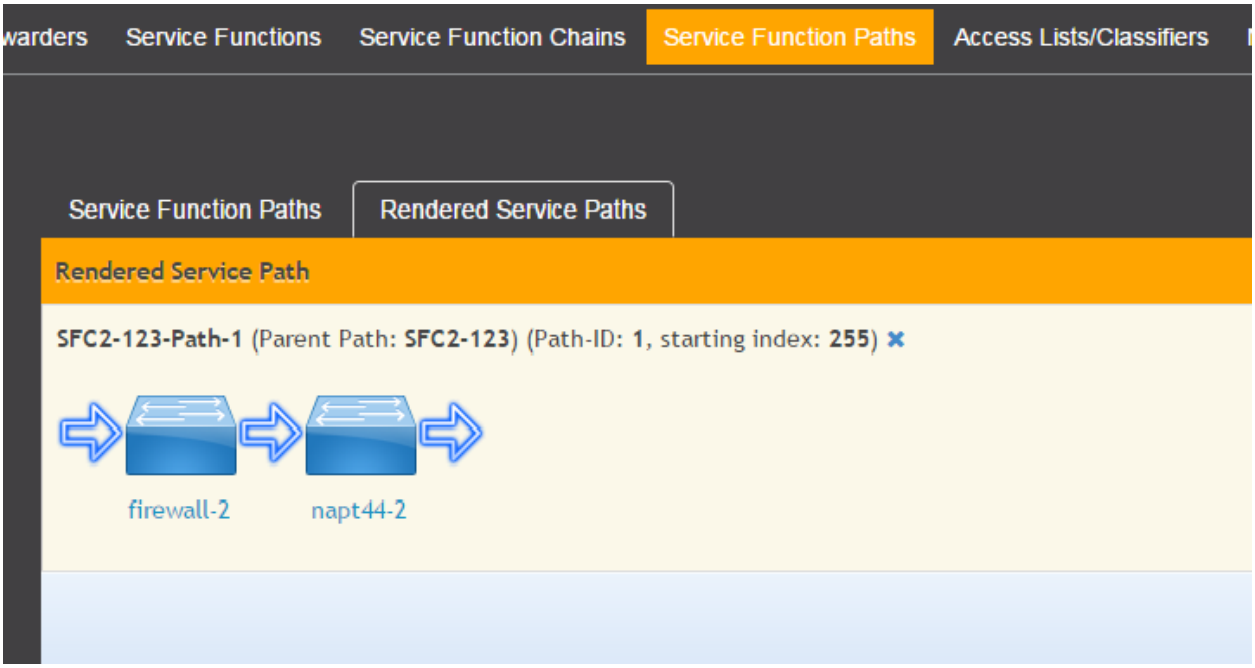


Fig. 11: rendered service path

2.9 Service Function Load Balancing User Guide

2.9.1 Overview

SFC Load-Balancing feature implements load balancing of Service Functions, rather than a one-to-one mapping between Service-Function-Forwarder and Service-Function.

2.9.2 Load Balancing Architecture

Service Function Groups (SFG) can replace Service Functions (SF) in the Rendered Path model. A Service Path can only be defined using SFGs or SFs, but not a combination of both.

Relevant objects in the YANG model are as follows:

1. Service-Function-Group-Algorithm:

```
Service-Function-Group-Algorithms {
    Service-Function-Group-Algorithm {
        String name
        String type
    }
}
```

```
Available types: ALL, SELECT, INDIRECT, FAST_FAILURE
```

2. Service-Function-Group:

```
Service-Function-Groups {
    Service-Function-Group {
        String name
        String serviceFunctionGroupAlgorithmName
        String type
        String groupId
        Service-Function-Group-Element {
            String service-function-name
            int index
        }
    }
}
```

3. ServiceFunctionHop: holds a reference to a name of SFG (or SF)

2.9.3 Tutorials

This tutorial will explain how to create a simple SFC configuration, with SFG instead of SF. In this example, the SFG will include two existing SF.

Setup SFC

For general SFC setup and scenarios, please see the SFC wiki page: https://wiki.opendaylight.org/view/Service_Function_Chaining:Main#SFC_101

Create an algorithm

POST - http://127.0.0.1:8181/restconf/config/service-function-group-algorithm:service-function-group-algorithms

```
{
    "service-function-group-algorithm": [
      {
        "name": "alg1"
        "type": "ALL"
      }
   ]
}
```

(Header "content-type": application/json)

Verify: get all algorithms

GET - http://127.0.0.1:8181/restconf/config/service-function-group-algorithm:service-function-group-algorithms

In order to delete all algorithms: DELETE - http://127.0.0.1:8181/restconf/config/service-function-group-algorithm:service-function-group-algorithms

Create a group

POST - http://127.0.0.1:8181/restconf/config/service-function-group:service-function-groups

```
{
    "service-function-group": [
    {
        "rest-uri": "http://localhost:10002",
        "ip-mgmt-address": "10.3.1.103",
        "algorithm": "alg1",
        "name": "SFG1",
        "type": "napt44",
        "sfc-service-function": [
            {
                "name":"napt44-104"
            },
            {
                "name":"napt44-103-1"
            }
        ]
      }
    ]
}
```

Verify: get all SFG's

GET - http://127.0.0.1:8181/restconf/config/service-function-group:service-function-groups

2.10 SFC Proof of Transit User Guide

2.10.1 Overview

Several deployments use traffic engineering, policy routing, segment routing or service function chaining (SFC) to steer packets through a specific set of nodes. In certain cases regulatory obligations or a compliance policy require to prove that all packets that are supposed to follow a specific path are indeed being forwarded across the exact set of nodes specified. I.e. if a packet flow is supposed to go through a series of service functions or network nodes, it has to be proven that all packets of the flow actually went through the service chain or collection of nodes specified by the policy. In case the packets of a flow weren't appropriately processed, a proof of transit egress device would be required to identify the policy violation and take corresponding actions (e.g. drop or redirect the packet, send an alert etc.) corresponding to the policy.

Service Function Chaining (SFC) Proof of Transit (SFC PoT) implements Service Chaining Proof of Transit functionality on capable network devices. Proof of Transit defines mechanisms to securely prove that traffic transited the defined path. After the creation of an Rendered Service Path (RSP), a user can configure to enable SFC proof of transit on the selected RSP to effect the proof of transit.

To ensure that the data traffic follows a specified path or a function chain, meta-data is added to user traffic in the form of a header. The meta-data is based on a 'share of a secret' and provisioned by the SFC PoT configuration from ODL over a secure channel to each of the nodes in the SFC. This meta-data is updated at each of the service-hop while a designated node called the verifier checks whether the collected meta-data allows the retrieval of the secret.

The following diagram shows the overview and essentially utilizes Shamir's secret sharing algorithm, where each service is given a point on the curve and when the packet travels through each service, it collects these points (meta-data) and a verifier node tries to re-construct the curve using the collected points, thus verifying that the packet traversed through all the service functions along the chain.

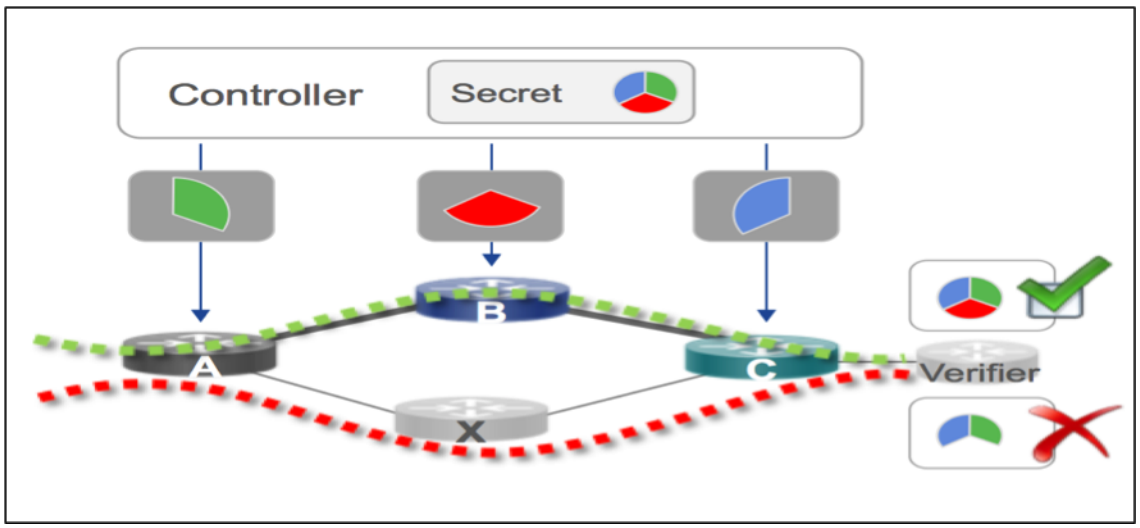


Fig. 12: SFC Proof of Transit overview

Transport options for different protocols includes a new TLV in SR header for Segment Routing, NSH Type-2 meta-data, IPv6 extension headers, IPv4 variants and for VXLAN-GPE. More details are captured in the following link.

In-situ OAM: https://github.com/CiscoDevNet/iOAM

Common acronyms used in the following sections:

- SF - Service Function
- SFF - Service Function Forwarder
- SFC - Service Function Chain
- SFP - Service Function Path
- RSP - Rendered Service Path
- SFC PoT - Service Function Chain Proof of Transit

2.10.2 SFC Proof of Transit Architecture

SFC PoT feature is implemented as a two-part implementation with a north-bound handler that augments the RSP while a south-bound renderer auto-generates the required parameters and passes it on to the nodes that belong to the SFC.

The north-bound feature is enabled via odl-sfc-pot feature while the south-bound renderer is enabled via the odl-sfc-pot-netconf-renderer feature. For the purposes of SFC PoT handling, both features must be installed.

RPC handlers to augment the RSP are part of `SfcPotRpc` while the RSP augmentation to enable or disable SFC PoT feature is done via `SfcPotRspProcessor`.

2.10.3 SFC Proof of Transit entities

In order to implement SFC Proof of Transit for a service function chain, an RSP is a pre-requisite to identify the SFC to enable SFC PoT on. SFC Proof of Transit for a particular RSP is enabled by an RPC request to the controller along with necessary parameters to control some of the aspects of the SFC Proof of Transit process.

The RPC handler identifies the RSP and adds PoT feature meta-data like enable/disable, number of PoT profiles, profiles refresh parameters etc., that directs the south-bound renderer appropriately when RSP changes are noticed via call-backs in the renderer handlers.

2.10.4 Administering SFC Proof of Transit

To use the SFC Proof of Transit Karaf, at least the following Karaf features must be installed:

- odl-sfc-model
- odl-sfc-provider
- odl-sfc-netconf
- odl-restconf
- odl-netconf-topology
- odl-netconf-connector-all
- odl-sfc-pot

Please note that the odl-sfc-pot-netconf-renderer or other renderers in future must be installed for the feature to take full-effect. The details of the renderer features are described in other parts of this document.

2.10.5 SFC Proof of Transit Tutorial

Overview

This tutorial is a simple example how to configure Service Function Chain Proof of Transit using SFC POT feature.

Preconditions

To enable a device to handle SFC Proof of Transit, it is expected that the NETCONF node device advertise capability as under ioam-sb-pot.yang present under sfc-model/src/main/yang folder. It is also expected that base NETCONF support be enabled and its support capability advertised as capabilities.

NETCONF support:`urn:ietf:params:netconf:base:1.0`

PoT support: `(urn:cisco:params:xml:ns:yang:sfc-ioam-sb-pot?revision=2017-01-12)sfc-ioam-sb-pot`

It is also expected that the devices are netconf mounted and available in the topology-netconf store.

Instructions

When SFC Proof of Transit is installed, all netconf nodes in topology-netconf are processed and all capable nodes with accessible mountpoints are cached.

First step is to create the required RSP as is usually done using RSP creation steps in SFC main.

Once RSP name is available it is used to send a POST RPC to the controller similar to below:

POST - http://ODL-IP:8181/restconf/operations/sfc-ioam-nb-pot:enable-sfc-ioam-pot-rendered-path/

```
{
    "input":
    {
        "sfc-ioam-pot-rsp-name": "sfc-path-3sf3sff",
        "ioam-pot-enable":true,
        "ioam-pot-num-profiles":2,
        "ioam-pot-bit-mask":"bits32",
        "refresh-period-time-units":"milliseconds",
        "refresh-period-value":5000
    }
}
```

The following can be used to disable the SFC Proof of Transit on an RSP which disables the PoT feature.

POST - http://ODL-IP:8181/restconf/operations/sfc-ioam-nb-pot:disable-sfc-ioam-pot-rendered-path/

```
{
    "input":
    {
        "sfc-ioam-pot-rsp-name": "sfc-path-3sf3sff",
    }
}
```

2.11 SFC PoT NETCONF Renderer User Guide

2.11.1 Overview

The SFC Proof of Transit (PoT) NETCONF renderer implements SFC Proof of Transit functionality on NETCONF-capable devices, that have advertised support for in-situ OAM (iOAM) support.

It listens for an update to an existing RSP with enable or disable proof of transit support and adds the auto-generated SFC PoT configuration parameters to all the SFC hop nodes. The last node in the SFC is configured as a verifier node to allow SFC PoT process to be completed.

Common acronyms are used as below:

- SF - Service Function
- SFC - Service Function Chain
- RSP - Rendered Service Path
- SFF - Service Function Forwarder

2.11.2 Mapping to SFC entities

The renderer module listens to RSP updates in `SfcPotNetconfRSPListener` and triggers configuration generation in `SfcPotNetconfIoam` class. Node arrival and leaving are managed via `SfcPotNetconfNodeManager` and `SfcPotNetconfNodeListener`. In addition there is a timer thread that runs to generate configuration periodically to refresh the profiles in the nodes that are part of the SFC.

2.11.3 Administering SFC PoT NETCONF Renderer

To use the SFC Proof of Transit Karaf, the following Karaf features must be installed:

- odl-sfc-model
- odl-sfc-provider
- odl-sfc-netconf
- odl-restconf-all
- odl-netconf-topology
- odl-netconf-connector-all
- odl-sfc-pot
- odl-sfc-pot-netconf-renderer

2.11.4 SFC PoT NETCONF Renderer Tutorial

Overview

This tutorial is a simple example how to enable SFC PoT on NETCONF-capable devices.

Preconditions

The NETCONF-capable device will have to support sfc-ioam-sb-pot.yang file.

It is expected that a NETCONF-capable VPP device has Honeycomb (Hc2vpp) Java-based agent that helps to translate between NETCONF and VPP internal APIs.

More details are here: In-situ OAM: https://github.com/CiscoDevNet/iOAM

Steps

When the SFC PoT NETCONF renderer module is installed, all NETCONF nodes in topology-netconf are processed and all sfc-ioam-sb-pot yang capable nodes with accessible mountpoints are cached.

The first step is to create RSP for the SFC as per SFC guidelines above.

Enable SFC PoT is done on the RSP via RESTCONF to the ODL as outlined above.

Internally, the NETCONF renderer will act on the callback to a modified RSP that has PoT enabled.

In-situ OAM algorithms for auto-generation of SFC PoT parameters are generated automatically and sent to these nodes via NETCONF.

2.12 Logical Service Function Forwarder

2.12.1 Overview

Rationale

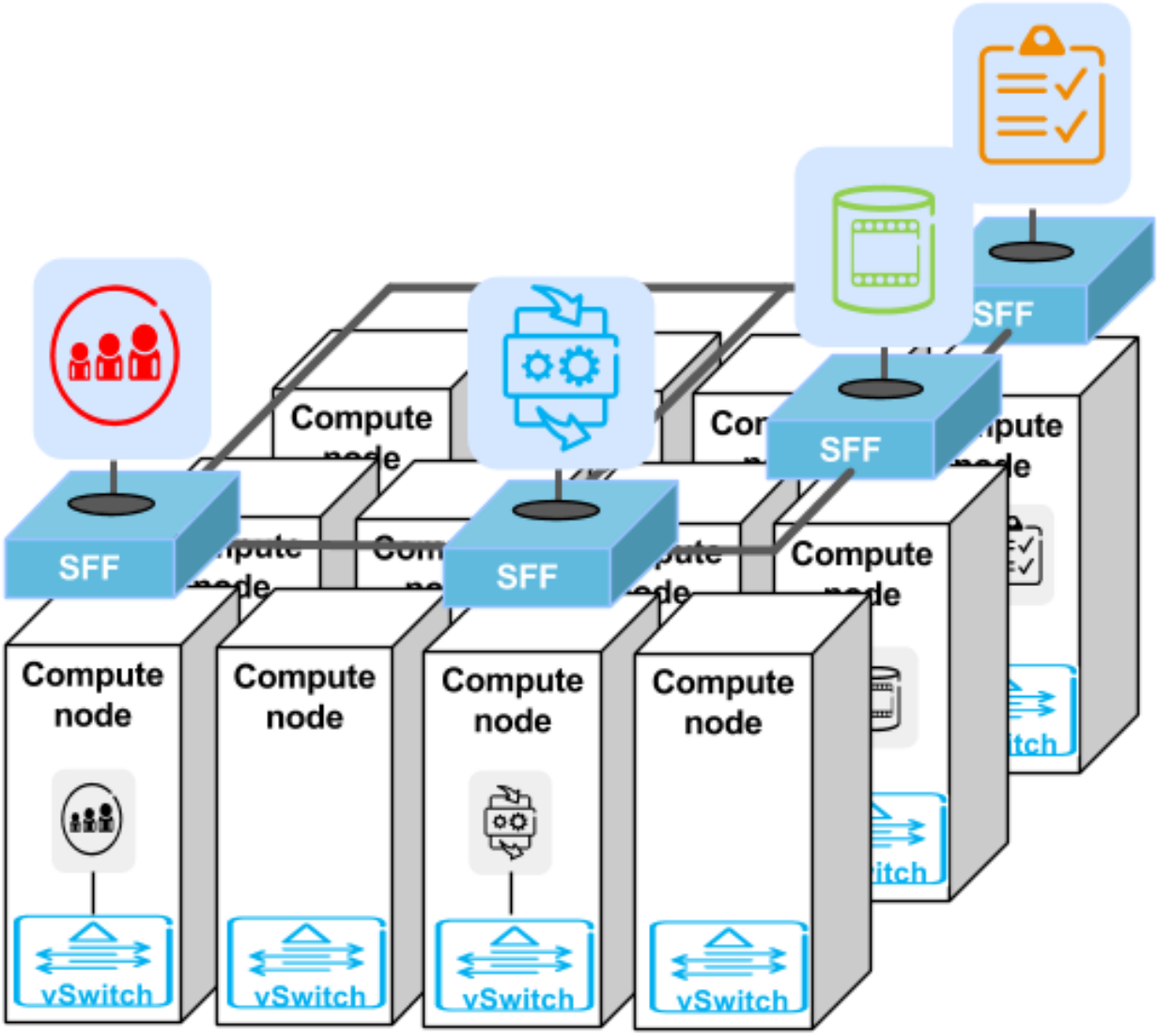
When the current SFC is deployed in a cloud environment, it is assumed that each switch connected to a Service Function is configured as a Service Function Forwarder and each Service Function is connected to its Service Function Forwarder depending on the Compute Node where the Virtual Machine is located.

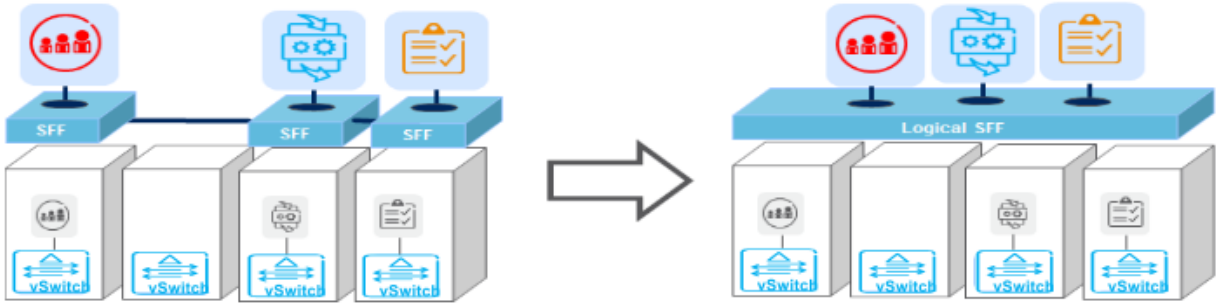
As shown in the picture above, this solution allows the basic cloud use cases to be fulfilled, as for example, the ones required in OPNFV Brahmaputra, however, some advanced use cases like the transparent migration of VMs can not be implemented. The Logical Service Function Forwarder enables the following advanced use cases:

1. Service Function mobility without service disruption
2. Service Functions load balancing and failover

As shown in the picture below, the Logical Service Function Forwarder concept extends the current SFC northbound API to provide an abstraction of the underlying Data Center infrastructure. The Data Center underlaying network can be abstracted by a single SFF. This single SFF uses the logical port UUID as data plane locator to connect SFs globally and in a location-transparent manner. SFC makes use of Genius project to track the location of the SF's logical ports.

The SFC internally distributes the necessary flow state over the relevant switches based on the internal Data Center topology and the deployment of SFs.


SFF
SFF
SFF
Compute node
Compute node
Compute node
Compute node
vSwitch
vSwitch
vSwitch
vSwitch

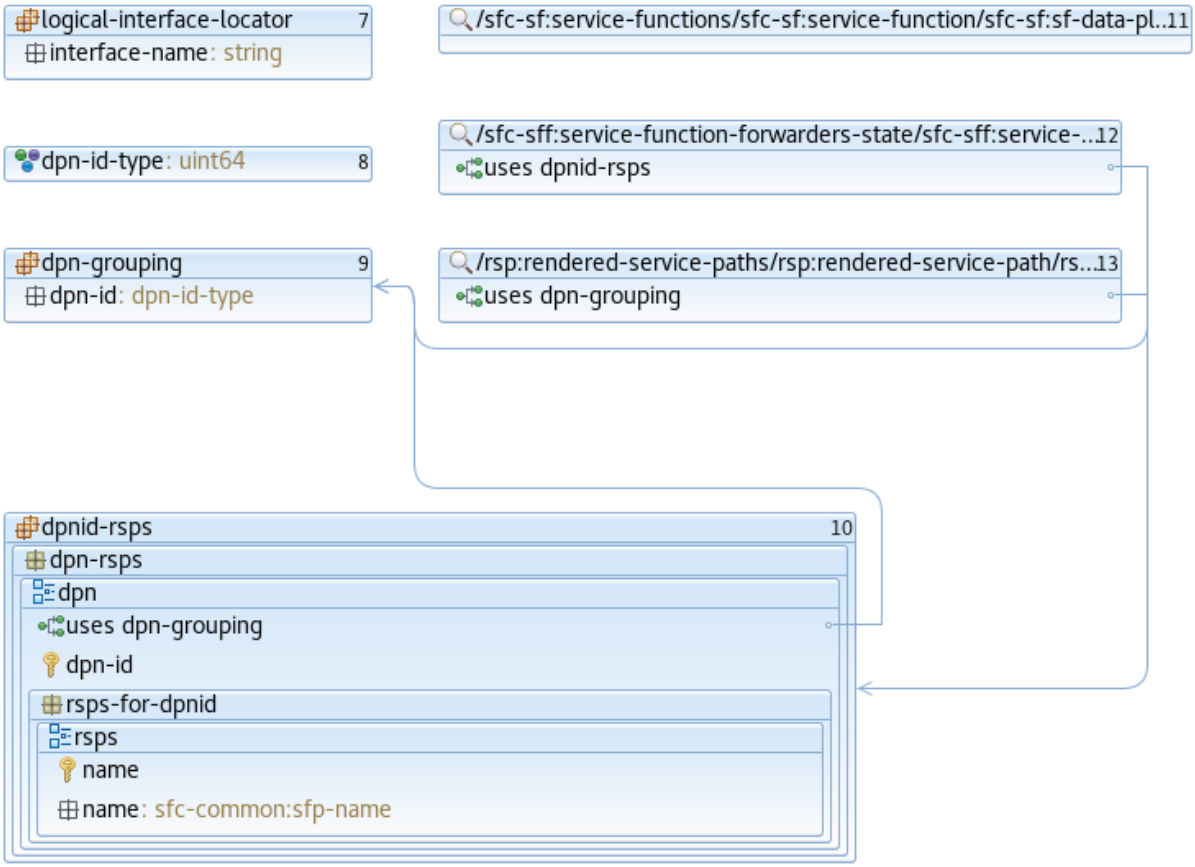

SFF
SFF
SFF
vSwitch
Logical SFF

2.12.2 Changes in data model

The Logical Service Function Forwarder concept extends the current SFC northbound API to provide an abstraction of the underlying Data Center infrastructure.

The Logical SFF simplifies the configuration of the current SFC data model by reducing the number of parameters to be be configured in every SFF, since the controller will discover those parameters by interacting with the services offered by the Genius project.

The following picture shows the Logical SFF data model. The model gets simplified as most of the configuration parameters of the current SFC data model are discovered in runtime. The complete YANG model can be found here logical SFF model.



2.12.3 How to configure the Logical SFF

The following are examples to configure the Logical SFF:

```
curl -i -H "Content-Type: application/json" -H "Cache-Control: no-cache"
--data '${JSON}' -X PUT --user
admin:admin http://localhost:8181/restconf/config/restconf/config/service-
↪function:service-functions/
```

Service Functions JSON.

```
{
"service-functions": {
    "service-function": [
        {
            "name": "firewall-1",
            "type": "firewall",
            "sf-data-plane-locator": [
                {
                    "name": "firewall-dpl",
                    "interface-name": "eccb57ae-5a2e-467f-823e-45d7bb2a6a9a",
                    "transport": "service-locator:eth-nsh",
                    "service-function-forwarder": "sfflogical1"

                }
            ]
        },
        {
            "name": "dpi-1",
            "type": "dpi",
            "sf-data-plane-locator": [
                {
                    "name": "dpi-dpl",
                    "interface-name": "df15ac52-e8ef-4e9a-8340-ae0738aba0c0",
                    "transport": "service-locator:eth-nsh",
                    "service-function-forwarder": "sfflogical1"
                }
            ]
        }
    ]
}
}
```

```
curl -i -H "Content-Type: application/json" -H "Cache-Control: no-cache"
--data '${JSON}' -X PUT --user
admin:admin http://localhost:8181/restconf/config/service-function-forwarder:service-
↪function-forwarders/
```

Service Function Forwarders JSON.

```
{
"service-function-forwarders": {
    "service-function-forwarder": [
       {
            "name": "sfflogical1"
        }
    ]
}
}
```

```
curl -i -H "Content-Type: application/json" -H "Cache-Control: no-cache"
--data '${JSON}' -X PUT --user
admin:admin http://localhost:8181/restconf/config/service-function-chain:service-
↪function-chains/
```

Service Function Chains JSON.

```
{
"service-function-chains": {
    "service-function-chain": [
        {
            "name": "SFC1",
            "sfc-service-function": [
                {
                    "name": "dpi-abstract1",
                    "type": "dpi"
                },
                {
                    "name": "firewall-abstract1",
                    "type": "firewall"
                }
            ]
        },
        {
            "name": "SFC2",
            "sfc-service-function": [
                {
                    "name": "dpi-abstract1",
                    "type": "dpi"
                }
            ]
        }
    ]
}
}
```

```
curl -i -H "Content-Type: application/json" -H "Cache-Control: no-cache"
--data '${JSON}' -X PUT --user
 admin:admin http://localhost:8182/restconf/config/service-function-chain:service-
↪function-paths/
```

Service Function Paths JSON.

```
{
"service-function-paths": {
    "service-function-path": [
        {
            "name": "SFP1",
            "service-chain-name": "SFC1",
            "starting-index": 255,
            "symmetric": "true",
            "context-metadata": "NSH1",
            "transport-type": "service-locator:vxlan-gpe"

        }
    ]
}
}
```

As a result of above configuration, OpenDaylight renders the needed flows in all involved SFFs. Those flows implement:

- Two Rendered Service Paths:
  - dpi-1 (SF1), firewall-1 (SF2)

  - firewall-1 (SF2), dpi-1 (SF1)
- The communication between SFFs and SFs based on eth-nsh
- The communication between SFFs based on vxlan-gpe

The following picture shows a topology and traffic flow (in green) which corresponds to the above configuration.

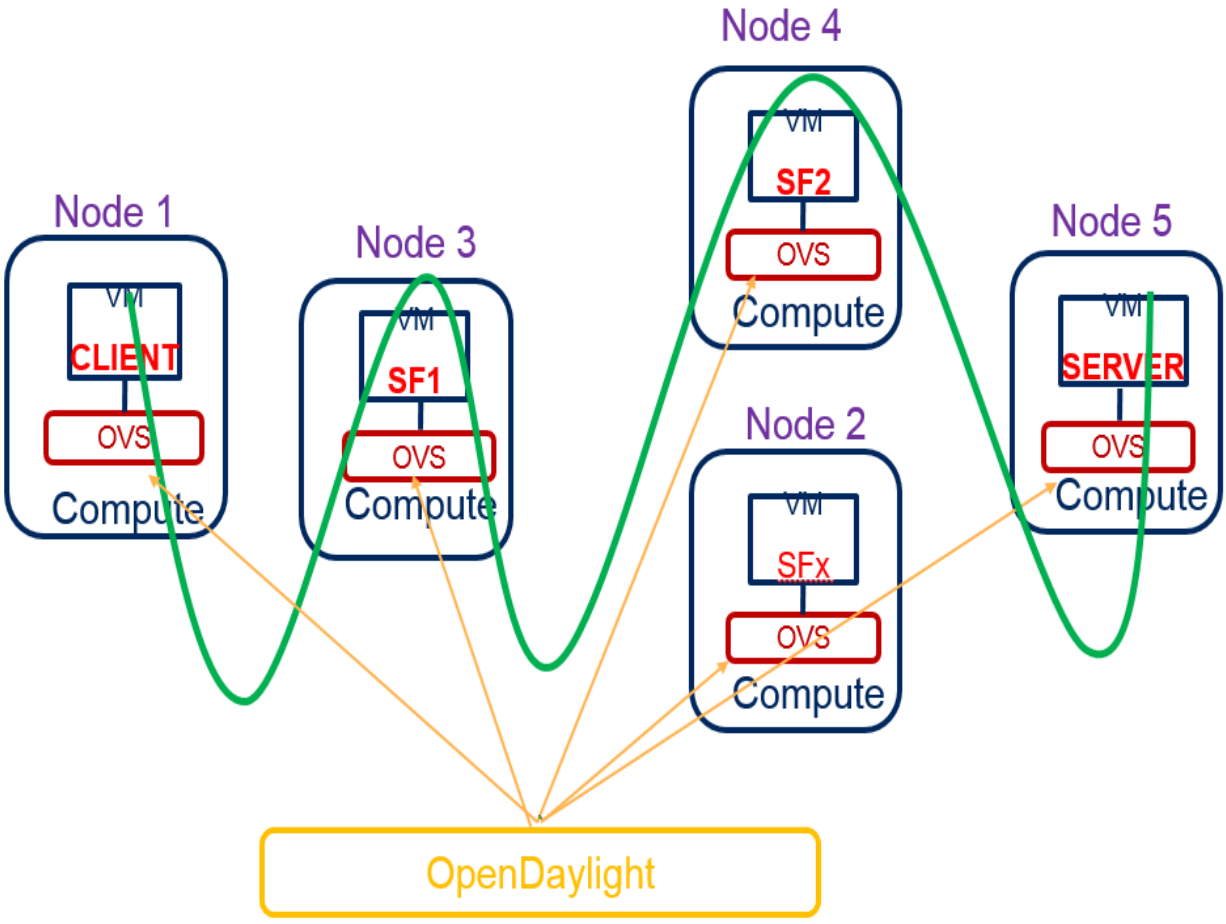


Fig. 13: Logical SFF Example

The Logical SFF functionality allows OpenDaylight to find out the SFFs holding the SFs involved in a path. In this example the SFFs affected are Node3 and Node4 thus the controller renders the flows containing NSH parameters just in those SFFs.

Here you have the new flows rendered in Node3 and Node4 which implement the NSH protocol. Every Rendered Service Path is represented by an NSP value. We provisioned a symmetric RSP so we get two NSPs: 8388613 and 5. Node3 holds the first SF of NSP 8388613 and the last SF of NSP 5. Node 4 holds the first SF of NSP 5 and the last SF of NSP 8388613. Both Node3 and Node4 will pop the NSH header when the received packet has gone through the last SF of its path.

Rendered flows Node 3

```
cookie=0x14, duration=59.264s, table=83, n_packets=0, n_bytes=0, priority=250,nsp=5␣
↪actions=goto_table:86
cookie=0x14, duration=59.194s, table=83, n_packets=0, n_bytes=0, priority=250,
↪nsp=8388613 actions=goto_table:86
```

(continues on next page)

(continued from previous page)

```
cookie=0x14, duration=59.257s, table=86, n_packets=0, n_bytes=0, priority=550,nsi=254,
↪nsp=5 actions=load:0x8e0a37cc9094->NXM_NX_ENCAP_ETH_SRC[],load:0x6ee006b4c51e->NXM_
↪NX_ENCAP_ETH_DST[],goto_table:87
cookie=0x14, duration=59.189s, table=86, n_packets=0, n_bytes=0, priority=550,nsi=255,
↪nsp=8388613 actions=load:0x8e0a37cc9094->NXM_NX_ENCAP_ETH_SRC[],load:0x6ee006b4c51e-
↪>NXM_NX_ENCAP_ETH_DST[],goto_table:87
cookie=0xba5eba1100000203, duration=59.213s, table=87, n_packets=0, n_bytes=0,␣
↪priority=650,nsi=253,nsp=5 actions=pop_nsh,set_field:6e:e0:06:b4:c5:1e->eth_src,
↪resubmit(,17)
cookie=0xba5eba1100000201, duration=59.213s, table=87, n_packets=0, n_bytes=0,␣
↪priority=650,nsi=254,nsp=5 actions=load:0x800->NXM_NX_REG6[],resubmit(,220)
cookie=0xba5eba1100000201, duration=59.188s, table=87, n_packets=0, n_bytes=0,␣
↪priority=650,nsi=255,nsp=8388613 actions=load:0x800->NXM_NX_REG6[],resubmit(,220)
cookie=0xba5eba1100000201, duration=59.182s, table=87, n_packets=0, n_bytes=0,␣
↪priority=650,nsi=254,nsp=8388613 actions=set_field:0->tun_id,output:6
```

Rendered Flows Node 4

```
cookie=0x14, duration=69.040s, table=83, n_packets=0, n_bytes=0, priority=250,nsp=5␣
↪actions=goto_table:86
cookie=0x14, duration=69.008s, table=83, n_packets=0, n_bytes=0, priority=250,
↪nsp=8388613 actions=goto_table:86
cookie=0x14, duration=69.040s, table=86, n_packets=0, n_bytes=0, priority=550,nsi=255,
↪nsp=5 actions=load:0xbea93873f4fa->NXM_NX_ENCAP_ETH_SRC[],load:0x214845ea85d->NXM_
↪NX_ENCAP_ETH_DST[],goto_table:87
cookie=0x14, duration=69.005s, table=86, n_packets=0, n_bytes=0, priority=550,nsi=254,
↪nsp=8388613 actions=load:0xbea93873f4fa->NXM_NX_ENCAP_ETH_SRC[],load:0x214845ea85d->
↪NXM_NX_ENCAP_ETH_DST[],goto_table:87
cookie=0xba5eba1100000201, duration=69.029s, table=87, n_packets=0, n_bytes=0,␣
↪priority=650,nsi=255,nsp=5 actions=load:0x1100->NXM_NX_REG6[],resubmit(,220)
cookie=0xba5eba1100000201, duration=69.029s, table=87, n_packets=0, n_bytes=0,␣
↪priority=650,nsi=254,nsp=5 actions=set_field:0->tun_id,output:1
cookie=0xba5eba1100000201, duration=68.999s, table=87, n_packets=0, n_bytes=0,␣
↪priority=650,nsi=254,nsp=8388613 actions=load:0x1100->NXM_NX_REG6[],resubmit(,220)
cookie=0xba5eba1100000203, duration=68.996s, table=87, n_packets=0, n_bytes=0,␣
↪priority=650,nsi=253,nsp=8388613 actions=pop_nsh,set_field:02:14:84:5e:a8:5d->eth_
↪src,resubmit(,17)
```

An interesting scenario to show the Logical SFF strength is the migration of a SF from a compute node to another. The OpenDaylight will learn the new topology by itself, then it will re-render the new flows to the new SFFs affected.

In our example, SF2 is moved from Node4 to Node2 then OpenDaylight removes NSH specific flows from Node4 and puts them in Node2. Check below flows showing this effect. Now Node3 keeps holding the first SF of NSP 8388613 and the last SF of NSP 5; but Node2 becomes the new holder of the first SF of NSP 5 and the last SF of NSP 8388613.

Rendered Flows Node 3 After Migration

```
cookie=0x14, duration=64.044s, table=83, n_packets=0, n_bytes=0, priority=250,nsp=5␣
↪actions=goto_table:86
cookie=0x14, duration=63.947s, table=83, n_packets=0, n_bytes=0, priority=250,
↪nsp=8388613 actions=goto_table:86
cookie=0x14, duration=64.044s, table=86, n_packets=0, n_bytes=0, priority=550,nsi=254,
↪nsp=5 actions=load:0x8e0a37cc9094->NXM_NX_ENCAP_ETH_SRC[],load:0x6ee006b4c51e->NXM_
↪NX_ENCAP_ETH_DST[],goto_table:87
cookie=0x14, duration=63.947s, table=86, n_packets=0, n_bytes=0, priority=550,nsi=255,
↪nsp=8388613 actions=load:0x8e0a37cc9094->NXM_NX_ENCAP_ETH_SRC[],load:0x6ee006b4c51e-
↪>NXM_NX_ENCAP_ETH_DST[],goto_table:87
```

(continues on next page)

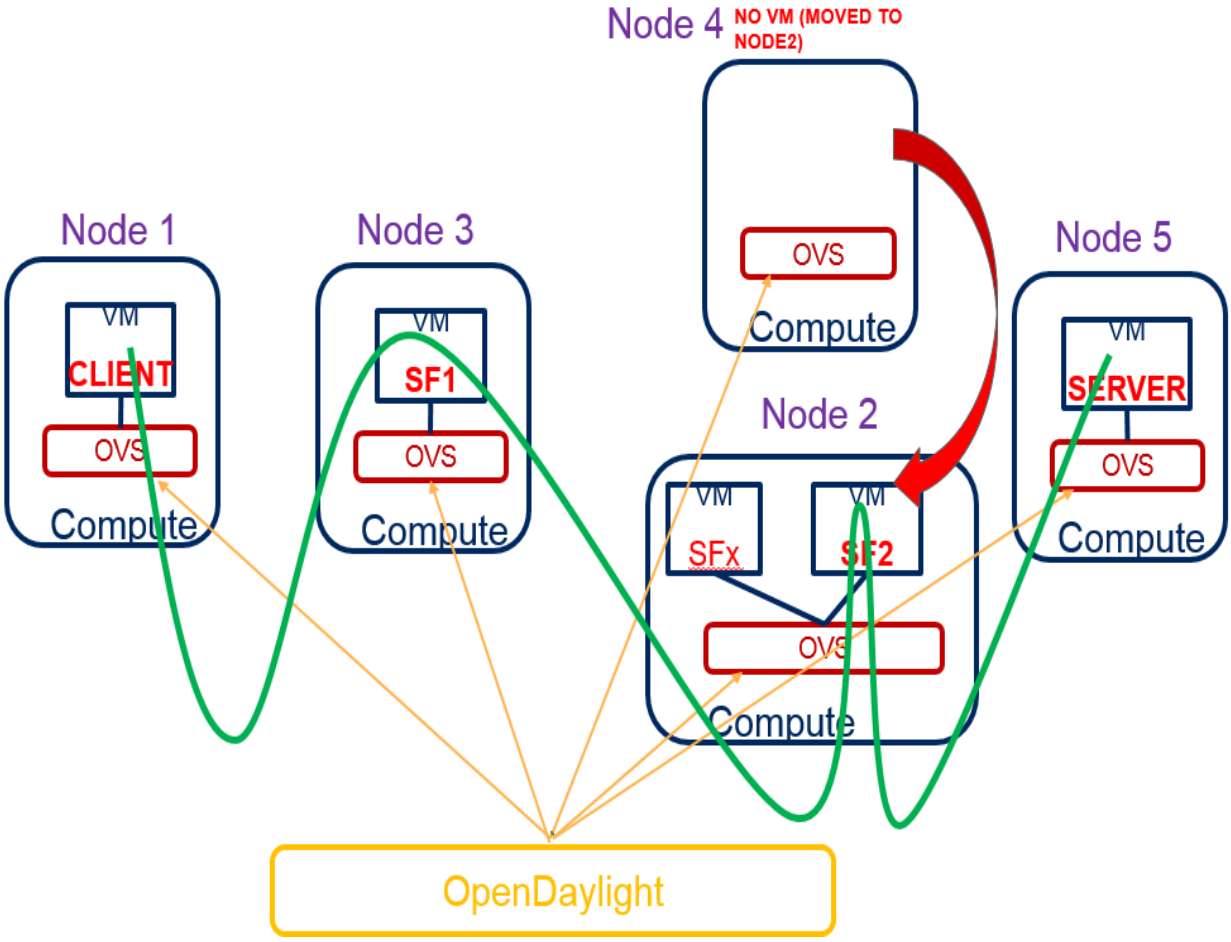


Fig. 14: Logical SFF - SF Migration Example

(continued from previous page)

```
cookie=0xba5eba1100000201, duration=64.034s, table=87, n_packets=0, n_bytes=0,␣
↪priority=650,nsi=254,nsp=5 actions=load:0x800->NXM_NX_REG6[],resubmit(,220)
cookie=0xba5eba1100000203, duration=64.034s, table=87, n_packets=0, n_bytes=0,␣
↪priority=650,nsi=253,nsp=5 actions=pop_nsh,set_field:6e:e0:06:b4:c5:1e->eth_src,
↪resubmit(,17)
cookie=0xba5eba1100000201, duration=63.947s, table=87, n_packets=0, n_bytes=0,␣
↪priority=650,nsi=255,nsp=8388613 actions=load:0x800->NXM_NX_REG6[],resubmit(,220)
cookie=0xba5eba1100000201, duration=63.942s, table=87, n_packets=0, n_bytes=0,␣
↪priority=650,nsi=254,nsp=8388613 actions=set_field:0->tun_id,output:2
```

Rendered Flows Node 2 After Migration

```
cookie=0x14, duration=56.856s, table=83, n_packets=0, n_bytes=0, priority=250,nsp=5␣
↪actions=goto_table:86
cookie=0x14, duration=56.755s, table=83, n_packets=0, n_bytes=0, priority=250,
↪nsp=8388613 actions=goto_table:86
cookie=0x14, duration=56.847s, table=86, n_packets=0, n_bytes=0, priority=550,nsi=255,
↪nsp=5 actions=load:0xbea93873f4fa->NXM_NX_ENCAP_ETH_SRC[],load:0x214845ea85d->NXM_
↪NX_ENCAP_ETH_DST[],goto_table:87
cookie=0x14, duration=56.755s, table=86, n_packets=0, n_bytes=0, priority=550,nsi=254,
↪nsp=8388613 actions=load:0xbea93873f4fa->NXM_NX_ENCAP_ETH_SRC[],load:0x214845ea85d->
↪NXM_NX_ENCAP_ETH_DST[],goto_table:87
cookie=0xba5eba1100000201, duration=56.823s, table=87, n_packets=0, n_bytes=0,␣
↪priority=650,nsi=255,nsp=5 actions=load:0x1100->NXM_NX_REG6[],resubmit(,220)
cookie=0xba5eba1100000201, duration=56.823s, table=87, n_packets=0, n_bytes=0,␣
↪priority=650,nsi=254,nsp=5 actions=set_field:0->tun_id,output:4
cookie=0xba5eba1100000201, duration=56.755s, table=87, n_packets=0, n_bytes=0,␣
↪priority=650,nsi=254,nsp=8388613 actions=load:0x1100->NXM_NX_REG6[],resubmit(,220)
cookie=0xba5eba1100000203, duration=56.750s, table=87, n_packets=0, n_bytes=0,␣
↪priority=650,nsi=253,nsp=8388613 actions=pop_nsh,set_field:02:14:84:5e:a8:5d->eth_
↪src,resubmit(,17)
```

Rendered Flows Node 4 After Migration

```
-- No flows for NSH processing --
```

2.12.4 Classifier impacts

As previously mentioned, in the *Logical SFF rationale*, the Logical SFF feature relies on Genius to get the dataplane IDs of the OpenFlow switches, in order to properly steer the traffic through the chain.

Since one of the classifier's objectives is to steer the packets *into* the SFC domain, the classifier has to be aware of where the first Service Function is located - if it migrates somewhere else, the classifier table has to be updated accordingly, thus enabling the seemless migration of Service Functions.

For this feature, mobility of the client VM is out of scope, and should be managed by its high-availability module, or VNF manager.

Keep in mind that classification *always* occur in the compute-node where the client VM (i.e. traffic origin) is running.

2.12.5 How to attach the classifier to a Logical SFF

In order to leverage this functionality, the classifier has to be configured using a Logical SFF as an attachment-point, specifying within it the neutron port to classify.

The following examples show how to configure an ACL, and a classifier having a Logical SFF as an attachment-point:

Configure an ACL

The following ACL enables traffic intended for port 80 within the subnetwork 192.168.2.0/24, for RSP1 and RSP1-Reverse.

```
{
  "access-lists": {
    "acl": [
      {
        "acl-name": "ACL1",
        "acl-type": "ietf-access-control-list:ipv4-acl",
        "access-list-entries": {
          "ace": [
            {
              "rule-name": "ACE1",
              "actions": {
                "service-function-acl:rendered-service-path": "RSP1"
              },
              "matches": {
                "destination-ipv4-network": "192.168.2.0/24",
                "source-ipv4-network": "192.168.2.0/24",
                "protocol": "6",
                "source-port-range": {
                    "lower-port": 0
                },
                "destination-port-range": {
                    "lower-port": 80
                }
              }
            }
          ]
        }
      },
      {
        "acl-name": "ACL2",
        "acl-type": "ietf-access-control-list:ipv4-acl",
        "access-list-entries": {
          "ace": [
            {
              "rule-name": "ACE2",
              "actions": {
                "service-function-acl:rendered-service-path": "RSP1-Reverse"
              },
              "matches": {
                "destination-ipv4-network": "192.168.2.0/24",
                "source-ipv4-network": "192.168.2.0/24",
                "protocol": "6",
                "source-port-range": {
                    "lower-port": 80
                },
                "destination-port-range": {
                    "lower-port": 0
```

(continues on next page)

(continued from previous page)

```
                }
              }
            }
          ]
        }
      }
    ]
  }
}
```

```
curl -i -H "Content-Type: application/json" -H "Cache-Control: no-cache"
--data '${JSON}' -X PUT --user
admin:admin http://localhost:8181/restconf/config/ietf-access-control-list:access-
↪lists/
```

Configure a classifier JSON

The following JSON provisions a classifier, having a Logical SFF as an attachment point. The value of the field 'interface' is where you indicate the neutron ports of the VMs you want to classify.

```
{
  "service-function-classifiers": {
    "service-function-classifier": [
      {
        "name": "Classifier1",
        "scl-service-function-forwarder": [
          {
            "name": "sfflogical1",
            "interface": "09a78ba3-78ba-40f5-a3ea-1ce708367f2b"
          }
        ],
        "acl": {
            "name": "ACL1",
            "type": "ietf-access-control-list:ipv4-acl"
         }
      }
    ]
  }
}
```

```
curl -i -H "Content-Type: application/json" -H "Cache-Control: no-cache"
--data '${JSON}' -X PUT --user
admin:admin http://localhost:8181/restconf/config/service-function-classifier:service-
↪function-classifiers/
```

2.12.6 SFC pipeline impacts

After binding SFC service with a particular interface by means of Genius, as explained in the Genius User Guide, the entry point in the SFC pipeline will be table 82 (SFC_TRANSPORT_CLASSIFIER_TABLE), and from that point, packet processing will be similar to the *SFC OpenFlow pipeline*, just with another set of specific tables for the SFC service.

This picture shows the SFC pipeline after service integration with Genius:

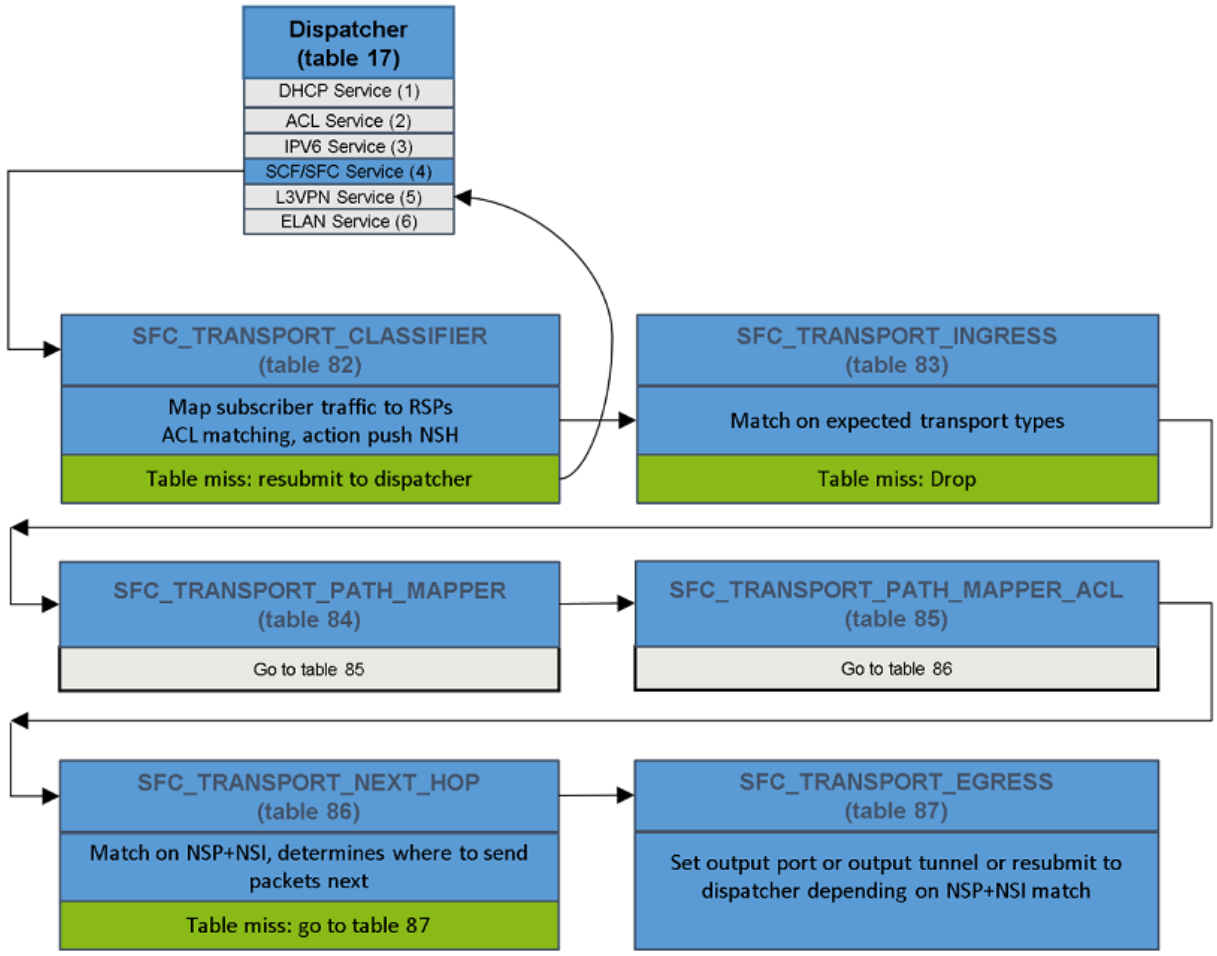


Fig. 15: SFC Logical SFF OpenFlow pipeline

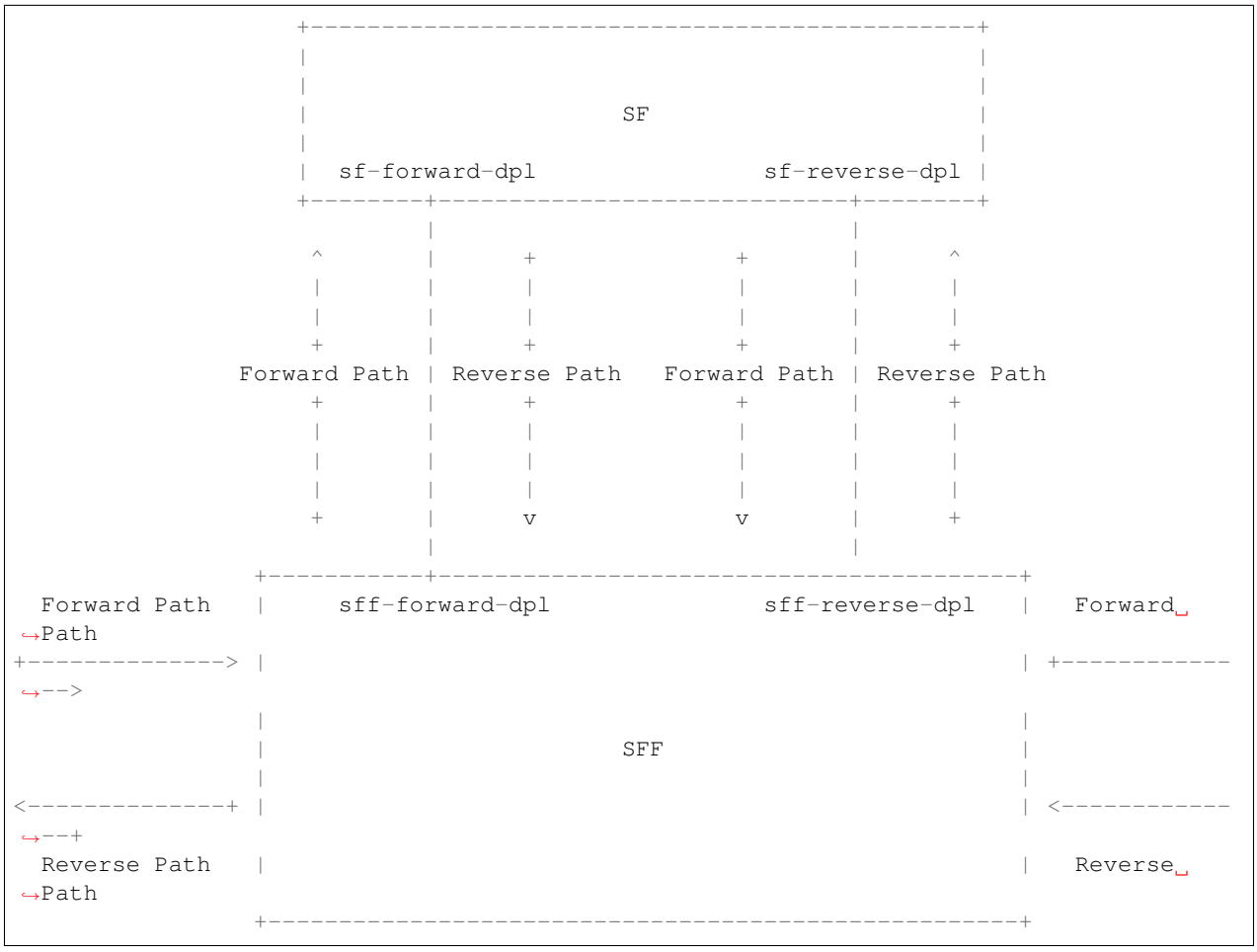
2.13 Directional data plane locators for symmetric paths

2.13.1 Overview

A symmetric path results from a Service Function Path with the symmetric field set or when any of the constituent Service Functions is set as bidirectional. Such a path is defined by two Rendered Service Paths where one of them steers the traffic through the same Service Functions as the other but in opposite order. These two Rendered Service Paths are also said to be symmetric to each other and gives to each path a sense of direction: The Rendered Service Path that corresponds to the same order of Service Functions as that defined on the Service Function Chain is tagged

as the forward or up-link path, while the Rendered Service Path that corresponds to the opposite order is tagged as reverse or down-link path.

Directional data plane locators allow the use of different interfaces or interface details between the Service Function Forwarder and the Service Function in relation with the direction of the path for which they are being used. This function is relevant for Service Functions that would have no other way of discerning the direction of the traffic, like for example legacy bump-in-the-wire network devices.



As shown in the previous figure, the forward path egress from the Service Function Forwarder towards the Service Function is defined by the sff-forward-dpl and sf-forward-dpl data plane locators. The forward path ingress from the Service Function to the Service Function Forwarder is defined by the sf-reverse-dpl and sff-reverse-dpl data plane locators. For the reverse path, it's the opposite: the sff-reverse-dpl and sf-reverse-dpl define the egress from the Service Function Forwarder to the Service Function, and the sf-forward-dpl and sff-forward-dpl define the ingress into the Service Function Forwarder from the Service Function.

Note: Directional data plane locators are only supported in combination with the SFC OF Renderer at this time.

2.13.2 Configuration

Directional data plane locators are configured within the service-function-forwarder in the service-function-dictionary entity, which describes the association between a Service Function Forwarder and Service Functions:

Listing 1: service-function-forwarder.yang

```
    list service-function-dictionary {
        key "name";
        leaf name {
          type sfc-common:sf-name;
          description
              "The name of the service function.";
        }
        container sff-sf-data-plane-locator {
          description
            "SFF and SF data plane locators to use when sending
             packets from this SFF to the associated SF";
          leaf sf-dpl-name {
            type sfc-common:sf-data-plane-locator-name;
            description
              "The SF data plane locator to use when sending
               packets to the associated service function.
               Used both as forward and reverse locators for
               paths of a symmetric chain.";
          }
          leaf sff-dpl-name {
            type sfc-common:sff-data-plane-locator-name;
            description
              "The SFF data plane locator to use when sending
               packets to the associated service function.
               Used both as forward and reverse locators for
               paths of a symmetric chain.";
          }
          leaf sf-forward-dpl-name {
            type sfc-common:sf-data-plane-locator-name;
            description
              "The SF data plane locator to use when sending
               packets to the associated service function
               on the forward path of a symmetric chain";
          }
          leaf sf-reverse-dpl-name {
            type sfc-common:sf-data-plane-locator-name;
            description
              "The SF data plane locator to use when sending
               packets to the associated service function
               on the reverse path of a symmetric chain";
          }
          leaf sff-forward-dpl-name {
            type sfc-common:sff-data-plane-locator-name;
            description
              "The SFF data plane locator to use when sending
               packets to the associated service function
               on the forward path of a symmetric chain.";
          }
          leaf sff-reverse-dpl-name {
            type sfc-common:sff-data-plane-locator-name;
            description
```

(continues on next page)

(continued from previous page)

```
                "The SFF data plane locator to use when sending
                 packets to the associated service function
                 on the reverse path of a symmetric chain.";
            }
          }
      }
```

2.13.3 Example

The following configuration example is based on the Logical SFF configuration one. Only the Service Function and Service Function Forwarder configuration changes with respect to that example:

```
curl -i -H "Content-Type: application/json" -H "Cache-Control: no-cache"
--data '${JSON}' -X PUT --user
admin:admin http://localhost:8181/restconf/config/restconf/config/service-
↪function:service-functions/
```

Service Functions JSON.

```
{
"service-functions": {
    "service-function": [
        {
            "name": "firewall-1",
            "type": "firewall",
            "sf-data-plane-locator": [
                {
                    "name": "sf-firewall-net-A-dpl",
                    "interface-name": "eccb57ae-5a2e-467f-823e-45d7bb2a6a9a",
                    "transport": "service-locator:mac",
                    "service-function-forwarder": "sfflogical1"

                },
                {
                    "name": "sf-firewall-net-B-dpl",
                    "interface-name": "7764b6f1-a5cd-46be-9201-78f917ddee1d",
                    "transport": "service-locator:mac",
                    "service-function-forwarder": "sfflogical1"

                }
            ]
        },
        {
            "name": "dpi-1",
            "type": "dpi",
            "sf-data-plane-locator": [
                {
                    "name": "sf-dpi-net-A-dpl",
                    "interface-name": "df15ac52-e8ef-4e9a-8340-ae0738aba0c0",
                    "transport": "service-locator:mac",
                    "service-function-forwarder": "sfflogical1"
                },
                {
                    "name": "sf-dpi-net-B-dpl",
                    "interface-name": "1bb09b01-422d-4ccf-8d7a-9ebf00d1a1a5",
```

(continues on next page)

(continued from previous page)

```
                    "transport": "service-locator:mac",
                    "service-function-forwarder": "sfflogical1"
                }
            ]
        }
    ]
}
}
```

```
curl -i -H "Content-Type: application/json" -H "Cache-Control: no-cache"
--data '${JSON}' -X PUT --user
admin:admin http://localhost:8181/restconf/config/service-function-forwarder:service-
↪function-forwarders/
```

Service Function Forwarders JSON.

```
{
"service-function-forwarders": {
    "service-function-forwarder": [
        {
            "name": "sfflogical1"
            "sff-data-plane-locator": [
                {
                    "name": "sff-firewall-net-A-dpl",
                    "data-plane-locator": {
                        "interface-name": "eccb57ae-5a2e-467f-823e-45d7bb2a6a9a",
                        "transport": "service-locator:mac"
                    }
                },
                {
                    "name": "sff-firewall-net-B-dpl",
                    "data-plane-locator": {
                        "interface-name": "7764b6f1-a5cd-46be-9201-78f917ddee1d",
                        "transport": "service-locator:mac"
                    }
                },
                {
                    "name": "sff-dpi-net-A-dpl",
                    "data-plane-locator": {
                        "interface-name": "df15ac52-e8ef-4e9a-8340-ae0738aba0c0",
                        "transport": "service-locator:mac"
                    }
                },
                {
                    "name": "sff-dpi-net-B-dpl",
                    "data-plane-locator": {
                        "interface-name": "1bb09b01-422d-4ccf-8d7a-9ebf00d1a1a5",
                        "transport": "service-locator:mac"
                    }
                }
            ],
            "service-function-dictionary": [
                {
                    "name": "firewall-1",
                    "sff-sf-data-plane-locator": {
                        "sf-forward-dpl-name": "sf-firewall-net-A-dpl",
```

(continues on next page)

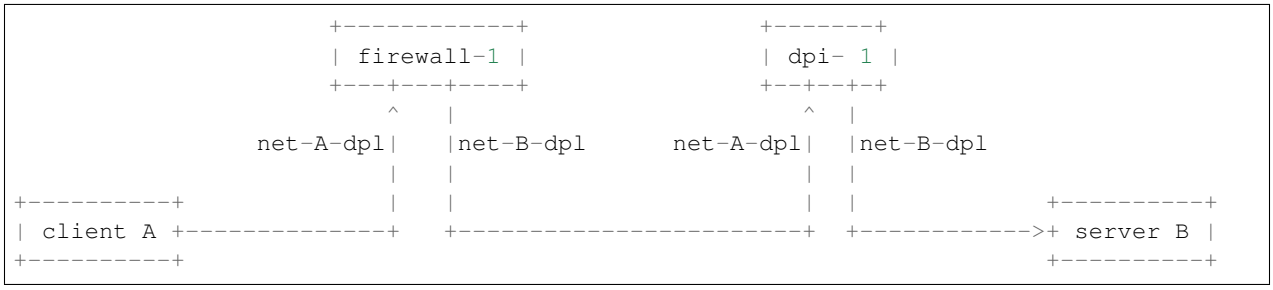
(continued from previous page)

```
                        "sf-reverse-dpl-name": "sf-firewall-net-B-dpl",
                        "sff-forward-dpl-name": "sff-firewall-net-A-dpl",
                        "sff-reverse-dpl-name": "sff-firewall-net-B-dpl",
                    }
                },
                {
                    "name": "dpi-1",
                    "sff-sf-data-plane-locator": {
                        "sf-forward-dpl-name": "sf-dpi-net-A-dpl",
                        "sf-reverse-dpl-name": "sf-dpi-net-B-dpl",
                        "sff-forward-dpl-name": "sff-dpi-net-A-dpl",
                        "sff-reverse-dpl-name": "sff-dpi-net-B-dpl",
                    }
                }
            ]
        }
    ]
}
}
```

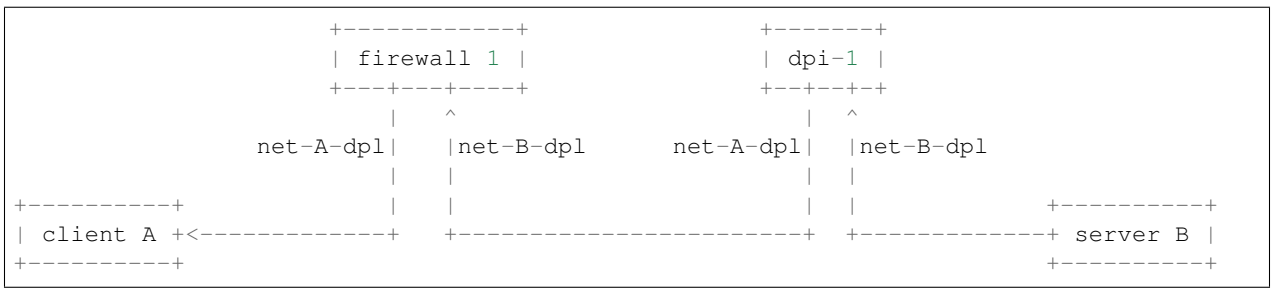
In comparison with the Logical SFF example, noticed that each Service Function is configured with two data plane locators instead of one so that each can be used in different directions of the path. To specify which locator is used on which direction, the Service Function Forwarder configuration is also more extensive compared to the previous example.

When comparing this example with the Logical SFF one, that the Service Function Forwarder is configured with data plane locators and that they hold the same interface name values as the corresponding Service Function interfaces. This is because in the Logical SFF particular case, a single logical interface fully describes an attachment of a Service Function Forwarder to a Service Function on both the Service Function and Service Function Forwarder sides. For non-Logical SFF scenarios, it would be expected for the data plane locators to have different values as we have seen on other examples through out this user guide. For example, if mac addresses are to be specified in the locators, the Service Function would have a different mac address than the Service Function Forwarder.

As a result of the overall configuration, two Rendered Service Paths are implemented. The forward path:



And the reverse path:



Consider the following notes to put the example in context:

- The classification function is obviated from the illustration.
- The forward path is up-link traffic from a client in network A to a server in network B.
- The reverse path is down-link traffic from a server in network B to a client in network A.
- The service functions might be legacy bump-in-the-wire network devices that need to use different interfaces for each network.

2.14 SFC Statistics User Guide

Statistics can be queried for Rendered Service Paths created on OVS bridges. Future support will be added for Service Function Forwarders and Service Functions. Future support will also be added for VPP and IOs-XE devices.

To use SFC statistics the 'odl-sfc-statistics' Karaf feature needs to be installed.

Statistics are queried by sending an RPC RESTconf message to ODL. For RSPs, its possible to either query statistics for one individual RSP or for all RSPs, as follows:

Querying statistics for a specific RSP:

```
curl -i -H "Content-Type: application/json" -H "Cache-Control: no-cache"
--data '{ "input": { "name" : "path1-Path-42" } }' -X POST --user admin:admin
http://localhost:8181/restconf/operations/sfc-statistics-operations:get-rsp-statistics
```

Querying statistics for all RSPs:

```
curl -i -H "Content-Type: application/json" -H "Cache-Control: no-cache"
--data '{ "input": { } }' -X POST --user admin:admin
http://localhost:8181/restconf/operations/sfc-statistics-operations:get-rsp-statistics
```

The following is the sort of output that can be expected for each RSP.

```
{
    "output": {
        "statistics": [
            {
                "name": "sfc-path-1sf1sff-Path-34",
                "statistic-by-timestamp": [
                    {
                        "service-statistic": {
                            "bytes-in": 0,
                            "bytes-out": 0,
                            "packets-in": 0,
                            "packets-out": 0
                        },
                        "timestamp": 1518561500480
                    }
                ]
            }
        ]
    }
}
```

CHAPTER

THREE

SFC DESIGN SPECIFICATIONS

Starting from Nitrogen, SFC uses RST format Design Specification document for all new features. These specifications are perfect way to understand various SFC features.

Contents:

Table of Contents

3.1 Title of the feature

[gerrit filter: https://git.opendaylight.org/gerrit/#/q/topic:cool-topic]

Brief introduction of the feature.

3.1.1 Problem description

Detailed description of the problem being solved by this feature

Use Cases

Use cases addressed by this feature.

3.1.2 Proposed change

Details of the proposed change.

Pipeline changes

Any changes to pipeline must be captured explicitly in this section.

Yang changes

This should detail any changes to yang models.

Listing 1: example.yang

```
module example {
    namespace "urn:opendaylight:sfc:example";
    prefix "example";

    import ietf-yang-types {prefix yang; revision-date "2013-07-15";}

    description "An example YANG model.";

    revision 2017-02-14 { description "Initial revision"; }
}
```

Configuration impact

Any configuration parameters being added/deprecated for this feature? What will be defaults for these? How will it impact existing deployments?

Note that outright deletion/modification of existing configuration is not allowed due to backward compatibility. They can only be deprecated and deleted in later release(s).

Clustering considerations

This should capture how clustering will be supported. This can include but not limited to use of CDTCL, EOS, Cluster Singleton etc.

Other Infra considerations

This should capture impact from/to different infra components like MDSAL Datastore, karaf, AAA etc.

Security considerations

Document any security related issues impacted by this feature.

Scale and Performance Impact

What are the potential scale and performance impacts of this change? Does it help improve scale and performance or make it worse?

Targeted Release

What release is this feature targeted for?

Alternatives

Alternatives considered and why they were not selected.

3.1.3 Usage

How will end user use this feature? Primary focus here is how this feature will be used in an actual deployment.

This section will be primary input for Test and Documentation teams. Along with above this should also capture REST API and CLI.

Features to Install

odl-sfc-openflow-renderer

Identify existing karaf feature to which this change applies and/or new karaf features being introduced. These can be user facing features which are added to integration/distribution or internal features to be used by other projects.

REST API

Sample JSONS/URIs. These will be an offshoot of yang changes. Capture these for User Guide, CSIT, etc.

CLI

Any CLI if being added.

3.1.4 Implementation

Assignee(s)

Who is implementing this feature? In case of multiple authors, designate a primary assignee and other contributors.

Primary assignee: <developer-a>, <irc nick>, <email>

Other contributors: <developer-b>, <irc nick>, <email> <developer-c>, <irc nick>, <email>

Work Items

Break up work into individual items. This should be a checklist on a Trello card for this feature. Provide the link to the trello card or duplicate it.

3.1.5 Dependencies

Any dependencies being added/removed? Dependencies here refers to internal [other ODL projects] as well as external [OVS, karaf, JDK etc]. This should also capture specific versions if any of these dependencies. e.g. OVS version, Linux kernel version, JDK etc.

This should also capture impacts on existing projects that depend on SFC.

Following projects currently depend on SFC: GBP Netvirt

3.1.6 Testing

Capture details of testing that will need to be added.

Unit Tests

Integration Tests

CSIT

3.1.7 Documentation Impact

What is the impact on documentation for this change? If documentation changes are needed call out one of the <contributors> who will work with the Project Documentation Lead to get the changes done.

Don't repeat details already discussed but do reference and call them out.

3.1.8 References

Add any useful references. Some examples:

- Links to Summit presentation, discussion etc.
- Links to mail list discussions
- Links to patches in other projects
- Links to external documentation

[1] OpenDaylight Documentation Guide

[2] https://specs.openstack.org/openstack/nova-specs/specs/kilo/template.html

---

Note: This template was derived from [2], and has been modified to support our project.

Table of Contents

3.2 Title of the feature

[gerrit filter: https://git.opendaylight.org/gerrit/#/q/topic:rsp-config]

Currently Rendered Service Paths (RSPs) are created directly in the Operational Data Store via an RPC operation. This Spec details the refactoring involved to create RSPs first in the Config data store, and then to the Operational data store.

3.2.1 Problem description

Since the Rendered Service Paths (RSPs) are currently only written to the Operational Data Store, the RSPs will not be present in the data store when ODL restarts.

With the refactoring proposed in this spec, SFP creation will trigger RSP creation, which will cause a write to the RSP Configuration Data Store with information that needs to be persisted across ODL restarts. The RSP Configuration Data Store creation will trigger RSP Operational Data Store creation. To avoid confusion, updates to the RSP Configuration Data Store will be rejected by a newly created RSP MD-SAL Validator.

Upon ODL restart, the RSPs will be written to Operational Data Store based on what is already present in the RSP Configuration Data Store. This change will make RSP creation be inline with traditional methods of creating entries using the Config/Operational data store methodology.

Use Cases

- RSP data integrity upon ODL restart.

3.2.2 Proposed change

Pipeline changes

The existing OpenFlow pipeline will not be affected by this change.

Yang changes

All yang models related to the current RSP creation via RPC will be deprecated in Oxygen and removed in Fluorine.

It will now be possible to create the RSP in the configuration data store. The actual RSP data model contents will not change otherwise.

The following RSP data model is the updated data model, with comments highlighting the changed nodes. The changed nodes will only have the "config false" attribute modified to reflect which nodes will only be written in the Operational data store.

Listing 2: rendered-service-path.yang

```
module rendered-service-path {
   container rendered-service-paths {
      // UPDATED This container is no longer "config false"
      description
        "A container that holds the list of all Rendered Service Paths
         in a SFC domain";
      list rendered-service-path {
        key "name";
        description
          "A list that holds operational data for all RSPs in the
           domain";
        leaf name {
          type sfc-common:rsp-name;
          description
            "The name of this rendered function path. This is the same
             name as the associated SFP";
        }
        leaf parent-service-function-path {
          type sfc-common:sfp-name;
          description
            "Service Function Path from which this RSP was
             instantiated";
        }
        leaf transport-type {
          // UPDATED This node is now "config false"
          config false;

          type sfc-sl:sl-transport-type-def;
          default "sfc-sl:vxlan-gpe";
          description
            "Transport type as set in the Parent Service Function
             Path";
```

(continues on next page)

(continued from previous page)

```
        }
        leaf context-metadata {
          // UPDATED This node is now "config false"
          config false;

          type sfc-md:context-metadata-ref;
          description
            "The name of the associated context metadata";
        }
        leaf variable-metadata {
          // UPDATED This node is now "config false"
          config false;

          type sfc-md:variable-metadata-ref;
          description
            "The name of the associated variable metadata";
        }
        leaf tenant-id {
          type string;
          description
            "This RSP was created for a specific tenant-id";
        }
        uses sfc-ss:service-statistics-group {
          // UPDATED This node is now "config false"
          config false;

          description "Global Rendered Service Path statistics";
        }
        list rendered-service-path-hop {
          key "hop-number";
          leaf hop-number {
            type uint8;
            description
              "A Monotonically increasing number";
          }
          leaf service-function-name {
            type sfc-common:sf-name;
            description
              "Service Function name";
          }
          leaf service-function-group-name {
            type string;
            description
              "Service Function group name";
          }
          leaf service-function-forwarder {
            type sfc-common:sff-name;
            description
              "Service Function Forwarder name";
          }
          leaf service-function-forwarder-locator {
            type sfc-common:sff-data-plane-locator-name;
            description
              "The name of the SFF data plane locator";
          }
          leaf service-index {
            type uint8;
```

(continues on next page)

(continued from previous page)

```
            description
              "Provides location within the service path.
               Service index MUST be decremented by service functions
               or proxy nodes after performing required services.  MAY
               be used in conjunction with service path for path
               selection.  Service Index is also valuable when
               troubleshooting/reporting service paths.  In addition to
               location within a path, SI can be used for loop
               detection.";
          }
          ordered-by user;
          description
            "A list of service functions that compose the
             service path";
        }
        leaf service-chain-name {
          // UPDATED This node is now "config false"
          config false;

          type sfc-common:sfc-name;
          mandatory true;
          description
            "The Service Function Chain used as blueprint for this
             path";
        }
        leaf starting-index {
          // UPDATED This node is now "config false"
          config false;

          type uint8;
          description
            "Starting service index";
        }
        leaf path-id {
          type uint32 {
            range "0..16777216";
          }
          mandatory true;
          description
            "Identifies a service path.
             Participating nodes MUST use this identifier for path
             selection.  An administrator can use the service path
             value for reporting and troubleshooting packets along
             a specific path.";
        }
        leaf symmetric-path-id {
          type uint32 {
            range "0..16777216";
          }
          description
            "Identifies the associated symmetric path, if any.";
        }
        leaf sfc-encapsulation {
          // UPDATED This node is now "config false"
          config false;

          type sfc-sl:sfc-encapsulation-type;
```

(continues on next page)

(continued from previous page)

```
            description
              "The type of encapsulation used in this path for passing
              SFC information along the chain";
          }
        }
    }
}
```

Configuration impact

All yang models related to the current RSP creation via RPC will be deprecated in Oxygen and removed in Fluorine. It will now be possible to create the RSP in the configuration data store. The "config false" flag will be removed from the RSP data model, thus allowing it to be created in the Config data store.

Although the RSP creation via RPC will be deprecated in the Oxygen release, it will still be supported until Fluorine. Once this change is implemented, the preferred way of creating RSPs will be via a write to the Config Data Store.

Clustering considerations

Currently RSPs support clustering, which will not be affected by this change.

Other Infra considerations

None

Security considerations

None

Scale and Performance Impact

With this change, there will be an additional write to the data store for each RSP creation. Considering there shouldnt be many RSPs created (typically less than 100) the impacts should be negligible.

Targeted Release

This feature is targeted for the Oxygen release.

Alternatives

None

3.2.3 Usage

Features to Install

All changes will be in the following existing Karaf features:

- odl-sfc-model
- odl-sfc-provider

REST API

The following JSON shows how an SFP is created, which will trigger the creation of the RSPs in both the Configuration and Operational data store.

```
URL: http://localhost:8181/config/service-function-path:rendered-service-paths/

{
  "service-function-paths": {
    "service-function-path": [
      {
        "name": "sfp1",
        "service-chain-name": "sfc1",
        "transport-type": "service-locator:vxlan-gpe",
        "symmetric": true
      }
    ]
  }
}
```

CLI

A new Karaf Shell command will be added to list the RSPs.

3.2.4 Implementation

Assignee(s)

Primary assignee:

- Brady Johnson, #ebrjohn, bradyallenjohnson@gmail.com

Other contributors:

- David Suárez, #edavsua, david.suarez.fuentes@gmail.com

Work Items

- Deprecate existing RSP RPC creation yang models.
- Deprecate existing RSP RPC Java classes and/or methods.
- Modify existing RSP data model "config false" values:
  - The entire RSP data model should no longer be "config false".
  - Mark those RSP data model leaf nodes as "config false" that will only be in operational.
- Create SFP configuration data store listener.
- Create RSP configuration data store listener.
- Create code to reject updates to the RSP configuration data store.
- Copy and retrofit existing code that writes RSPs to operational via RPCs to do so via the RSP configuration listener instead of via RPC.
- Create Karaf Shell CLI command to list RSPs in the config and operational data stores.

3.2.5 Dependencies

The following projects currently depend on SFC, and will be affected by this change:

- GBP
- Netvirt

3.2.6 Testing

Unit Tests

- The RSP creation in the existing UT will need to be updated as a result of this change.
- UT will need to be added to test RSP creation.

Integration Tests

None

CSIT

The RSP creation in the existing CSIT tests will need to be updated as a result of this change.

3.2.7 Documentation Impact

Both the User Guide and Developer Guide will need to be updated by the current ODL SFC Documentation contact: David Suárez.

3.2.8 References

[1] OpenDaylight Documentation Guide

Table of Contents

3.3 RSP Statistics

[gerrit filter: https://git.opendaylight.org/gerrit/#/q/topic:topic/rsp_stats]

This feature introduces functionality to provide statistics for Rendered Service Paths (RSPs) defined with the SFC OpenFlow renderer.

3.3.1 Problem description

Currently there is no way to easily determine how many packets and bytes have traversed the different RSPs created with SFC. Its possible to obtain this information by looking at the SFC OpenFlow flows, but this would require a detailed knowledge of the flows.

Use Cases

Provide input and output bytes and packets for each RSP.

3.3.2 Proposed change

The RSP statistics will be collected by retrieving the OpenFlow statistics counters from certain OpenFlow flow entries created by SFC. The RSP statistics will be the following:

- RSP bytes in
- RSP bytes out
- RSP packets in
- RSP packets out

The RSP bytes/packets out can only be less than or equal to the bytes/packets in. The RSP bytes/packets out will be less than the bytes/packets in if the packets are dropped by either the SF or SFF.

The RSP statistics will be collected by retrieving the SFC OpenFlow NextHop flow entry from the first-hop SFF in the RSP. The RSP out statistics will be collected by retrieving the SFC OpenFlow NextHop flow entry from the last-hop SFF in the RSP. The flows will be retrieved based on their flow name.

RSP statistics collection will be provided on-demand by issuing a Northbound RPC RESTconf command, specifying the RSP to query. It will also be possible to query the statistics for all RSPs using the same RPC call. If the RSP specified in the RPC does not exist, then an RPC error message will be returned. Otherwise the packets and bytes in and out will be returned indicating the traffic that has gone through the RSP since its been created.

Using this on-demand approach, if time-series RSP statistics is needed, it can be handled externally to ODL, with a proper database since the MD-SAL is not an appropriate place to store time-series data.

A new SFC Karaf shell command will be created allowing RSP statistics to be retrieved via the Karaf shell.

Pipeline changes

This change will not require the SFC OpenFlow pipeline to be changed.

Yang changes

Although this feature only covers RSP statistics, the following YANG model also includes operations for Service Function (SF) and Service Function Forwarder (SFF) statistics collection.

```
module sfc-statistics-operations {

  namespace "urn:inocybe:params:xml:ns:yang:sfc-stats-ops";
  prefix sfc-stats-ops;

  import service-statistics {
      prefix sfc-ss;
      revision-date 2014-07-01;
  }

  organization "Inocybe, Inc.";
  contact "Brady Johnson <bjohnson@inocybe.com>";

  description
    "This module contains RPC operations to collect SFC statistics";

  revision 2017-12-15 {
    description
      "Initial Revision";
  }

  rpc get-rsp-statistics {
    description
      "Requests statistics for the specified Rendered Service Path";
    input {
      leaf name {
        type string;
        description
          "The name of the Rendered Service Path. Leaving the
           name empty will return statistics for all Rendered
           Service Paths.";
      }
    }
    output {
      list statistics {
        leaf name {
          type string;
          description
            "The name of the Rendered Service Path.";
        }
        uses sfc-ss:service-statistics-group {
          description "Rendered Service Path statistics";
        }
      }
    }
  }

  rpc get-sff-statistics {
```

(continues on next page)

(continued from previous page)

```
    description
      "Requests statistics for the specified Service Function Forwarder";
    input {
      leaf name {
        type string;
        description
          "The name of the Service Function Forwarder. Leaving
           the name empty will return statistics for all Service
           Function Forwarders.";
      }
    }
    output {
      list statistics {
        leaf name {
          type string;
          description
            "The name of the Service Function Forwarder.";
        }
        uses sfc-ss:service-statistics-group {
          description "Service Function Forwarder statistics";
        }
      }
    }
  }

  rpc get-sf-statistics {
    description
      "Requests statistics for the specified Service Function";
    input {
      leaf name {
        type string;
        description
          "The name of the Service Function. Leaving the
           name empty will return statistics for all
           Service Functions.";
      }
    }
    output {
      list statistics {
        leaf name {
          type string;
          description
            "The name of the Service Function.";
        }
        uses sfc-ss:service-statistics-group {
          description "Service Function statistics";
        }
      }
    }
  }

}
```

Configuration impact

There will be no configuration impacts as a result of this feature.

Clustering considerations

The RSP statistics feature will not affect clustering, and will work with no problems in an ODL cluster

Other Infra considerations

N/A

Security considerations

N/A

Scale and Performance Impact

Since this will be an on-demand statistics request, there will be no scale and performance impacts.

Targeted Release

This feature is targeted to be implemented in the Oxygen release.

Alternatives

N/A

3.3.3 Usage

Nothing special needs to be done to use this feature, as it will be an on-demand request via the Northbound RPC RESTConf.

Features to Install

A new Karaf feature will be created called odl-sfc-statistics. No other existing SFC Karaf features will depend on this new feature.

REST API

The following example shows the new SFC statistics RPC definitions:

```
URL: http://localhost:8181/operations/sfc-statistics-operations:get-rsp-statistics

{
  "input": {
    "name": "RSP-1sf1sff"
```

(continues on next page)

(continued from previous page)

```
  }
}

{
  "output": {
    "statistics" : [
      {
        "name": "RSP-1sf1sff",
        "statistic-by-timestamp": [
          {
            "service-statistic": {
              "bytes-in": 0,
              "bytes-out": 0,
              "packets-in": 0,
              "packets-out": 0
            },
            "timestamp": 1512418230327
          }
        ]
      }
    ]
  }
}
```

CLI

A new Karaf CLI will be added to retrieve RSP statistics. The syntax will similar to the following. Leaving the RSP name empty will return the statistics for all RSPs.

- sfc:rsp-statistics [RSP-name]

3.3.4 Implementation

Assignee(s)

Primary assignee: <Brady Johnson>, <ebrjohn>, <bjohnson@inocybe.com>

Work Items

Break up work into individual items. This should be a checklist on a Trello card for this feature. Provide the link to the trello card or duplicate it.

- Create the SFC statistics collection RPC data model.
- Create an RSP statistics collection handler that will retrieve the relevant OpenFlow flows and return the results.
- Create the necessary utils to assist the RSP handler in getting the flows and storing the results.
- Create the new odl-sfc-statistics Karaf feature.
- Create the Karaf shell command to retrieve the statistics.

3.3.5 Dependencies

No external projects will depend on this new feature. Nor will any additional dependencies on other ODL project be introduced.

3.3.6 Testing

Capture details of testing that will need to be added.

Unit Tests

A new Unit Test will be added for each of the new Java classes added.

Integration Tests

N/A

CSIT

A new test case will be added to CSIT for this feature. The test should inject packets and will verify that the RSP statistics counters are incremented as expected.

3.3.7 Documentation Impact

The User Guide will be updated to show how to use this new feature.

3.3.8 References

N/A

Table of Contents

3.4 Karaf Command Line Interface (CLI) for SFC

[S: https://git.opendaylight.org/gerrit/#/q/topic:sfc-shell]

The Karaf Container offers a very complete Unix-like console that allows managing the container. This console can be extended with custom commands to manage the features deployed on it. This feature will add some basic commands to show the provisioned SFC's entities.

3.4.1 Problem description

This feature will implement commands to show some of the provisioned SFC's entities:

- Service Functions
- Service Function Forwarders
- Service Function Chains
- Service Function Paths
- Service Function Classifiers
- Service Nodes
- Service Function Types

Use Cases

- Use Case 1: list one/all provisioned Service Functions.
- Use Case 2: list one/all provisioned Service Function Forwarders.
- Use Case 3: list one/all provisioned Service Function Chains.
- Use Case 4: list one/all provisioned Service Function Paths.
- Use Case 5: list one/all provisioned Service Function Classifiers.
- Use Case 6: list one/all provisioned Service Nodes.
- Use Case 7: list one/all provisioned Service Function Types.

3.4.2 Proposed change

Details of the proposed change.

Pipeline changes

None

Yang changes

None

Configuration impact

None

Clustering considerations

None

Other Infra considerations

Creation of new commands for the Karaf's console.

Security considerations

None

Scale and Performance Impact

None

Targeted Release

Nitrogen

Alternatives

None

3.4.3 Usage

The feature will add CLI commands to the Karaf's console to list some of the provisioned SFC's entities. See the CLI section for details about the syntax of those commands.

Features to Install

odl-sfc-shell

REST API

None

CLI

- UC 1: list one/all provisioned Service Functions.

  sfc:sf-list [–name <name>]
- UC 2: list one/all provisioned Service Function Forwarders.

  sfc:sff-list [–name <name>]
- UC 3: list one/all provisioned Service Function Chains.

  sfc:sfc-list [–name <name>]
- UC 4: list one/all provisioned Service Function Paths.

  sfc:sfp-list [–name <name>]
- UC 5: list one/all provisioned Service Function Classifiers.

  sfc:sc-list [–name <name>]
- UC 6: list one/all provisioned Service Nodes.

  sfc:sn-list [–name <name>]
- UC 7: list one/all provisioned Service Function Types.

  sfc:sft-list [–name <name>]

3.4.4 Implementation

Assignee(s)

Primary assignee: David Suárez, #edavsua, david.suarez.fuentes@gmail.com Brady Johson, #ebrjohn, bradyallen-johnson@gmail.com

Work Items

- Implement UC 1: list one/all provisioned Service Functions.
- Implement UC 2: list one/all provisioned Service Function Forwarders.
- Implement UC 3: list one/all provisioned Service Function Chains.
- Implement UC 4: list one/all provisioned Service Function Paths.
- Implement UC 5: list one/all provisioned Service Function Classifiers.
- Implement UC 6: list one/all provisioned Service Nodes.
- Implement UC 7: list one/all provisioned Service Types.

3.4.5 Dependencies

This feature uses the new Karaf 4.x API to create CLI commands.

No changes needed on projects depending on SFC.

3.4.6 Testing

Capture details of testing that will need to be added.

Unit Tests

Integration Tests

CSIT

None

3.4.7 Documentation Impact

The new CLI for SFC will be documented in both the User and Developer guides.

3.4.8 References

Add any useful references. Some examples:

- https://docs.google.com/presentation/d/1RKkJsTUF65t40ASXVztNMcKAxMzI_owyZ-c6Mpm4Ss8/edit?usp=sharing

[1] OpenDaylight Documentation Guide

Table of Contents

3.5 Directional data plane locators

[gerrit filter: https://git.opendaylight.org/gerrit/#/q/topic:directional-dpl]

This specification proposes to have an optional direction associated with data plane locators.

3.5.1 Problem description

Service functions that are not SFC encapsulation aware need to use alternative mechanisms to recognize and apply SFC features. One common alternative is to use multiple interfaces. For example, each interface could be associated exclusively to the ingress traffic of a specific service function path.

Service functions for which this concept will most commonly apply are those network devices that operate as 'bump in the wire', where two interfaces acting as ingress and egress interfaces for traffic in one direction, would be the egress and ingress interfaces respectively in the opposite direction.

Currently, the connection between a service function and a service function forwarder is described by single pair of data plane locators, one for each side of the connection. For symmetric service chains, the same locator pair is used when rendering the forward and reverse paths. This prevents the possibility to apply the mechanism explained previously.

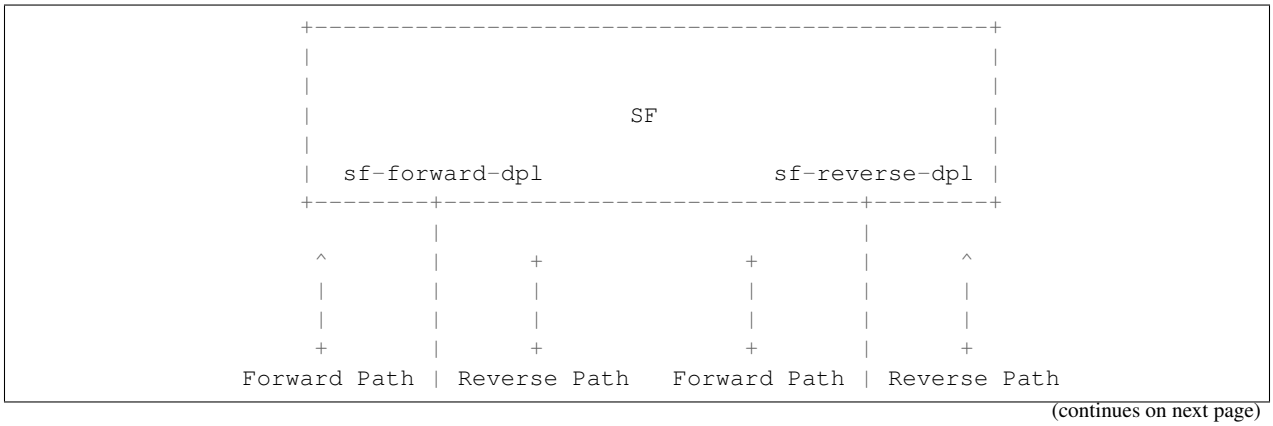
The purpose of this specification is to overcome this first limitation, enabling the support of bump in the wire service functions that are sfc aware, understood as service functions that allow to employ inline mechanisms to steer traffic (mac chaining, nsh, vlan, mpls...), versus bump in the wire service functions that are sfc unaware and thus might not allow any modification over the source traffic.

Use Cases

Support 'bump in the wire' network devices that can be made sfc aware to act as service functions.

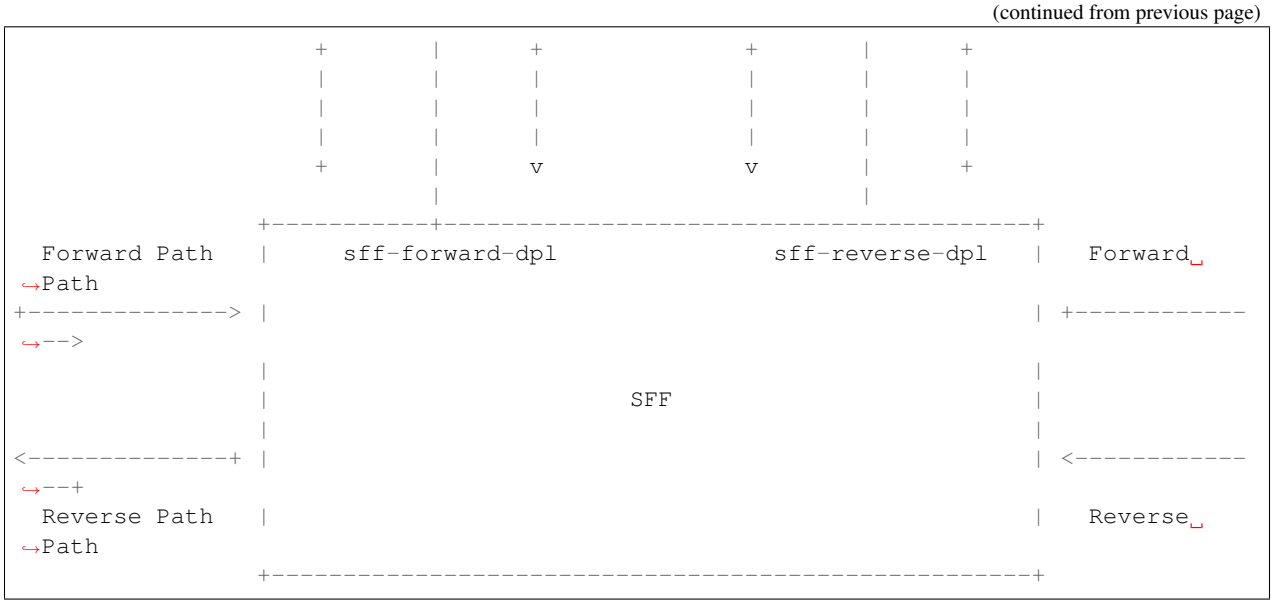
3.5.2 Proposed change

The proposed solution is to add the flexibility to configure two pairs of data plane locators to describe the association between the service function and service function forwarder. Each pair will be assigned a direction, either forward or reverse, and will be used as ingress for paths on the same direction and as egress from paths on the opposite direction. The forward pair will be used to ingress traffic to the service function on forward paths and for the egress traffic from the service function on reverse paths. The reverse pair will be used for the egress traffic from the service function for the forward path and to ingress the traffic to the service function for the reverse path. This can seen in the following figure:



(continues on next page)

(continued from previous page)



The SFF-SF dictionary will be expanded to accommodate two pairs of locators as depicted in *Yang changes*. Renderers that support this feature shall give precedence to the new directional locators over the old locators. Once all renderers support the feature, the single locator pair may be deprecated or may remain as a simple way to configure a bidirectional locator.

Logical SFF configuration model needs to change as it does not currently use the SFF-SF dictionary. Logical SFF configuration model uses a placeholder service function forwarder configuration. The proposal is to align the logical SFF configuration model with a standard SFC configuration, otherwise two different models will need to be continuously proposed for every other feature, causing more divergence over time. As depicted in *REST API*, logical interfaces shall be configured as locators both on the SF and the SFF side.

The implementation targets the openflow renderer. The openflow processor will provide to the transport specific processor the appropriate data plane locator pair based on the direction of the path being rendered.

Also in Logical SFF context, service binding will be performed on the service function forwarder logical interfaces as soon as intervenes on a path. On egress towards the service function, egress actions will be requested from genius for the interface provided by the openflow processor.

It is worth mentioning that the proposed change may not be enough to fully support legacy 'bump in the wire' network devices that are sfc unaware acting as service functions. For this, it might be additionally needed to:

- Provide the service function with the original unmodified source traffic.
- And as a consequence, on service function egress, reclassify the traffic to a path based on the service function forwarder ingress port.
- And as a consequence, avoid using that port as ingress for more than one path.

Directional data plane locators is a step towards 'bump in the wire' full support and useful in itself for those service functions that while operating in this mode, are sfc aware in that they allow to use already supported mechanisms (mac chaining, nsh...) to steer SFC traffic.

Pipeline changes

The existing OpenFlow pipeline will not be affected by this change.

Yang changes

The following data model is the updated service-function-dictionary within the service function forwarder.

Listing 3: service-function-forwarder.yang

```
    list service-function-dictionary {
        key "name";
        leaf name {
          type sfc-common:sf-name;
          description
              "The name of the service function.";
        }
        container sff-sf-data-plane-locator {
          description
            "SFF and SF data plane locators to use when sending
             packets from this SFF to the associated SF";
          leaf sf-dpl-name {
            type sfc-common:sf-data-plane-locator-name;
            description
              "The SF data plane locator to use when sending
               packets to the associated service function.
               Used both as forward and reverse locators for
               paths of a symmetric chain.";
          }
          leaf sff-dpl-name {
            type sfc-common:sff-data-plane-locator-name;
            description
              "The SFF data plane locator to use when sending
               packets to the associated service function.
               Used both as forward and reverse locators for
               paths of a symmetric chain.";
          }
          leaf sf-forward-dpl-name {
            type sfc-common:sf-data-plane-locator-name;
            description
              "The SF data plane locator to use when sending
               packets to the associated service function
               on the forward path of a symmetric chain";
          }
          leaf sf-reverse-dpl-name {
            type sfc-common:sf-data-plane-locator-name;
            description
              "The SF data plane locator to use when sending
               packets to the associated service function
               on the reverse path of a symmetric chain";
          }
          leaf sff-forward-dpl-name {
            type sfc-common:sff-data-plane-locator-name;
            description
              "The SFF data plane locator to use when sending
               packets to the associated service function
               on the forward path of a symmetric chain.";
```

(continues on next page)

(continued from previous page)

```
          }
          leaf sff-reverse-dpl-name {
            type sfc-common:sff-data-plane-locator-name;
            description
              "The SFF data plane locator to use when sending
               packets to the associated service function
               on the reverse path of a symmetric chain.";
          }
        }
    }
```

Logical interface locator support is also added to the service function forwarder data plane locator.

Listing 4: service-function-forwarder-logical.yang

```
    augment "/sfc-sff:service-function-forwarders/"
          + "sfc-sff:service-function-forwarder/"
          + "sfc-sff:sff-data-plane-locator/"
          + "sfc-sff:data-plane-locator/"
          + "sfc-sff:locator-type/" {
      description "Augments the Service Function Forwarder to allow the use of␣
↪logical
                   interface locators";
      case logical-interface {
        uses logical-interface-locator;
      }
    }
```

A new leaf is added to the rendered service path model to flag reverse paths.

Listing 5: rendered-service-path.yang

```
    leaf reverse-path {
      type boolean;
      mandatory true;
      description
        "True if this path is the reverse path of a symmetric
         chain.";
    }
```

Configuration impact

New optional parameters are added to the SFF-SF dictionary. These new parameters may not be configured in which case behavior is not changed.

The new flag introduced in the rendered service path model does not have configuration impact as the entity is not meant to be configured.

Logical SFF configuration model will change. Both, previous and new configuration models will be supported.

Thus backward compatibility is preserved despite the introduced changes.

Clustering considerations

Clustering support will not be affected by this change.

Other Infra considerations

None.

Security considerations

None.

Scale and Performance Impact

None.

Targeted Release

This feature is targeted for the Oxygen release.

Alternatives

One first consideration is that if one SF interface required two data plane locators, two SF interfaces is going to require four data plane locators to be fully described, specially considering a scenario where we would like to explicitly configure different openflow ports on the SFF side for each direction. The proposed solution leverages the fact that the data plane locators are already contained in lists on both the SFF and the SF.

One alternative is to introduce a new locator type that serves as an indirection through which the names of forward and reverse locators can be specified. Thus three locators are required total for each side of the SFF-SF association: one forward locator, one reverse locator and the new locator that tells which is which, whose name would be used in the SFF-SF dictionary. The advantage is that the SFF configuration model needs not to be changed. As disadvantages, it needs an extra data plane locator on each side, it might be confusing being able to specify different SFF names and transport types between the three locators on SF side, and finally, the indirection overall leads to a less explicit model/api and code wise it would probably require to hide all locator checks or manipulation behind helper code.

Another option is to expand the key of the SFF-SF dictionary to include direction so that two dictionary entries can be specified for each SFF/SF pair. This was discarded because is not backward compatible.

3.5.3 Usage

Features to Install

All changes will be in the following existing Karaf features:

- odl-sfc-genius
- odl-sfc-openflow-renderer

REST API

The following JSON shows how the service function and service function forwarder are configured in the context of Logical SFF with directional locators.

```
URL: http://localhost:8181/restconf/config/service-function:service-functions/

{
  "service-functions": {
    "service-function": [
      {
        "name": "firewall-1",
        "type": "firewall",
        "sf-data-plane-locator": [
          {
            "name": "firewall-ingress-dpl",
            "interface-name": "eccb57ae-5a2e-467f-823e-45d7bb2a6a9a",
            "transport": "service-locator:mac",
            "service-function-forwarder": "sfflogical1"
          },
          {
            "name": "firewall-egress-dpl",
            "interface-name": "df15ac52-e8ef-4e9a-8340-ae0738aba0c0",
            "transport": "service-locator:mac",
            "service-function-forwarder": "sfflogical1"
          }
        ]
      }
    ]
  }
}
```

```
URL: http://localhost:8181/restconf/config/service-function-forwarder:service-
↪function-forwarders/

{
  "service-function-forwarders": {
    "service-function-forwarder": [
      {
        "name": "sfflogical1",
        "sff-data-plane-locator": [
          {
            "name": "firewall-ingress-dpl",
            "data-plane-locator": {
              "interface-name": "df15ac52-e8ef-4e9a-8340-ae0738aba0c0",
              "transport": "service-locator:mac"
            }
          },
          {
            "name": "firewall-egress-dpl",
            "data-plane-locator": {
              "interface-name": "eccb57ae-5a2e-467f-823e-45d7bb2a6a9a",
              "transport": "service-locator:mac"
            }
          }
        ],
        "service-function-dictionary": [
```

(continues on next page)

(continued from previous page)

```
            {
              "name": "firewall-1",
              "sff-sf-data-plane-locator":
              {
                "sf-forward-dpl-name": "firewall-ingress-dpl",
                "sf-reverse-dpl-name": "firewall-egress-dpl",
                "sff-forward-dpl-name": "firewall-egress-dpl",
                "sff-reverse-dpl-name": "firewall-ingress-dpl",
              }
            }
          ]
        }
      ]
    }
}
```

CLI

No new CLI commands will be added but the existing ones will be enhanced to display more details about the associations between service function and service function forwarders.

3.5.4 Implementation

Assignee(s)

Primary assignee:

- Jaime Caamaño, #jaicaa, jcaamano@suse.com

Work Items

- Update the service function forwarder yang model.
- Update odl-sfc-genius to bind on service function forwarder interfaces.
- Update odl-openflow-renderer processor and surrounding utilities to use the proper data plane locator based on the direction of the path.
- Update provider to set the reverse flag on reverse rendered service paths.
- Update the shell command for service functions and service function forwarders to display the associations between them.
- Update CSIT Full Deploy to use new Logical SFF configuration model.
- Update the user & developer guide to document directional data plane locators.
- Update the user & developer guide to reflect the new Logical SFF configuration model.

3.5.5 Dependencies

The following projects currently depend on SFC:

- GBP
- Netvirt

No backward incompatible changes are introduced but these projects, as neutron implementations, are target users for the new feature in Logical SFF scenario.

3.5.6 Testing

Unit Tests

Unit tests will be added for new code introduced through this feature.

Integration Tests

None.

CSIT

Existing Full Deploy CSIT will be updated to use the new Logical SFF configuration model.

3.5.7 Documentation Impact

Both the User Guide and Developer Guide will need to be updated.

3.5.8 Open Issues

- Drop support for the old Logical SFF configuration model?
- New CSIT tests not proposed yet because it requires testing with traffic, which we don't currently have and is a major undertaking on itself.

3.5.9 References

NA

Table of Contents

3.6 SFC OpenFlow Pipeline

[gerrit filter: https://git.opendaylight.org/gerrit/#/q/topic:nsh-support]

This feature describes the changes needed to the SFC OpenFlow pipeline as a result of migrating the OpenDaylight OpenFlow Plugin to use the OVS 2.9 Network Service Headers (NSH) APIs.

3.6.1 Problem description

NSH has only recently been officially merged into the mainline OVS source code in the 2.8 and 2.9 releases. Previously, a branched version of the OVS 2.6 code was used for the SFC project. The OVS NSH APIs have changed from the branched version of the code to the official OVS 2.9 version of the code, which results in the OpenDaylight OpenFlow plugin NSH APIs needing to change and the way SFC uses the OpenFlow plugin.

The following is a quick summary of the changes when comparing the latest NSH implementation to the old one:

- All NSH fields now have the prerequisite of verifying the packet is a NSH packet. This is achieved either by matching ether_type=0x894f if the outermost header is Ethernet or by matching packet_type=(1,0x894f) if the

outermost header is NSH.

- push_nsh and pop_nsh actions are not available. Instead, encap(nsh) is used to add a NSH header to a packet followed by encap(ethernet) to add an Ethernet header on top of the nsh header. decap() can be used twice to remove both the Ethernet and NSH headers.
- tun_gpe_np field is not available. The corresponding header field is internally managed by OVS.
- encap_eth_type, encap_eth_src and encap_eth_dst fields are no longer available. Standard Ethernet fields will have to be used instead, which will apply to the outermost Ethernet header of the packet.
- nsh_mdtype and nsh_np fields are read only. nsh_mdtype can be set as an argument to the encap NSH action and defaults to 1. nsh_np is set internally by OVS.
- New fields available: nsh_flags and nsh_ttl.
- New action available: dec_nsh_ttl.
- A new OVS feature, packet type aware pipeline, allows packets without an outer Ethernet header to traverse the pipeline. A new tunnel interface option, packet_type=ptap, allows packets without an outer Ethernet header to ingress the pipeline for supporting tunnel types like vxlan-gpe. A new match field packet_type is used for prerequisite matching on these packets without outer Ethernet header.

Use Cases

SFC encapsulation with Network Service Headers (NSH), as described in the NSH RFC: https://tools.ietf.org/html/rfc8300

3.6.2 Proposed change

The details in this section focus on the changes to the SFC OpenFlow pipeline, and will focus primarily on the NSH aspects of the pipeline.

Pipeline changes

Current SFC pipeline

Currently, for a standalone SFC deployment, the SFC pipeline tables are as follows:

- Table 0, Standalone SFC classifier
- Table 1, Transport Ingress
- Table 2, Path Mapper
- Table 3, Path Mapper ACL
- Table 4, Next Hop
- Table 10, Transport Egress

When SFC is integrated with the Netvirt and Genius ODL projects for an OpenStack deployment, the table numbers are the following:

- Table 83, Transport Ingress
- Table 84, Path Mapper
- Table 85, Path Mapper ACL

- Table 86, Next Hop
- Table 87, Transport Egress

This table structure will not change as a result of this feature. Each of the tables is detailed in the following sections.

SFC flow output

Below is a dump of the existing NSH flows for a standalone SFC deployment using VXLAN-GPE. Notice the duration, n_packets, and n_bytes fields have been removed for brevity.

```
cookie=0x14, table=0, priority=5 actions=goto_table:1
cookie=0x14, table=1, priority=300,udp,nw_dst=10.0.0.10,tp_dst=6633 actions=output:0
cookie=0x14, table=1, priority=300,udp,in_port=0,tp_dst=6633 actions=LOCAL
cookie=0x14, table=1, priority=250,nsp=8388641 actions=goto_table:4
cookie=0x14, table=1, priority=250,nsp=33 actions=goto_table:4
cookie=0x14, table=1, priority=5 actions=drop
cookie=0x14, table=2, priority=5 actions=goto_table:3
cookie=0x14, table=3, priority=5 actions=goto_table:4
cookie=0x14, table=4, priority=550,nsi=255,nsp=8388641 actions=load:0xa00000a->NXM_NX_
↪TUN_IPV4_DST[],goto_table:10
cookie=0x14, table=4, priority=550,nsi=255,nsp=33 actions=load:0xa00000a->NXM_NX_TUN_
↪IPV4_DST[],goto_table:10
cookie=0x14, table=4, priority=5 actions=goto_table:10
cookie=0xba5eba1100000102, table=10, priority=660,nsi=254,nsp=8388641,nshc1=0␣
↪actions=load:0x4->NXM_NX_TUN_GPE_NP[],IN_PORT
cookie=0xba5eba1100000102, table=10, priority=660,nsi=254,nsp=33,nshc1=0␣
↪actions=load:0x4->NXM_NX_TUN_GPE_NP[],IN_PORT
cookie=0xba5eba1100000103, table=10, priority=655,nsi=254,nsp=8388641,in_port=1␣
↪actions=move:NXM_NX_NSH_MDTYPE[]->NXM_NX_NSH_MDTYPE[],move:NXM_NX_NSH_NP[]->NXM_NX_
↪NSH_NP[],move:NXM_NX_NSI[]->NXM_NX_NSI[],move:NXM_NX_NSP[0..23]->NXM_NX_NSP[0..23],
↪move:NXM_NX_NSH_C1[]->NXM_NX_TUN_IPV4_DST[],move:NXM_NX_NSH_C2[]->NXM_NX_TUN_ID[0..
↪31],load:0x4->NXM_NX_TUN_GPE_NP[],IN_PORT
cookie=0xba5eba1100000101, table=10, priority=655,nsi=255,nsp=8388641,in_port=1␣
↪actions=move:NXM_NX_NSH_MDTYPE[]->NXM_NX_NSH_MDTYPE[],move:NXM_NX_NSH_NP[]->NXM_NX_
↪NSH_NP[],move:NXM_NX_NSH_C1[]->NXM_NX_NSH_C1[],move:NXM_NX_NSH_C2[]->NXM_NX_NSH_
↪C2[],move:NXM_NX_NSH_C3[]->NXM_NX_NSH_C3[],move:NXM_NX_NSH_C4[]->NXM_NX_NSH_C4[],
↪move:NXM_NX_TUN_ID[0..31]->NXM_NX_TUN_ID[0..31],load:0x4->NXM_NX_TUN_GPE_NP[],IN_
↪PORT
cookie=0xba5eba1100000103, table=10, priority=655,nsi=254,nsp=33,in_port=1␣
↪actions=move:NXM_NX_NSH_MDTYPE[]->NXM_NX_NSH_MDTYPE[],move:NXM_NX_NSH_NP[]->NXM_NX_
↪NSH_NP[],move:NXM_NX_NSI[]->NXM_NX_NSI[],move:NXM_NX_NSP[0..23]->NXM_NX_NSP[0..23],
↪move:NXM_NX_NSH_C1[]->NXM_NX_TUN_IPV4_DST[],move:NXM_NX_NSH_C2[]->NXM_NX_TUN_ID[0..
↪31],load:0x4->NXM_NX_TUN_GPE_NP[],IN_PORT
cookie=0xba5eba1100000101, table=10, priority=655,nsi=255,nsp=33,in_port=1␣
↪actions=move:NXM_NX_NSH_MDTYPE[]->NXM_NX_NSH_MDTYPE[],move:NXM_NX_NSH_NP[]->NXM_NX_
↪NSH_NP[],move:NXM_NX_NSH_C1[]->NXM_NX_NSH_C1[],move:NXM_NX_NSH_C2[]->NXM_NX_NSH_
↪C2[],move:NXM_NX_NSH_C3[]->NXM_NX_NSH_C3[],move:NXM_NX_NSH_C4[]->NXM_NX_NSH_C4[],
↪move:NXM_NX_TUN_ID[0..31]->NXM_NX_TUN_ID[0..31],load:0x4->NXM_NX_TUN_GPE_NP[],IN_
↪PORT
cookie=0xba5eba1100000103, table=10, priority=650,nsi=254,nsp=8388641␣
↪actions=move:NXM_NX_NSH_MDTYPE[]->NXM_NX_NSH_MDTYPE[],move:NXM_NX_NSH_NP[]->NXM_NX_
↪NSH_NP[],move:NXM_NX_NSI[]->NXM_NX_NSI[],move:NXM_NX_NSP[0..23]->NXM_NX_NSP[0..23],
↪move:NXM_NX_NSH_C1[]->NXM_NX_TUN_IPV4_DST[],move:NXM_NX_NSH_C2[]->NXM_NX_TUN_ID[0..
↪31],load:0x4->NXM_NX_TUN_GPE_NP[],output:1
cookie=0xba5eba1100000101, table=10, priority=650,nsi=255,nsp=8388641␣
↪actions=move:NXM_NX_NSH_MDTYPE[]->NXM_NX_NSH_MDTYPE[],move:NXM_NX_NSH_NP[]->NXM_NX_
↪NSH_NP[],move:NXM_NX_NSH_C1[]->NXM_NX_NSH_C1[],move:NXM_NX_NSH_C2[]->NXM_NX_NSH_
↪C2[],move:NXM_NX_NSH_C3[]->NXM_NX_NSH_C3[],move:NXM_NX_NSH_C4[]->NXM_NX_NSH_C4[],
↪move:NXM_NX_TUN_ID[0..31]->NXM_NX_TUN_ID[0..31],load:0x4->NXM_NX_TUN_GPE_NP[],
↪output:1
```

(continues on next page)

(continued from previous page)

```
cookie=0xba5eba1100000101, table=10, priority=650,nsi=255,nsp=33 actions=move:NXM_NX_
↪NSH_MDTYPE[]->NXM_NX_NSH_MDTYPE[],move:NXM_NX_NSH_NP[]->NXM_NX_NSH_NP[],move:NXM_NX_
↪NSH_C1[]->NXM_NX_NSH_C1[],move:NXM_NX_NSH_C2[]->NXM_NX_NSH_C2[],move:NXM_NX_NSH_
↪C3[]->NXM_NX_NSH_C3[],move:NXM_NX_NSH_C4[]->NXM_NX_NSH_C4[],move:NXM_NX_TUN_ID[0..
↪31]->NXM_NX_TUN_ID[0..31],load:0x4->NXM_NX_TUN_GPE_NP[],output:1
cookie=0xba5eba1100000103, table=10, priority=650,nsi=254,nsp=33 actions=move:NXM_NX_
↪NSH_MDTYPE[]->NXM_NX_NSH_MDTYPE[],move:NXM_NX_NSH_NP[]->NXM_NX_NSH_NP[],move:NXM_NX_
↪NSI[]->NXM_NX_NSI[],move:NXM_NX_NSP[0..23]->NXM_NX_NSP[0..23],move:NXM_NX_NSH_C1[]->
↪NXM_NX_TUN_IPV4_DST[],move:NXM_NX_NSH_C2[]->NXM_NX_TUN_ID[0..31],load:0x4->NXM_NX_
↪TUN_GPE_NP[],output:1
cookie=0x14, table=10, priority=5 actions=drop
```

The following 3 samples are taken from an OPNFV SFC deployment, using the ODL Netvirt project as a classifier. The classifier-SFF and SFF-SFF encapsulation is VXLAN-GPE, and the SFF-SF encapsulation is ETH+NSH.

Netvirt classifier tables:

```
cookie=0xf005ba1100000001, table=100, priority=520,nsi=253,nsp=39 actions=move:NXM_NX_
↪NSH_C4[]->NXM_NX_REG6[],pop_nsh,resubmit(,220)
cookie=0xf005ba1100000001, table=100, priority=511,encap_eth_type=0x894f,tun_dst=0.0.
↪0.0 actions=resubmit(,17)
cookie=0xf005ba1100000001, table=100, priority=510,encap_eth_type=0x894f␣
↪actions=resubmit(,83)
cookie=0xf005ba1100000001, table=100, priority=500 actions=goto_table:101

cookie=0xf005ba1100000002, table=101, priority=500,tcp,in_port=7,tp_dst=80
actions=push_nsh,load:0x1->NXM_NX_NSH_MDTYPE[],load:0x3->NXM_NX_NSH_NP[],load:0x27->
↪NXM_NX_NSP[0..23],load:0xff->NXM_NX_NSI[],load:0xffffff->NXM_NX_NSH_C1[],load:0->
↪NXM_NX_NSH_C2[],resubmit(,17)
cookie=0xf005ba1100000002, table=101, priority=10 actions=resubmit(,17)

cookie=0xf005ba1100000003, table=221, priority=260,nshc1=16777215 actions=load:0->NXM_
↪NX_NSH_C1[],goto_table:222
cookie=0xf005ba1100000003, table=221, priority=250 actions=resubmit(,220)

cookie=0xf005ba1100000004, table=222, priority=260,nshc1=0,nshc2=0
actions=move:NXM_NX_REG0[]->NXM_NX_NSH_C1[],move:NXM_NX_TUN_ID[0..31]->NXM_NX_NSH_
↪C2[],move:NXM_NX_REG6[]->NXM_NX_NSH_C4[],load:0->NXM_NX_TUN_ID[0..31],goto_table:223
cookie=0xf005ba1100000004, table=222, priority=250 actions=goto_table:223

cookie=0xf005ba1100000005, table=223, priority=260,nsp=39 actions=resubmit(,83)
```

These flows are the SFC flows when SFC is integrated with the Netvirt and Genius ODL projects.

```
cookie=0x14, table=83, priority=250,nsp=39 actions=goto_table:86
cookie=0x14, table=83, priority=5 actions=resubmit(,17)

cookie=0x14, table=84, priority=5 actions=goto_table:85

cookie=0x14, table=85, priority=5 actions=goto_table:86

cookie=0x14, table=86, priority=550,nsi=254,nsp=39 actions=load:0xfe163eccbf0c->NXM_
↪NX_ENCAP_ETH_SRC[],load:0xfa163eccbf0c->NXM_NX_ENCAP_ETH_DST[],goto_table:87
cookie=0x14, table=86, priority=550,nsi=255,nsp=39 actions=load:0xfe163e8e7bca->NXM_
↪NX_ENCAP_ETH_SRC[],load:0xfa163e8e7bca->NXM_NX_ENCAP_ETH_DST[],goto_table:87
cookie=0x14, table=86, priority=5 actions=goto_table:87
```

(continues on next page)

(continued from previous page)

```
cookie=0xba5eba1100000207, table=87, priority=680,nsi=253,nsp=39,nshc1=2887643148,
↪nshc2=0 actions=resubmit(,17)
cookie=0xba5eba1100000205, table=87, priority=660,nsi=253,nsp=39,nshc1=2887643148␣
↪actions=move:NXM_NX_NSH_C1[]->NXM_NX_TUN_IPV4_DST[],move:NXM_NX_NSH_C2[]->NXM_NX_
↪TUN_ID[0..31],pop_nsh,resubmit(,36)
cookie=0xba5eba1100000203, table=87, priority=680,nsi=253,nsp=39,nshc1=0 actions=pop_
↪nsh,set_field:fa:16:3e:cc:bf:0c->eth_src,resubmit(,17)
cookie=0xba5eba1100000206, table=87, priority=670,nsi=253,nsp=39,nshc2=0␣
↪actions=move:NXM_NX_NSH_C1[]->NXM_NX_TUN_IPV4_DST[],move:NXM_NX_NSH_C2[]->NXM_NX_
↪TUN_ID[0..31],output:1
cookie=0xba5eba1100000202, table=87, priority=650,nsi=254,nsp=39 actions=load:0x1700->
↪NXM_NX_REG6[],resubmit(,220)
cookie=0xba5eba1100000202, table=87, priority=650,nsi=255,nsp=39 actions=load:0x1800->
↪NXM_NX_REG6[],resubmit(,220)
cookie=0xba5eba1100000204, table=87, priority=650,nsi=253,nsp=39 actions=move:NXM_NX_
↪NSH_C1[]->NXM_NX_TUN_IPV4_DST[],move:NXM_NX_NSH_C2[]->NXM_NX_TUN_ID[0..31],pop_nsh,
↪output:1
cookie=0x14, table=87, n_packets=0, priority=5 actions=resubmit(,17)
```

The following flows are the rest of the pertinent ODL Genius project flows shown for completeness.

```
cookie=0x8000001, table=0, priority=5,in_port=1 actions=write_metadata:0x10000000001/
↪0xffffff0000000001,goto_table:36
cookie=0x8000000, table=0, priority=4,in_port=2,vlan_tci=0x0000/0x1fff actions=write_
↪metadata:0x40000000001/0xffffff0000000001,goto_table:17
cookie=0x8000000, table=0, priority=4,in_port=6,vlan_tci=0x0000/0x1fff actions=write_
↪metadata:0x150000000000/0xffffff0000000001,goto_table:17
cookie=0x8000000, table=0, priority=4,in_port=7,vlan_tci=0x0000/0x1fff actions=write_
↪metadata:0x160000000000/0xffffff0000000001,goto_table:17
cookie=0x8000000, table=0, priority=4,in_port=8,vlan_tci=0x0000/0x1fff actions=write_
↪metadata:0x170000000000/0xffffff0000000001,goto_table:17
cookie=0x8000000, table=0, priority=4,in_port=9,vlan_tci=0x0000/0x1fff actions=write_
↪metadata:0x180000000000/0xffffff0000000001,goto_table:17

cookie=0x8000001, table=17, priority=10,metadata=0x40000000000/0xffffff0000000000␣
↪actions=load:0x186a0->NXM_NX_REG3[0..24],write_metadata:0x9000040000030d40/
↪0xfffffffffffffffe,goto_table:19
cookie=0x8040000, table=17, priority=10,metadata=0x9000040000000000/
↪0xffffff0000000000 actions=load:0x4->NXM_NX_REG1[0..19],load:0x138a->NXM_NX_REG7[0..
↪15],write_metadata:0xa00004138a000000/0xfffffffffffffffe,goto_table:43
cookie=0x6900000, table=17, priority=10,metadata=0x150000000000/0xffffff0000000000␣
↪actions=write_metadata:0x8000150000000000/0xfffffffffffffffe,goto_table:210
cookie=0x8040000, table=17, priority=10,metadata=0x9000150000000000/
↪0xffffff0000000000 actions=load:0x15->NXM_NX_REG1[0..19],load:0x139c->NXM_NX_REG7[0.
↪.15],write_metadata:0xa00015139c000000/0xfffffffffffffffe,goto_table:43
cookie=0x8000001, table=17, priority=10,metadata=0x8000150000000000/
↪0xffffff0000000000 actions=load:0x186b3->NXM_NX_REG3[0..24],write_
↪metadata:0x9000150000030d66/0xfffffffffffffffe,goto_table:19
cookie=0x8040000, table=17, priority=10,metadata=0x9000160000000000/
↪0xffffff0000000000 actions=load:0x16->NXM_NX_REG1[0..19],load:0x139c->NXM_NX_REG7[0.
↪.15],write_metadata:0xa00016139c000000/0xfffffffffffffffe,goto_table:43
cookie=0x8000001, table=17, priority=10,metadata=0x8000160000000000/
↪0xffffff0000000000 actions=load:0x186b3->NXM_NX_REG3[0..24],write_
↪metadata:0x9000160000030d66/0xfffffffffffffffe,goto_table:19
cookie=0x8040000, table=17, priority=10,metadata=0x9000170000000000/
↪0xffffff0000000000 actions=load:0x17->NXM_NX_REG1[0..19],load:0x139c->NXM_NX_REG7[0.
↪.15],write_metadata:0xa00017139c000000/0xfffffffffffffffe,goto_table:43
```

(continues on next page)

(continued from previous page)

```
cookie=0x8040000, table=17, priority=10,metadata=0x9000180000000000/
 ↪0xffffff0000000000 actions=load:0x18->NXM_NX_REG1[0..19],load:0x139c->NXM_NX_REG7[0.
 ↪.15],write_metadata:0xa00018139c000000/0xfffffffffffffffe,goto_table:43
cookie=0x8000001, table=17, priority=10,metadata=0x8000180000000000/
 ↪0xffffff0000000000 actions=load:0x186b3->NXM_NX_REG3[0..24],write_
 ↪metadata:0x9000180000030d66/0xfffffffffffffffe,goto_table:19
cookie=0x8030000, table=17, priority=10,metadata=0x180000000000/0xffffff0000000000␣
 ↪actions=write_metadata:0x8000180000000000/0xfffffffffffffffe,goto_table:83
cookie=0x8000001, table=17, priority=10,metadata=0x8000170000000000/
 ↪0xffffff0000000000 actions=load:0x186b3->NXM_NX_REG3[0..24],write_
 ↪metadata:0x9000170000030d66/0xfffffffffffffffe,goto_table:19
cookie=0xf005ba1100000001, table=17, priority=10,metadata=0x4000160000000000/
 ↪0xffffff0000000000 actions=write_metadata:0x8000160000000000/0xfffffffffffffffe,
 ↪goto_table:100
cookie=0x6900000, table=17, priority=10,metadata=0x160000000000/0xffffff0000000000␣
 ↪actions=write_metadata:0x4000160000000000/0xfffffffffffffffe,goto_table:210
cookie=0x8030000, table=17, priority=10,metadata=0x170000000000/0xffffff0000000000␣
 ↪actions=write_metadata:0x4000170000000000/0xfffffffffffffffe,goto_table:83
cookie=0xf005ba1100000001, table=17, priority=10,metadata=0x4000170000000000/
 ↪0xffffff0000000000 actions=write_metadata:0x8000170000000000/0xfffffffffffffffe,
 ↪goto_table:100
cookie=0x8000000, table=17, priority=0,metadata=0x8000000000000000/0xf000000000000000␣
 ↪actions=write_metadata:0x9000000000000000/0xf000000000000000,goto_table:80

cookie=0xf005ba1100000006, table=36, priority=10,encap_eth_type=0x894f,tun_id=0␣
 ↪actions=resubmit(,100)
cookie=0x900139c, table=36, priority=5,tun_id=0x2f actions=write_
 ↪metadata:0x139c000000/0xffffffffff000000,goto_table:51
cookie=0x9000000, table=36, priority=5,tun_id=0 actions=goto_table:83
cookie=0x90186bb, table=36, priority=5,tun_id=0x186bb actions=resubmit(,25)
cookie=0x90186bc, table=36, priority=5,tun_id=0x186bc actions=resubmit(,25)
cookie=0x90186bd, table=36, priority=5,tun_id=0x186bd actions=resubmit(,25)
cookie=0x90186be, table=36, priority=5,tun_id=0x186be actions=resubmit(,25)

cookie=0x8000007, table=220, priority=10,reg6=0x90000400,metadata=0x1/0x1 actions=drop
cookie=0x8000007, table=220, priority=9,reg6=0x90001500 actions=output:6
cookie=0x8000007, table=220, priority=9,reg6=0x90001600 actions=output:7
cookie=0x8000007, table=220, priority=9,reg6=0x90001700 actions=output:8
cookie=0xf005ba1100000003, table=220, priority=8,reg6=0x1700 actions=load:0x90001700->
 ↪NXM_NX_REG6[],load:0xac1df00c->NXM_NX_REG0[],write_metadata:0/0xfffffffffe,goto_
 ↪table:221
cookie=0xf005ba1100000003, table=220, priority=8,reg6=0x80001500␣
 ↪actions=load:0x90001500->NXM_NX_REG6[],load:0xac1df00c->NXM_NX_REG0[],write_
 ↪metadata:0/0xfffffffffe,goto_table:221
cookie=0x6900000, table=220, priority=6,reg6=0x1500 actions=load:0x80001500->NXM_NX_
 ↪REG6[],write_metadata:0/0xfffffffffe,goto_table:239
cookie=0x8000007, table=220, priority=9,reg6=0x90000400 actions=output:2
cookie=0xf005ba1100000003, table=220, priority=8,reg6=0x400 actions=load:0x90000400->
 ↪NXM_NX_REG6[],load:0xac1df00c->NXM_NX_REG0[],write_metadata:0/0xfffffffffe,goto_
 ↪table:221
cookie=0x8000007, table=220, priority=9,reg6=0x90001800 actions=output:9
cookie=0xf005ba1100000003, table=220, priority=8,reg6=0x1800 actions=load:0x90001800->
 ↪NXM_NX_REG6[],load:0xac1df00c->NXM_NX_REG0[],write_metadata:0/0xfffffffffe,goto_
 ↪table:221
cookie=0xf005ba1100000003, table=220, priority=8,reg6=0x100 actions=load:0x90000100->
 ↪NXM_NX_REG6[],load:0xac1df00b->NXM_NX_REG0[],write_metadata:0/0xfffffffffe,goto_
 ↪table:221
```

(continues on next page)

(continued from previous page)

```
cookie=0x8000007, table=220, priority=9,reg6=0x90000100 actions=output:1
cookie=0xf005ba1100000003, table=220, priority=8,reg6=0x80001600␣
↪actions=load:0x90001600->NXM_NX_REG6[],load:0xac1df00c->NXM_NX_REG0[],write_
↪metadata:0/0xfffffffffe,goto_table:221
cookie=0x6900000, table=220, priority=6,reg6=0x1600 actions=load:0x80001600->NXM_NX_
↪REG6[],write_metadata:0/0xfffffffffe,goto_table:239
```

Standalone SFC classifier Table

This table serves as an SFC classifier when SFC is not used with OpenStack. This table maps subscriber traffic to Rendered Service Paths (RSPs) by implementing simple ACLs.

Transport Ingress Table

This table serves to only allow the expected transports or protocols to enter SFC, and drops everything else. There will be an entry per expected tunnel transport type to be received in SFC, as established in the SFC configuration.

Currently the only way to check for packets with NSH is to check if the NSP (Network Services Path), which is the Service Chain ID, is present. This means that there will be a transport ingress flow for each service chain configured, as follows. Notice this forwards packets directly to the NextHop table since neither of the PathMapper tables are needed for NSH.

```
priority=250,nsp=33 actions=goto_table:4
priority=250,nsp=8388641 actions=goto_table:4
```

Path Mapper Table

This table maps transport information to a particular Service Chain. Currently this table is not used for NSH, but is used for instance with VLAN or MPLS to map a VLAN tag or MPLS label to a particular service chain. Currently the VLAN and MPLS transports have limited support.

Path Mapper ACL Table

This table is used for TCP Proxy type Service Functions (SFs). Flows are only added to this table as a result of a PacketIn, and they will have an inactivity expiration timeout of 60 seconds. If a SF has the TCP Proxy flag set true, then a flow will be created in the Transport Egress table for the SF that will cause a PacketIn to OpenDaylight for packets that egress to the SF. Since TCP Proxy SFs can generate their own packets, this table maps those TCP Proxy SF generated packets to the corresponding service chain.

Next Hop Table

This table determines where SFC packets should be sent next, typically either to an SF or to another SFF. For NSH, there will be a match on both the NSP (service chain ID) and NSI (service chain hop) to determine the next hop.

For a standalone SFC deployment, when using VXLAN-GPE towards the SFs, the VXLAN-GPE tunnel destination IPv4 address is set, and the packets are sent to the TransportEgress table, as follows.

```
priority=550,nsi=255,nsp=8388641 actions=load:0xa00000a->NXM_NX_TUN_IPV4_DST[],goto_
↪table:10
priority=550,nsi=255,nsp=33 actions=load:0xa00000a->NXM_NX_TUN_IPV4_DST[],goto_
↪table:10
```

For as OpenStack SFC deployment, when using Eth+NSH towards the SFs, the outer Ethernet addresses are set, and the packets are sent to the TransportEgress table, as follows.

```
priority=550,nsi=254,nsp=39 actions=load:0xfe163eccbf0c->NXM_NX_ENCAP_ETH_SRC[],
↪load:0xfa163eccbf0c->NXM_NX_ENCAP_ETH_DST[],goto_table:87
priority=550,nsi=255,nsp=39 actions=load:0xfe163e8e7bca->NXM_NX_ENCAP_ETH_SRC[],
↪load:0xfa163e8e7bca->NXM_NX_ENCAP_ETH_DST[],goto_table:87
```

Transport Egress Table

This table prepares packets for egress by either setting tunnel information, such as VLAN tags, VXLAN-GPE information, or encapsulating MPLS. These flows also determine the output port where the packets should be sent. The NSH TransportEgress flows are more complicated than the rest, and are identified by their cookie values. The available NSH TransportEgress cookies are listed below.

- 0xba5eba1100000101 - TRANSPORT_EGRESS_NSH_VXGPE_COOKIE
- 0xba5eba1100000102 - TRANSPORT_EGRESS_NSH_VXGPE_NSC_COOKIE
- 0xba5eba1100000103 - TRANSPORT_EGRESS_NSH_VXGPE_LASTHOP_COOKIE
- 0xba5eba1100000201 - TRANSPORT_EGRESS_NSH_ETH_COOKIE
- 0xba5eba1100000202 - TRANSPORT_EGRESS_NSH_ETH_LOGICAL_COOKIE
- 0xba5eba1100000203 - TRANSPORT_EGRESS_NSH_ETH_LASTHOP_PIPELINE_COOKIE
- 0xba5eba1100000204 - TRANSPORT_EGRESS_NSH_ETH_LASTHOP_TUNNEL_REMOTE_COOKIE
- 0xba5eba1100000205 - TRANSPORT_EGRESS_NSH_ETH_LASTHOP_TUNNEL_LOCAL_COOKIE
- 0xba5eba1100000206 - TRANSPORT_EGRESS_NSH_ETH_LASTHOP_NSH_REMOTE_COOKIE
- 0xba5eba1100000207 - TRANSPORT_EGRESS_NSH_ETH_LASTHOP_NSH_LOCAL_COOKIE

As can be seen in the VXGPE NSH flows below, all of the NSH TransportEgress flows match on at least the NSP (service chain ID) and NSI (hop in the chain). Notice some of the flows match on the in_port and then output the packets to IN_PORT, while other seemingly duplicate flows output the packets to a specific port without matching on the in_port. These flows are indeed exactly the same, except for the differences just mentioned and the flow priorities. This is because according to the OpenFlow specification, the only way a packet can be sent out on the same port it was received on is by deliberately sending it out using the IN_PORT port string, or it will be dropped, in an effort to avoid packet loops.

Notice that many of these flows have move actions. These are because in the branched version of OVS 2.6 with NSH, these values are not explicitly maintained when the packet is egressed.

Some additional logic is needed on the last hop, which is when packets have traversed the entire service chain, and need to be sent out of SFC. Information for where to send the packet after SFC is set in the NSH C1 and C2 metadata headers by the SFC classifier. The C1 header is the VXLAN-GPE tunnel destination IPv4 address, and C2 is the VXLAN-GPE VNI field.

Standalone SFC TransportEgress flows:

```
cookie=0xba5eba1100000102, table=10,
priority=660,nsi=254,nsp=33,nshc1=0
actions=load:0x4->NXM_NX_TUN_GPE_NP[],IN_PORT

cookie=0xba5eba1100000103, table=10,
priority=655,nsi=254,nsp=33,in_port=1
actions=move:NXM_NX_NSH_MDTYPE[]->NXM_NX_NSH_MDTYPE[],
    move:NXM_NX_NSH_NP[]->NXM_NX_NSH_NP[],
    move:NXM_NX_NSI[]->NXM_NX_NSI[],
    move:NXM_NX_NSP[0..23]->NXM_NX_NSP[0..23],
    move:NXM_NX_NSH_C1[]->NXM_NX_TUN_IPV4_DST[],
    move:NXM_NX_NSH_C2[]->NXM_NX_TUN_ID[0..31],
    load:0x4->NXM_NX_TUN_GPE_NP[],
    IN_PORT

cookie=0xba5eba1100000101, table=10,
priority=655,nsi=255,nsp=33,in_port=1
actions=move:NXM_NX_NSH_MDTYPE[]->NXM_NX_NSH_MDTYPE[],
    move:NXM_NX_NSH_NP[]->NXM_NX_NSH_NP[],
    move:NXM_NX_NSH_C1[]->NXM_NX_NSH_C1[],
    move:NXM_NX_NSH_C2[]->NXM_NX_NSH_C2[],
    move:NXM_NX_NSH_C3[]->NXM_NX_NSH_C3[],
    move:NXM_NX_NSH_C4[]->NXM_NX_NSH_C4[],
    move:NXM_NX_TUN_ID[0..31]->NXM_NX_TUN_ID[0..31],
    load:0x4->NXM_NX_TUN_GPE_NP[],
    IN_PORT

cookie=0xba5eba1100000101, table=10,
priority=650,nsi=255,nsp=33
actions=move:NXM_NX_NSH_MDTYPE[]->NXM_NX_NSH_MDTYPE[],
    move:NXM_NX_NSH_NP[]->NXM_NX_NSH_NP[],
    move:NXM_NX_NSH_C1[]->NXM_NX_NSH_C1[],
    move:NXM_NX_NSH_C2[]->NXM_NX_NSH_C2[],
    move:NXM_NX_NSH_C3[]->NXM_NX_NSH_C3[],
    move:NXM_NX_NSH_C4[]->NXM_NX_NSH_C4[],
    move:NXM_NX_TUN_ID[0..31]->NXM_NX_TUN_ID[0..31],
    load:0x4->NXM_NX_TUN_GPE_NP[],
    output:1

cookie=0xba5eba1100000103, table=10,
priority=650,nsi=254,nsp=33
actions=move:NXM_NX_NSH_MDTYPE[]->NXM_NX_NSH_MDTYPE[],
    move:NXM_NX_NSH_NP[]->NXM_NX_NSH_NP[],
    move:NXM_NX_NSI[]->NXM_NX_NSI[],
    move:NXM_NX_NSP[0..23]->NXM_NX_NSP[0..23],
    move:NXM_NX_NSH_C1[]->NXM_NX_TUN_IPV4_DST[],
    move:NXM_NX_NSH_C2[]->NXM_NX_TUN_ID[0..31],
    load:0x4->NXM_NX_TUN_GPE_NP[],
    output:1
```

SFC integrated with OpenStack flows:

```
cookie=0xba5eba1100000207, table=87,
priority=680,nsi=253,nsp=39,nshc1=2887643148,nshc2=0
actions=resubmit(,17)

cookie=0xba5eba1100000205, table=87,
priority=660,nsi=253,nsp=39,nshc1=2887643148
actions=move:NXM_NX_NSH_C1[]->NXM_NX_TUN_IPV4_DST[],
    move:NXM_NX_NSH_C2[]->NXM_NX_TUN_ID[0..31],
    pop_nsh,resubmit(,36)

cookie=0xba5eba1100000203, table=87,
priority=680,nsi=253,nsp=39,nshc1=0
actions=pop_nsh,set_field:fa:16:3e:cc:bf:0c->eth_src,resubmit(,17)

cookie=0xba5eba1100000206, table=87,
priority=670,nsi=253,nsp=39,nshc2=0
actions=move:NXM_NX_NSH_C1[]->NXM_NX_TUN_IPV4_DST[],
    move:NXM_NX_NSH_C2[]->NXM_NX_TUN_ID[0..31],
    output:1

cookie=0xba5eba1100000202, table=87,
priority=650,nsi=254,nsp=39
actions=load:0x1700->NXM_NX_REG6[],resubmit(,220)

cookie=0xba5eba1100000202, table=87,
priority=650,nsi=255,nsp=39
actions=load:0x1800->NXM_NX_REG6[],resubmit(,220)

cookie=0xba5eba1100000204, table=87,
priority=650,nsi=253,nsp=39
actions=move:NXM_NX_NSH_C1[]->NXM_NX_TUN_IPV4_DST[],
    move:NXM_NX_NSH_C2[]->NXM_NX_TUN_ID[0..31],
    pop_nsh,output:1
```

Netvirt Classifier Tables

The Netvirt Classifier tables are divided between what are called Ingress and Egress classifier tables. The Ingress classifier tables determine if packets should be sent to SFC or not, and if they are, it determines on what Service Chain. The Ingress classifier also restores packets to their final destination at chain egress if it is on the same node as origin.

Once the packets are processed by the Ingress classifier, they are sent down the rest of the Netvirt pipeline to allow the rest of the services to process the packets. The Egress classifier tables send the packets to the SFC tables if necessary, be it on the same SFF (compute host) or on a different SFF.

In addition to the table numbers, the Netvirt Classifier tables are identified by their cookie values, as can be seen below.

- 0xF005BA1100000001 - INGRESS_CLASSIFIER_FILTER_COOKIE, table = 100
- 0xF005BA1100000002 - INGRESS_CLASSIFIER_ACL_COOKIE, table = 101
- 0xF005BA1100000003 - EGRESS_CLASSIFIER_FILTER_COOKIE, table = 200
- 0xF005BA1100000004 - EGRESS_CLASSIFIER_NEXTHOP_COOKIE, table = 200
- 0xF005BA1100000005 - EGRESS_CLASSIFIER_TPORTEGRESS_COOKIE, table = 200
- 0xF005BA1100000006 - INGRESS_CLASSIFIER_CAPTURE_SFC_TUNNEL_TRAFFIC_COOKIE, table = 36

The Netvirt Classifier tables are listed below.

```
cookie=0xf005ba1100000001, table=100, priority=520,nsi=253,nsp=39 actions=move:NXM_NX_
↪NSH_C4[]->NXM_NX_REG6[],pop_nsh,resubmit(,220)
cookie=0xf005ba1100000001, table=100, priority=511,encap_eth_type=0x894f,tun_dst=0.0.
↪0.0 actions=resubmit(,17)
cookie=0xf005ba1100000001, table=100, priority=510,encap_eth_type=0x894f␣
↪actions=resubmit(,83)
cookie=0xf005ba1100000001, table=100, priority=500 actions=goto_table:101
cookie=0xf005ba1100000002, table=101, priority=500,tcp,in_port=7,tp_dst=80
actions=push_nsh,load:0x1->NXM_NX_NSH_MDTYPE[],
        load:0x3->NXM_NX_NSH_NP[],
        load:0x27->NXM_NX_NSP[0..23],
        load:0xff->NXM_NX_NSI[],
        load:0xffffff->NXM_NX_NSH_C1[],
        load:0->NXM_NX_NSH_C2[],
        resubmit(,17)
cookie=0xf005ba1100000002, table=101, priority=10 actions=resubmit(,17)
cookie=0xf005ba1100000003, table=221, priority=260,nshc1=16777215 actions=load:0->NXM_
↪NX_NSH_C1[],goto_table:222
cookie=0xf005ba1100000003, table=221, priority=250 actions=resubmit(,220)
cookie=0xf005ba1100000004, table=222, priority=260,nshc1=0,nshc2=0
actions=move:NXM_NX_REG0[]->NXM_NX_NSH_C1[],
        move:NXM_NX_TUN_ID[0..31]->NXM_NX_NSH_C2[],
        move:NXM_NX_REG6[]->NXM_NX_NSH_C4[],
        load:0->NXM_NX_TUN_ID[0..31],
        goto_table:223
cookie=0xf005ba1100000004, table=222, priority=250 actions=goto_table:223
cookie=0xf005ba1100000005, table=223, priority=260,nsp=39 actions=resubmit(,83)
```

Changes to the SFC pipeline

The tables that will be affected by this feature are detailed below.

Transport Ingress Table

For NSH, this table will now match on either the ether_type or the packet_type as follows:

- When the packet arrives from a standard port as eth+nsh, then match on ether_type=0x894F
- When the packet arrives from a vxlan+eth+nsh tunnel port and the resulting packet after tunnel decapsulation is eth+nsh, then match on ether_type=0x894F
- When the packet arrives from a vxlan+nsh tunnel port, and the resulting packet after tunnel decapsulation is directly nsh, then match on packet_type=(1,0x894F)

Thus, a packet will only ingress the pipeline without an outer ethernet header when doing so on a tunnel port. And in such cases, the packet will then egress to a service function where an outer ethernet header will be required. So it makes sense to normalize all packets by adding an ethernet header when missing.

This is how the transport ingress flows would look like:

priority=250,eth_type=0x894f actions=goto_table:84 priority=250,packet_type=(1,0x894f) actions=encap(ethernet),goto_table:84

Path Mapper Table

Currently this table is only used for VLAN or MPLS, but since VXLAN will be added soon, this table will be used for all transports and encapsulations. A register will be used to store the Service Chain ID and the Hop counter. For NSH, we will need to match on ether_type or packet_type, in addition to the NSP and NSI.

priority=250,eth_type=0x894f actions=move:nsp->regX[8..31],move:nsi->regX[0..7],goto_table:86

Next Hop Table

Since the PathMapper table will now be used by all protocols and transports, there will no longer be matches in this table for specific protocol details like the NSH NSP and NSI fields; instead the matches in this table will be on the register set in the PathMapper table.

When next hop is a SFF:

```
priority=550,regX[8..31]={nsp},regX[0..7]={nsi}
actions=load:{sff_ip}->NXM_NX_TUN_IPV4_DST[],goto_table:87
```

When next hop is a SF:

```
priority=550,regX[8..31]={nsp},regX[0..7]={nsi}
actions=load:{sff_mac}->NXM_NX_ENCAP_ETH_SRC[],
    load:{sf_mac}->NXM_NX_ENCAP_ETH_DST[],goto_table:87
```

Transport Egress Table

Similar to the matching changes in the NextHop table, this table will now match on the register set in the PathMapper table. Additionally, the previous move actions to reset NSH fields on egress will no longer be needed. Notice the NXM_NX_TUN_GPE field will no longer be available, and the GPE NP fields will be handled internally by OVS. The NXM_NX_NSH_MDTYPE field will now be read-only.

When egress is on a tunnel port to the next SFF and transport is vxlan-gpe the extra ethernet header will be removed:

```
cookie=0xba5eba1100000101, table=10,
priority=650,regX[8..31]={nsp},regX[0..7]={nsi}
actions=decap(),output:1
```

For vxlan transport, the extra ethernet header will remain in place:

```
cookie=0xba5eba1100000101, table=10,
priority=650,regX[8..31]={nsp},regX[0..7]={nsi}
actions=output:1
```

Egress to a SF will be eth+nsh. This a sample for an OpenStack deployment:

```
cookie=0xba5eba1100000202, table=87,
priority=650,regX[8..31]={nsp},regX[0..7]={nsi}
actions=load:0x1800->NXM_NX_REG6[],resubmit(,220)
```

TODO Add tunnel egress sample flow on OpenStack deployment

On last hop for OpenStack deployments, there are some different scenarios to look at. These scenarios depend on different values of nsh context metadata so all the flows will have to match on nsh ethernet type.The most basic case is forwarding the original packet through a tunnel port with a specified destination address and tunnel id:

```
cookie=0xba5eba1100000204, table=87,
priority=650,eth_type=0x894f,regX[8..31]={nsp},regX[0..7]={nsi}
actions=move:NXM_NX_NSH_C1[]->NXM_NX_TUN_IPV4_DST[],
    move:NXM_NX_NSH_C2[]->NXM_NX_TUN_ID[0..31],decap(),decap(),output:1
```

The destination ip address might be that of the local node:

```
cookie=0xba5eba1100000205, table=87,
priority=660,eth_type=0x894f,regX[8..31]={nsp},regX[0..7]={nsi},
    nsh_mdtype=1,nshc1={local_ip}
actions=move:NXM_NX_NSH_C1[]->NXM_NX_TUN_IPV4_DST[],
    move:NXM_NX_NSH_C2[]->NXM_NX_TUN_ID[0..31],decap(),decap(),
    resubmit(,36)
```

The packet may egress through a tunnel port with the nsh encapsulation in place in case there is a nsh service on a different node that can handle it. The presence of an extra ethernet header in this case depends on the path transport. For example, for vxlan-gpe transport:

```
cookie=0xba5eba1100000206, table=87,
priority=670,eth_type=0x894f,regX[8..31]={nsp},regX[0..7]={nsi},
    nsh_mdtype=1,nshc2=0
actions=move:NXM_NX_NSH_C1[]->NXM_NX_TUN_IPV4_DST[],
    move:NXM_NX_NSH_C2[]->NXM_NX_TUN_ID[0..31],decap(),output:1
```

For vxlan transport:

```
cookie=0xba5eba1100000204, table=87,
priority=670,eth_type=0x894f,regX[8..31]={nsp},regX[0..7]={nsi},
    nsh_mdtype=1,nshc2=0
actions=move:NXM_NX_NSH_C1[]->NXM_NX_TUN_IPV4_DST[],
    move:NXM_NX_NSH_C2[]->NXM_NX_TUN_ID[0..31],output:1
```

Again, the tunnel destination might be the local node, both for vxlan-gpe and vxlan respectively:

```
cookie=0xba5eba1100000207, table=87,
priority=680,eth_type=0x894f,regX[8..31]={nsp},regX[0..7]={nsi},
    nsh_mdtype=1,nshc1={local_ip},nshc2=0
actions=decap(),resubmit(,36)

cookie=0xba5eba1100000207, table=87,
priority=680,eth_type=0x894f,regX[8..31]={nsp},regX[0..7]={nsi},
    nsh_mdtype=1,nshc1={local_ip},nshc2=0
actions=resubmit(,36)
```

One final variation for last hop is to rely on Netvirt L2/L3 services to forward the original packet to final destination:

```
cookie=0xba5eba1100000203, table=87,
priority=680,eth_type=0x894f,regX[8..31]={nsp},regX[0..7]={nsi},
   nsh_mdtype=1,nshc1=0
actions=decap(),decap(),set_field:{sf_mac}->eth_src,resubmit(,17)
```

Changes to the Nevirt Classifier pipeline

All Nevirt Classifier flows will suffer minor modifications at the very least to properly match packets by packet type.

Netvirt classifier handles nsh packets for chain termination when origin and final destination of traffic is on the same node. These packets may ingress the classifier pipeline from another nsh service, typically SFC, and it is assumed that the packet type is nsh. It may also ingress from the internal tunnel table and for this case a slight modification of the flows there is needed to normalize the packet type:

```
cookie=0xf005ba1100000006, table=36,
priority=10,eth_type=0x894f,tun_id=0
actions=decap(),resubmit(,100)

cookie=0xf005ba1100000006, table=36,
priority=10,packet_type=(1,0x894f),tun_id=0
actions=resubmit(,100)
```

The chain termination flow restores the encapsulated packet to its original destination by resetting the logical port to reg6 from nsh_c4 and resubmitting to the egress dispatcher after removing the nsh header:

```
cookie=0xf005ba1100000001, table=100,
priority=520,packet_type=(1,0x894f),nsi=<FINAL NSI>,nsp=<NSP>
actions=move:NXM_NX_NSH_C4[]->NXM_NX_REG6[],decap(),resubmit(,220)
```

The nsh filter flows make sure that any other nsh packets are not handled, via resubmit to the appropriate table. Unfortunately, since there is no service dispatching on the internal tunnel table, packet coming from there are sent to the SFC pipeline directly. These flows only have changes to properly match by packet_type:

```
cookie=0xf005ba1100000001, table=100,
priority=511,eth_type=0x894f,tun_dst=0.0.0.0
actions=resubmit(,17)

cookie=0xf005ba1100000001, table=100,
priority=511,packet_type=(1,0x894f),tun_dst=0.0.0.0
actions=resubmit(,17)

cookie=0xf005ba1100000001, table=100,
priority=510,packet_type=(1,0x894f) actions=resubmit(,83)

cookie=0xf005ba1100000001, table=100,
priority=500
actions=goto_table:101
```

Apart from the obvious changes required by the new OVS NSH implementation, another consequence is that the packet cannot be Eth+NSH encapsulated previously to being handled by other ingress services coming after the ingress classifier, as read/write operations on Ethernet header fields would then apply to the outer header instead of the inner one. This requires to delay encapsulation until the egress classifier and use a temporary register to store the path id and index. The acl flow stores the NSP in the first 3 bytes of a register and the NSI on the last byte of that same register:

```
cookie=0xf005ba1100000002, table=101,
priority=500,tcp,in_port=7,tp_dst=80
actions=load:{NSP}->regX[8..31],load:{START NSI}->regX[0..7],
    resubmit(,17)

cookie=0xf005ba1100000002, table=101,
priority=10
actions=resubmit(,17)
```

The egress classifier filter flows check that register holds a value as a precondition to continue handling the packet, otherwise the packet was not really classified by the ingress classifier:

```
cookie=0xf005ba1100000003, table=221,
priority=260,regX=0
actions=resubmit(,220)

cookie=0xf005ba1100000003, table=221,
priority=250
actions=goto_table:222
```

The next hop flow add the nsh encapsulation and sets the header values, restoring nsp and nsi from the register and resetting that register to avoid further classification:

```
cookie=0xf005ba1100000004, table=222,
priority=260,
actions=encap(nsh),move:regX[8..31]->nsp,move:regX[0..7]->nsi,load:0->regX,
    move:NXM_NX_REG0[]->NXM_NX_NSH_C1[],
    move:NXM_NX_TUN_ID[0..31]->NXM_NX_NSH_C2[],
    move:NXM_NX_REG6[]->NXM_NX_NSH_C4[],load:0->NXM_NX_TUN_ID[0..31],
    goto_table:223
```

The transport egress flow forwards the packet appropriately. If the service path transport is standard vxlan, an Ethernet header is added:

```
cookie=0xf005ba1100000005, table=223,
priority=260,packet_type=(1,0x894f),nsp=<NSP>
actions=encap(ethernet),load:<SFF IP>->NXM_NX_TUN_IPV4_DST,output:1
```

Otherwise, if the service path uses vxlan-gpe, the Ethernet header is not added:

```
cookie=0xf005ba1100000005, table=223,
priority=260,packet_type=(1,0x894f),nsp=<NSP>
actions=load:<SFF IP>->NXM_NX_TUN_IPV4_DST,output:1
```

Finally, the SFF might be local to the classifier. The SFC service most likely is not bound to the ingress port, so resubmit to SFC service directly:

```
cookie=0xf005ba1100000005, table=223,
priority=260,packet_type=(1,0x894f),nsp=<NSP>
actions=resubmit(,83)
```

Yang changes

This feature will not introduce any changes to the SFC Yang data model.

Configuration impact

The SFC configuration API will not need to be changed for this feature.

Clustering considerations

There will be no clustering impacts as a result of this feature.

Other Infra considerations

The SFC infrastructure will no longer need to use the branched version of OVS, called the Yi Yang patch, which was based on OVS 2.6. The infrastructure will now need to use OVS 2.9, and a suitable version of Linux.

Security considerations

There are no additional security considerations as a result of this feature.

Scale and Performance Impact

The changes to the SFC pipeline will be minimal, so no scaling nor performance impacts will be introduced.

Targeted Release

This feature is targeted for Fluorine.

Alternatives

The only alternative is to stay with the older branched version of OVS, which is not ideal, since we should always strive to use official versions of upstream projects, which this feature will do.

3.6.3 Usage

How will end user use this feature? Primary focus here is how this feature will be used in an actual deployment.

This section will be primary input for Test and Documentation teams. Along with above this should also capture REST API and CLI.

Features to Install

odl-sfc-openflow-renderer

Identify existing karaf feature to which this change applies and/or new karaf features being introduced. These can be user facing features which are added to integration/distribution or internal features to be used by other projects.

REST API

Sample JSONS/URIs. These will be an offshoot of yang changes. Capture these for User Guide, CSIT, etc.

CLI

There will not be any CLI changes as a result of this feature.

3.6.4 Implementation

Assignee(s)

Primary assignee: Brady Johnson, IRC: bjohnson, bjohnson@inocybe.com

Other contributors: Jaime Caamaño, IRC: jaicaa, jcaamano@suse.com

Work Items

Break up work into individual items. This should be a checklist on a Trello card for this feature. Provide the link to the trello card or duplicate it.

3.6.5 Dependencies

Any dependencies being added/removed? Dependencies here refers to internal [other ODL projects] as well as external [OVS, karaf, JDK etc]. This should also capture specific versions if any of these dependencies. e.g. OVS version, Linux kernel version, JDK etc.

This should also capture impacts on existing projects that depend on SFC.

Following projects currently depend on SFC: GBP Netvirt

3.6.6 Testing

Capture details of testing that will need to be added.

Unit Tests

Integration Tests

CSIT

3.6.7 Documentation Impact

The SFC OpenFlow pipeline will be updated in the User Guide as a result of the changes for this new feature.

3.6.8 References

[1] Network Service Headers RFC

[2] https://specs.openstack.org/openstack/nova-specs/specs/kilo/template.html

---

Note: This template was derived from [2], and has been modified to support our project.
