Genius : Design doc

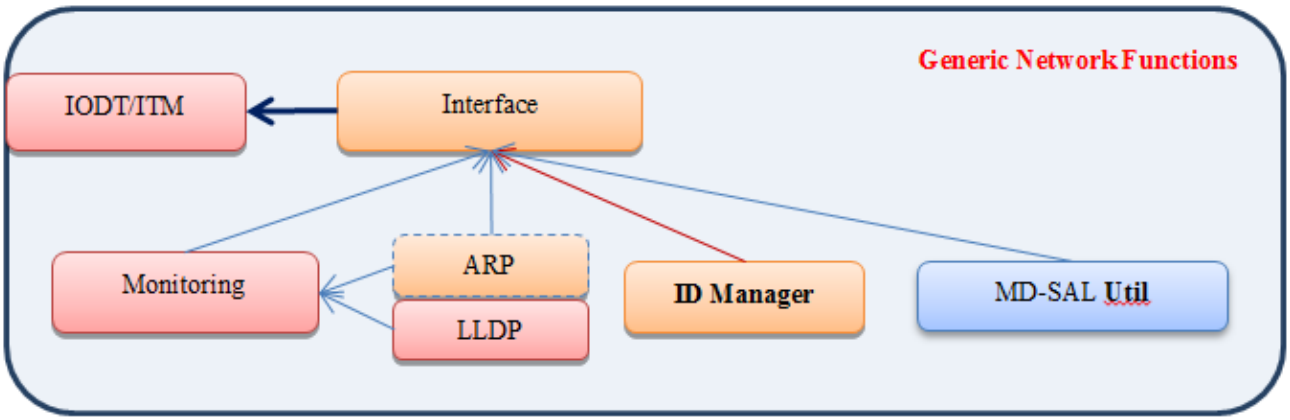
Contents

Design Overview

Genius contains four main components:

1. Interface Manager
2. Tunnel Manager
3. Aliveness Monitor
4. ID-Manager

these modules are developed as karaf features which can be independently installed. However, there is some dependency among these modules. The diagram below provides a dependency relationship of these modules.



the central/main component of Genius is the Interface Manager which uses other modules for its operations. All these modules expose Yang based API which can be used to configure/interact with these modules and fetch services provided by these modules. Thus all these modules can be used/configured by other ODL modules and can also be accessed via REST interface.

Following sections provide details about each of these components.

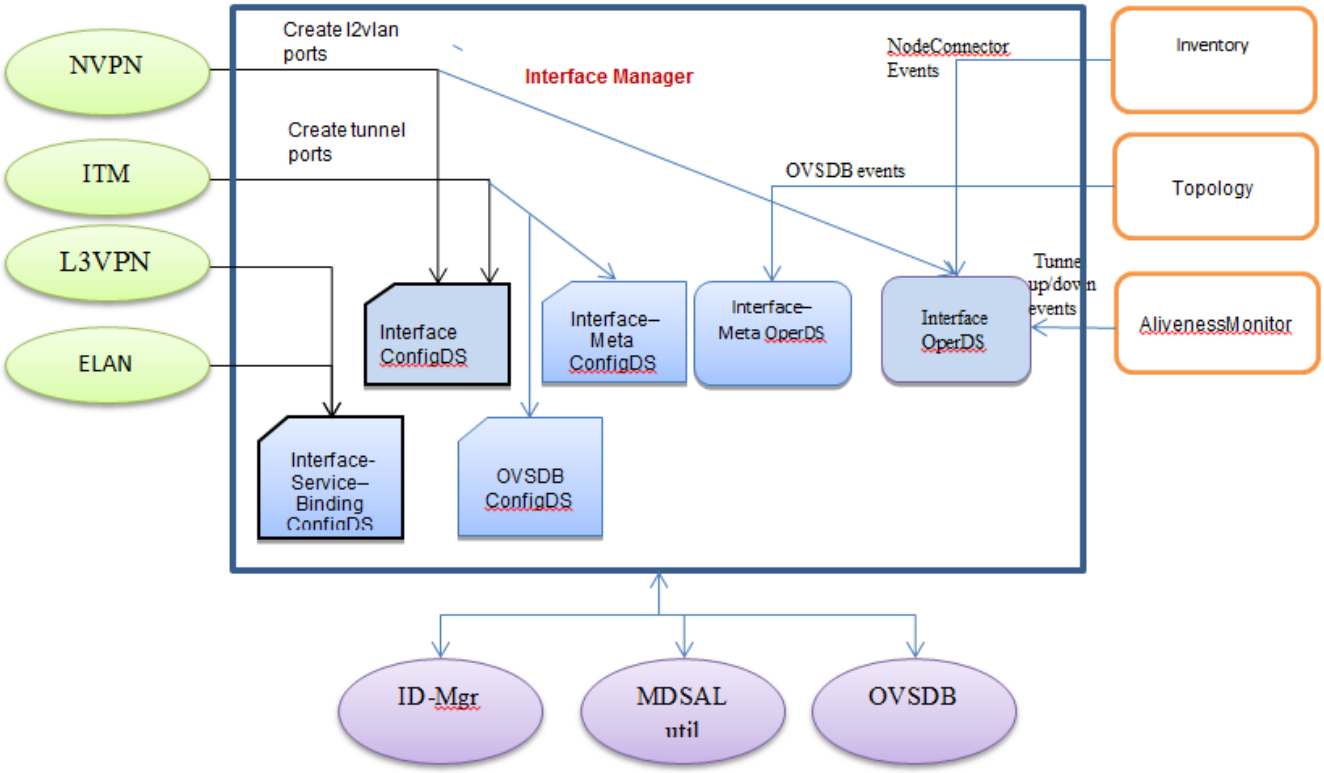
Interface Manager

The Interface Manager (IFM) uses an MD-SAL based architecture, where different software components operate on, and interact via a set of data-models. Interface manager defines configuration data-stores where other opendaylight modules can write interface configurations and register for services. These configuration data-stores can also be accessed by external entities through REST interface. IFM listens to changes in these config data-stores and accordingly programs the data-plane. Data in Configuration data-stores remains persistent across controller restarts.

Operational data like network state and other service specific operational data are stored in operational data-stores. Change in network state is updated in southbound interfaces (OFplugin, OVSDB) data-stores. IFM listens to these updates and accordingly updates its own operational data-stores. Operational data stores are cleaned up after a controller restart.

Additionally, IFM also provides a set of RPCs to access IFM data-stores and provide other useful information. Following figure presents different IFM data-stores and its interaction with other modules.

Follwoing diagram provides a toplevel architecture of Interface Manager.



In addition to these datamodels, it also implements several RPCs for accessing interface operational data. Details of these datamodels and RPCs are described in following sections. Interface Manager also uses ODL Inventory and Topology datastores to retrive southbound configurations and events. As described above Interface Manager uses other Genius modules for its operations. It mainly interacts with following other modules-

Modules used by InterfaceManager

1. **Id Manager** – For allocating dataplane interface-id (if-index)
2. **Aliveness Monitor** - For registering the interfaces for monitoring
3. **MdSalUtil** – For interactions with MD-SAL and other openflow operations

Code structure

Interface manager code is organized in following folders -

1. **interfacemanager-api** contains the interface yang data models and corresponding interface implementation.
2. **interfacemanager-impl** contains the interfacemanager implementation
3. **interface-manager-shell** contains Karaf CLI implementation for interfacemanager

interfacemanager-api

```
main
   java
      org
          opendaylight
              genius
                  interfacemanager
                      exceptions
                      globals
                      interfaces
   yang
```

interfacemanager-impl

```
commons                              <--- contains common utility functions
listeners                            <--- Contains interfacemanager DCN listenenrs for differnt MD-SAL datastores
renderer                             <--- Contains different southbound renderers' implementation
   hwvtep                               <--- HWVTEP specific renderer
      confighelpers
      statehelpers
      utilities
   ovs                                  <--- OVS specific SBI renderer
       confighelpers
       statehelpers
       utilities
servicebindings                      <--- contains interface service binding DCN listener and corresponding
implementation
   flowbased
       confighelpers
       listeners
       statehelpers
       utilities
rpcservice                           <--- Contains interfacemanager RPCs' implementation
pmcounters                           <--- Contains PM counters gathering
statusanddiag                        <--- contains status and diagnostics implementations
```

'*interfacemanager-shell*

Data-model

InterfaceManager mainly uses two Yang data models to accept configurations.

1. **odl-interface** datamodel () where verious type of interface can be configuted.
2. **service-binding** datamodel () where different applications can bind services to interfaces.

In addition to these datamodels, it also implements several RPCs for accessing interface operational data. Details of these datamodels and RPCs are described in following sections. Interface Manager also uses ODL Inventory and Topology datastores to retrive southbound configurations and events. As described above Interface Manager uses other Genius modules for its operations. It mainly interacts with following other modules-

Interface Config DS

IFM datamodel is defined in odl-interface.yang . It is based on 'ietf-interfaces' datamodel (imported in odl_interface.yang) with additional augmentations to it. Common interface configurations are –

- **name (string)** : this is the unique interface name/identifier.
- **type (identityref:iana-if-type)** : this configuration sets the interface type. Interface types are defined in iana-if-types data model. Odl-interfaces. yang data model adds augmentations to iana-if-types to define new interface types. Currently supported interface types are -
  - **l2vlan** (trunk, vlan classified sub-ports/trunk-member)
  - **tunnel** (OVS based VxLAN, GRE, MPLSoverGRE/MPLSoverUDP)
- **enabled (Boolean)** : this configuration sets the administrative state of the interface.
- **parent-refs :** this configuration specifies the parent of the interface, which feeds data/hosts this interface. It can be a physical switch port or a virtual switch port.
  - Parent-interface (string) : is the port name with which a network port in dataplane in that appearing on the southbound interface. E.g. neutron port. this can also be another interface, thus supporting a hierarchy of linked interfaces.
  - Node-identifier (topology_id, node_id) : is used for configuring parent node for HW nodes/VTEPs

Additional configuration parameters are defined for specific interface type. Please see the table below.

Vlan-xparent	**Vlan-trunk**	**Vlan-trunk-member**	**vxlan**	**gre**
Configuration Data				
Name =uuid	**Name** =uuid	**Name** =uuid	**Name** =uuid	**Name** =uuid
description	description	description	description	description
Type =l2vlan	**Type** =l2valn	**Type** =l2vlan	**Type** =tunnel	**Type** =tunnel
enabled	**enabled**	**enabled**	**enabled**	**enabled**

Parent-if =port-name	**Parent-if** =port-name	**Parent-if** =vlan-trunkIf	Vlan-id	Vlan-id
vlan-mode =transparent	**vlan-mode =**trunk	**vlan-mode =**trunk-member	**tunnel-type** = vxlan	**tunnel-type** = gre
	vlan-list= [trunk-member-list]	**Vlan-Id** = trunk-vlanId	dpn-id	dpn-id
		Parent-if =vlan-trunkIf	Vlan-id	Vlan-id
			local-ip	**local-ip**
			remote-ip	**remote-ip**
			gayeway-ip	**gayeway-ip**

Interface service binding config

Yang Data Model odl-interface-service-bindings.yang contains the service binding configuration daatmodel.

An application can bind services to a particular interface by configuring MD-SAL data node at path /config/interface-service-binding. Binding services on interface allows particular service to pull traffic arriving on that interface, depending upon the a service priority. Service modules can specify openflow-rules to be applied on the packet belonging to the interface. Usually these rules include sending the packet to specific service table/pipeline. Service modules /applications are responsible for sending the packet back (if not consumed) to service dispatcher table, for next service to process the packet. Following are the service binding parameters –

- **interface-name** is name of the interface to which service binding is being configured
- **Service-Priority** parameter is used to define the order in which the packet will be delivered to different services bind to the particular interface.
- Service-Name
- **Service-Info** parameter is used to configure flow rule to be applied to the packets as needed by services/applications.
    - (for service-type openflow-based)
    - Flow-priority
    - Instruction-list

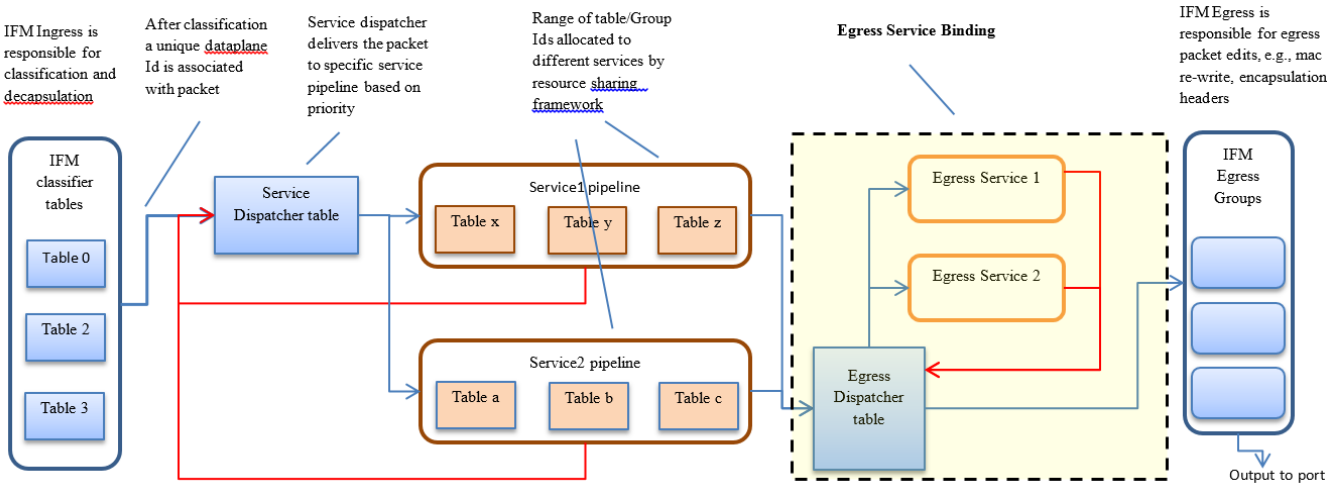
When a service is bind to an interface, Interface Manager programs the service dispatcher table with a rule to match on the interface data-plane-id and the service-index (based on priority) and the instruction-set provided by the service/application. Every time when the packet leaves the dispatcher table the service-index (in metadata) is incremented to match the next service rule when the packet is resubmitted back to dispatcher table. Following table gives an example of the service dispatcher flows, where one interface is bind to 2 services.

Service Dispatcher Table	
Match	**Actions**
• if-index = I • ServiceIndex = 1	• Set SI=2 in metadata • service specific actions <e.g., Goto prio 1 Service table>
• if-index = I • ServiceIndex = 2	• Set SI=3 in metadata • service specific actions <e.g., Goto prio 2 Service table>
miss	Drop

Interface Manager programs openflow rules in the service dispatcher table.

Introducing Egress Service Binding

Currently Interface Manager supports service binding on port ingress only. However, there are services that need packet processing on the egress, before sending the packet out to particular port/interface. To accommodate this, interface manager will be enhanced to support egress service binding also. This will be achieved by introducing a new “egress dispatcher table” at the egress of packet pipeline before the interface egress groups.



On different application request, Interface Manager returns the egress actions for interfaces. Service modules program use these actions to send the packet to particular interface. Generally, these egress actions include sending packet out to port or appropriate interface egress group. With the inclusion of the egress dispatcher table the egress actions for the services would be to

- Update REG1
  - Set service_index =0
- Update REG1 = egress if_index
  - send the packet to Egress Dispatcher table

IFM shall add a default table entry for each interface which

- Match on if_index with REG1
- Send to output port or Egress group.

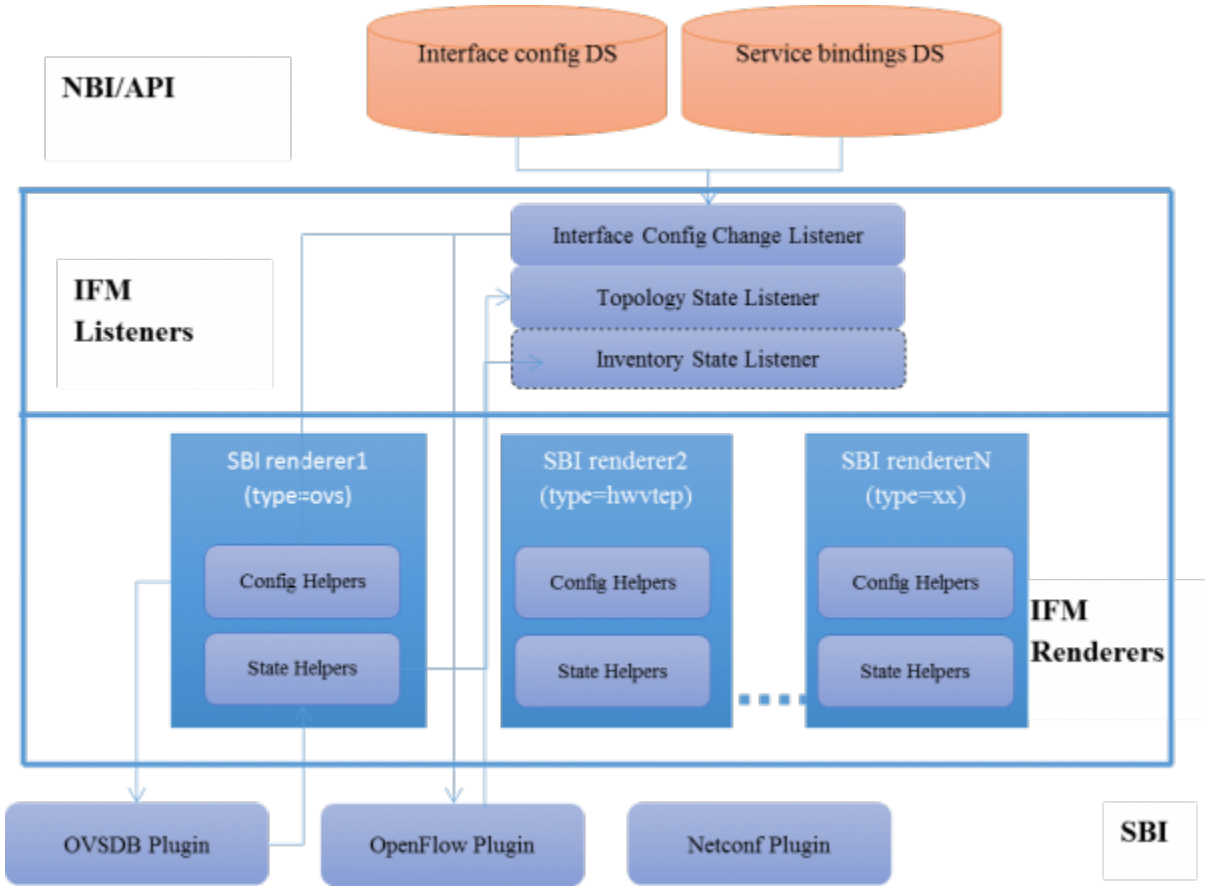
On Egress Service binding, IFM shall add rules to Egress Dispatcher table with following parameters –

- Match on
  - ServiceIndex=egress Service priority
  - if_index in REG1 = if_index for egress interface
- Actions
  - Increment service_index
  - Actions provided by egress service binding.

Egress Services will be responsible for sending packet back to Egress Dispatcher table, if the packet is not consumed (dropped/ send out). In this case the packet will hit the lowest priority default entry and the packet will be send out.

Interface Manager modules

Interface manager is designed in a modular fashion to provide a flexible way to support multiple southbound protocols. North-bound interface/data-model is decoupled from south bound plugins. NBI Data change listeners select and interact with appropriate SBI renderers. The modular design also allows addition of new renderers to support new southbound interfaces, protocols plugins. Following figure presents interface manager design –



Threading Model and Clustering Considerations

InterfaceManager uses the datastore job coordinator module for all its operations. Datastore job coordinator solves the following problems which is observed in the previous Li-based interface manager :

1. The Business Logic for the Interface configuration/state handling is performed in the Actor Thread itself.
2. This will cause the Actor's mailbox to get filled up and may start causing unnecessary back-pressure.
3. Actions that can be executed independently will get unnecessarily serialized.
4. Can cause other unrelated applications starve for chance to execute.
5. Available CPU power may not be utilized fully. (for instance, if 1000 interfaces are created on different ports, all 1000 interfaces creation will happen one after the other.)
6. May depend on external applications to distribute the load across the actors.

Datastore Job Coordination framework

The datastore job coordinator framework offers the following benefits :

1. "Datastore Job" is a set of updates to the Config/Operational Datastore.
2. Dependent Jobs [eg. Operations on interfaces on same port] that need to be run one after the other will continue to be run in sequence.
3. Independent Jobs [eg. Operations on interfaces across different Ports] will be allowed to run paralelly.
4. Makes use of ForkJoin Pools that allows for work-stealing across threads. ThreadPool executor flavor is also available… But would be deprecating that soon.
5. Jobs are enqueued and dequeued to/from a two-level Hash structure that ensures point 1 & 2 above are satisfied and are executed using the ForkJoinPool mentioned in point 3.
6. The jobs are enqueued by the application along with an application job-key (type: string). The Coordinator dequeues and schedules the job for execution as appropriate. All jobs enqueued with the same job-key will be executed sequentially.
7. DataStoreJob Coordination to distribute jobs and execute them paralelly within a single node.
8. This will still work in a clustered mode by handling optimistic lock exceptions and retrying of the job.
9. Framework provides the capability to retry and rollback Jobs.
10. Applications can specify how-many retries and provide callbacks for rollback.
11. Aids movement of Application Datastore listeners to "Follower" also listening mode without any change to the business logic of the application.
12. Datastore Job Coordination function gets the list of listenable futures returned from each job.
13. The Job is deemed complete only when the onSuccess callback is invoked and the next enqueued job for that job-key will be dequeued and executed.
14. On Failure, based on application input, retries and/or rollback will be performed. Rollback failures are considered as double-fault and system bails out with error message and moves on to the next job with that Job-Key.

IFM Listeners

IFM listeners listen to different MD-SAL data-stores. On the NBI side it implements data change listeners for interface config data-store and the service-binding data store. On the SBI side IFM implements listeners for Topology and Inventory data-stores in opendaylight.

Interface Config change listener

Interface config change listener listens to ietf-interface/interfaces data node.

service-binding change listener

Interface config change listener listens to ietf-interface/interfaces data node.

Topology state change listener

Interface config change listener listens to ietf-interface/interfaces data node.

inventory state change listener

+++ this page is under construction +++

Dynamic Behavior

when a l2vlan interface is configured

1. Interface ConfigDS is populated
2. Interface DCN in InterfaceManager does the following :
    a. Add interface-state entry for the new interface along with if-index generated
    b. Add ingress flow entry
    c. If it is a trunk VLAN, need to add the interface-state for all child interfaces, and add ingress flows for all child interfaces

when a tunnel interface is configured

1. Interface ConfigDS is populated
2. Interface DCN in InterfaceManager does the following :
    a. Creates bridge interface entry in odl-interface-meta Config DS
    b. Add port to Bridge using OVSDB
        i. retrieves the bridge UUID corresponding to the interface and
        ii. populates the OVSDB Termination Point Datastore with the following information

```
tpAugmentationBuilder.setName(portName);
tpAugmentationBuilder.setInterfaceType(type);
options.put("key", "flow");
options.put("local_ip", localIp.getIpv4Address().getValue());
options.put("remote_ip", remoteIp.getIpv4Address().getValue());
tpAugmentationBuilder.setOptions(options);
```

OVSDB plugin acts upon this data change and configures the tunnel end points on the switch with the supplied information.

NodeConnector comes up on vSwitch

1. Inventory DCN Listener in InterfaceManager does the following:
    a. Updates interface-state DS.
    b. Generate if-index for the interface
    c. Update if-index to interface reverse lookup map
    d. If interface maps to a vlan trunk entity, operational states of all vlan trunk members are updated
    e. If interface maps to tunnel entity, add ingress tunnel flow

Bridge is created on vSWitch

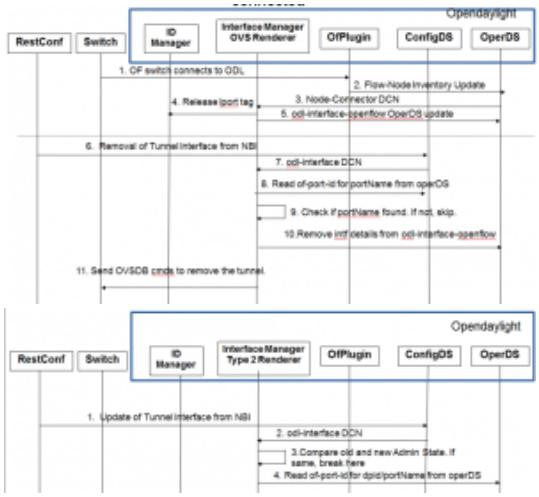
1. Topology DCN Listener in InterfaceManager does the following:
    a. Update odl-interface-meta OperDS to have the dpid to bridge reference
    b. Retrieve all pre provisioned bridge Interface Entries for this dpn, and add ports to bridge using ovsdb

ELAN/VPNManager does a bind service

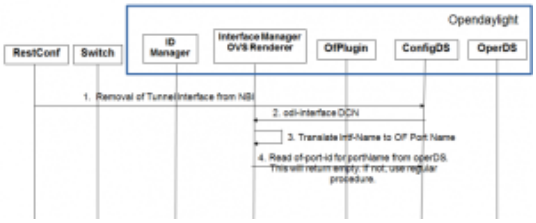
1. Interface service-bindings config DS is populated with service name, priority and lport dispatcher flow instruction details
2. Based on the service priority, the higher priority service flow will go in dispatcher table with match as if-index
3. Lower priority service will go in the same lport dispatcher table with match as if-index and service priority

Interface Manager Sequence Diagrams

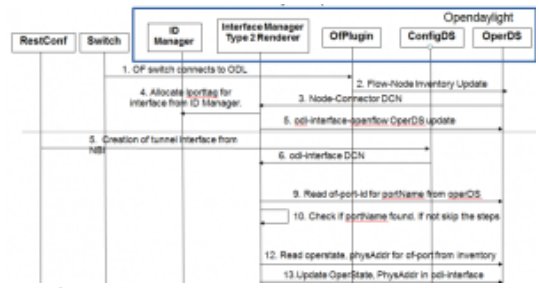
Following gallery contains sequence diagrams for different IFM operations -



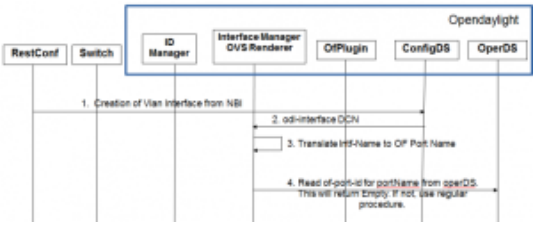
Removal of Tunnel Interface When OF Switch is Connected



Removal of Tunnel Interfaces in Pre provisioning Mode

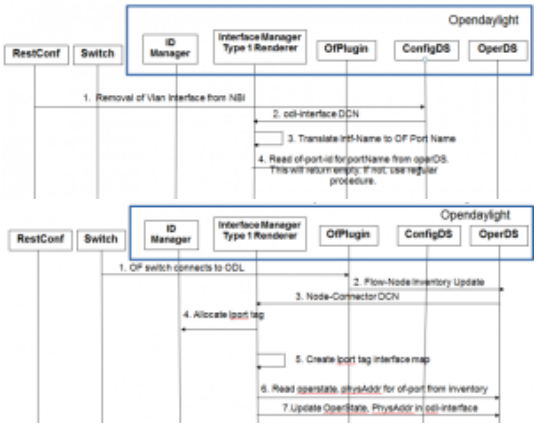


Creation of tunnel-interface when OF switch is connected and PortName already in OperDS

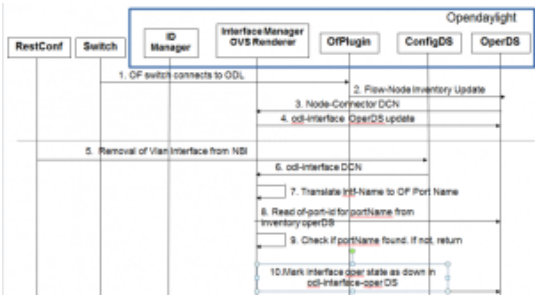


Creation of vlan interface in pre provisioning mode

Creation of vlan interface when switch is connected

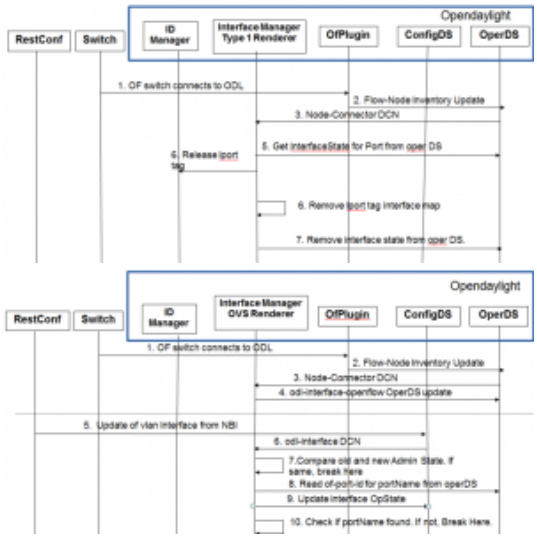


deletion of vlan interface in pre provisioning mode

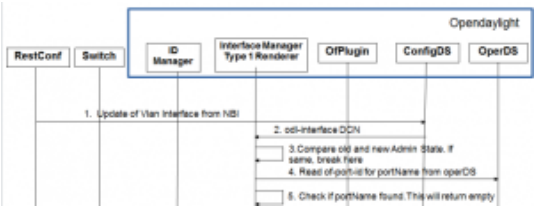


deletion of vlan interface when switch is connected

node connector added_updated DCN handling



node connector removed DCN handling



updation of vlan interface in pre provisioning mode

updation of vlan interface when switch is connected