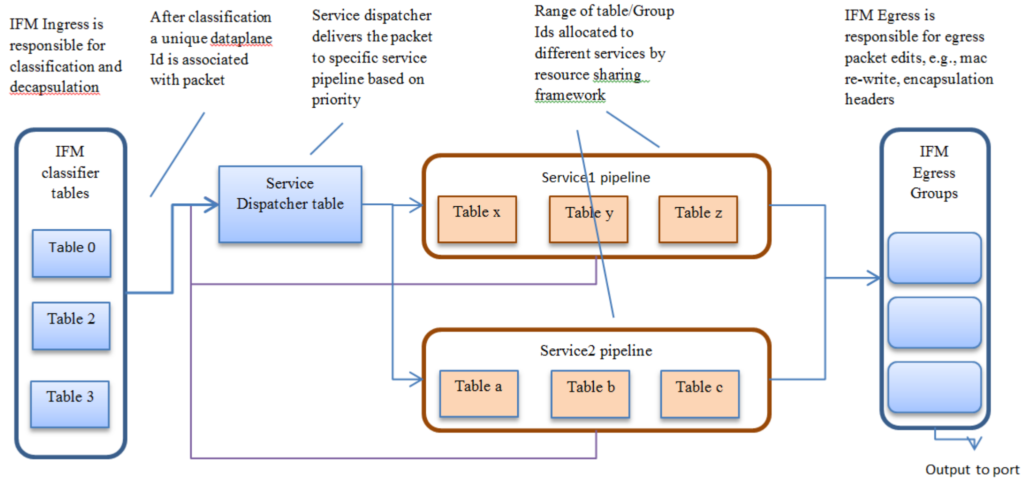
Genius User Guide

Contents

Genius Services

Genius provides components and framework for application co-existence. Following diagram gives an overview of how multiple applications/service based on genius components, can work together.


IFM Ingress is responsible for classification and decapsulation
After classification a unique dataplane Id is associated with packet
Service dispatcher delivers the packet to specific service pipeline based on priority
Range of table/Group Ids allocated to different services by resource sharing framework
IFM Egress is responsible for egress packet edits, e.g., mac re-write, encapsulation headers
IFM classifier tables
Table 0
Table 2
Table 3
Service Dispatcher table
Service1 pipeline
Table x
Table y
Table z
Service2 pipeline
Table a
Table b
Table c
IFM Egress Groups
Output to port

Genius addresses app co-existence issue based on following principles –

- Different applications use isolated set of openflow resources (tables, groups etc.)
  1. Achieved using Resource Sharing/Management framework
- Traffic ingress and egress is handled by a common entity serving multiple applications
  1. Achieved using Interface Manager/ Overlay Tunnel Manager

Interface Management Service

Genius Interface Manager (IFM) provides following main services --

1. Ingress Classification & decapsulation
2. Service Bindings
3. Egress encapsulation

Interface Classification & decapsulation

Interface Manager allows creation of different granular logical ports/ interfaces for different type of network connections and logical ports, e.g. VLAN ports (classified with in-port+VLAN tag), tunnel (VxLAN & GRE) interfaces classified with ([DPN-id]+[in-port]+[vlan]+local-ip+ remote-ip).

Interface Manager provides Yang based API for creation/configuration of different type of interfaces. Each interface is created with a unique interface-name (e.g. UUID of a neutron port) and its classification parameters. Interface configuration also contains a 'parent-ref' parameter which defines the parent port/interface which feeds to the child interface. This way a hierarchy of interfaces can be created. IFM classification tables in data-plane also form similar hierarchy.

Interface Manager Operations

When the parent (openflow) port appears on the south-bound interface, Interface Manager accordingly programs corresponding IFM classifier tables and groups.

When a packet hits table 0, IFM classification rules in table 0 removes the encapsulation/tags and direct packets to appropriate next level of classification table. Once the packet is classified belonging to a particular Interface, corresponding data-plane-id is associated with the packet and the packet is sent to the service dispatcher table.

Service dispatcher table, depending upon the services bind to the interface, apply service specific parameters to the packet and dispatches the packet to appropriate service pipeline. If the packet is not consumed (either dropped or sent out) by the particular service, the service pipeline must send(resubmit) the packet back to the dispatcher table for other services for further processing.

Metadata usage

First 24 bits of metadata is reserved by Interface Manager to carry the 21 bit data-plane-id (if-index) along with 3 bits service-index. Applications/ services must preserve these bits for use of interface manger when the packet is sent back to dispatcher table. If a particular service is sure of consuming the packet and packet is not required to be sent back to dispatcher table, in that case the service is allowed to make use of full 64 bits of the metadata.

Interfaces Data Model

Yang Data Model odl-interface.yang contains the interface configuration data-model.

An interface is created at MD-SAL Data Node Path **/config/if:interfaces/interface**, with following attributes -

Common attributes

- **name** -- Unique Interface name, can be any unique string (e.g., UUID string).
- **type** -- interface type, currently supported *iana-if-type:l2vlan and iana-if-type:tunnel*
- **enabled** -- Admin Status, possible values 'true' or 'false'

parent-refs : used to specify references to parent interface/port feeding to this interface

- **datapath-node-identifier** -- Identifier for a fixed/physical dataplane node, can be physical switch identifier
- **parent-interface** -- can be a physical switch port (in conjunction of above), virtual switch port (e.g., neutron port) or another interface
- **list node-identifier** -- Identifier of the dependant underlying configuration protocol
    1. *topology-id* -- can be ovsdb configuration protocol
    2. *node-id* -- can be hwvtep node-id

Type specific attributes

- when type = l2vlan
    1. **vlan-id** -- VLAN id for trunk-member l2vlan interfaces
    2. **l2vlan-mode** -- Currently supported ones are 'transparent', 'trunk' or 'trunk-member'
- when type = stacked_vlan (Not supported yet)
    1. **stacked-vlan-id** -- VLAN-Id for additional/second VLAN tag
- when type = tunnel
    1. **tunnel-interface-type** -- Tunnel type, currently supported ones are tunnel-type-vxlan, tunnel-type-gre or tunnel-type-mpls-over-gre
    2. **tunnel-source** -- tunnel source IP address
    3. **tunnel-destination** -- tunnel destination IP address
    4. **tunnel-gateway** -- gateway IP address
    5. **monitor-enabled** -- Tunnel monitoring enable control
    6. **monitor-interval** -- Tunnel monitoring interval in millisiconds
- when type = mpls (Not supported yet)
    1. **list labelStack** -- list of lables
    2. **num-labels** -- number of lables configured

Supported REST calls are **GET, PUT, DELETE, POST**

Creating L2 port interfaces

A normal L2 port (e.g. Neutron tap port) interface is created as an 'l2vlan' interface in 'transparent' mode. This type of interfce classifies packets passing through a particular L2 (OpenFlow) port. The base openflow port for this interface is configured port is interface can pass any L2 packet. In dataplane Packets for this interface are classified by matching of-port-id assigned to the port (as specified in parent-interface).

URL: /restconf/config/ietf-interfaces:interfaces

Sample JSON data

```
   "interfaces": {
       "interface": [
           {
               "name": "4158408c-942b-487c-9a03-0b603c39d3dd",
               "type": "iana-if-type:l2vlan",                       <--- interface type 'l2vlan' for normal L2
port
               "odl-interface:l2vlan-mode": "transparent",          <--- 'transparent' VLAN port mode allows any
(tagged, untagged) ethernet packet
               "odl-interface:parent-interface": "tap4158408c-94",  <--- port-name as it appears on southbound
interface
               "enabled": true
           }
       ]
   }
```

Creating VLAN interfaces

A VLAN interface is created as a 'l2vlan' interface in 'trunk-member' mode, by configuring a VLAN-Id and a particular L2 (vlan trunk) interface. Parent VLAN trunk interface is created in the same way as the 'transparent' interface as specified above. A 'trunk-member' interface defines a flow on a particular L2 port and having a particular VLAN tag. On ingress, after classification the VLAN tag is popped out and corresponding unique dataplane-id is associated with the packet, before delivering the packet to service processing. When a service module delivers the packet to this interface for egress, it pushes corresponding VLAN tag and sends the packet out of the parent L2 port.

URL: /restconf/config/ietf-interfaces:interfaces

Sample JSON data

```
"interfaces": {
       "interface": [
           {
               "name": "4158408c-942b-487c-9a03-0b603c39d3dd:100",
               "type": "iana-if-type:l2vlan",
               "odl-interface:l2vlan-mode": "trunk-member",        <--- for 'trunk-member', flow is classified
with particular vlan-id on an l2 port
               "odl-interface:parent-interface": "4158408c-942b-487c-9a03-0b603c39d3dd",  <--- Parent 'trunk'
iterface name
               "odl-interface:vlan-id": "100",
               "enabled": true
           }
       ]
   }
```

Creating Overlay Tunnel Interfaces

An overlay tunnel interface is created with type 'tunnel' and particular 'tunnel-interface-type'. Tunnel interfaces are created on a particular DPN with a pair of (local, remote) IP addresses. Currently supported tunnel interface types are VxLAN, GRE and MPLSoverGRE. overlay tunnel interfaces are based on tunnel interafces provided by OVS. Some of these interfaces are not yet available with standard OVS versions. However, there are OVS patches available add specific tunnel interface implementation in OVS. Following are tunnel types for which specific patches are available -

URL: /restconf/config/ietf-interfaces:interfaces

Sample JSON data

```
"interfaces": {
       "interface": [
           {
               "name": "MGRE_TUNNEL:1",
               "type": "iana-if-type:tunnel",
               "odl-interface:tunnel-interface-type": "odl-interface:tunnel-type-mpls-over-gre",
               "odl-interface:datapath-node-identifier": 156613701272907,
               "odl-interface:tunnel-source": "11.0.0.43",
               "odl-interface:tunnel-destination": "11.0.0.66",
               "odl-interface:monitor-enabled": false,
               "odl-interface:monitor-interval": 10000,
               "enabled": true
           }
       ]
   }
```

Services Binding

After creation of interfaces, different applications can bind services to it. Different services can bind to interfaces on ingress at the packet pipeline ingress as well as at the pipeline egress, before sending the packet out to particular port/interface. To bind a service to an Interface, applications will use a Yang based service binding API provided by interface manager. Following are the service binding parameters –

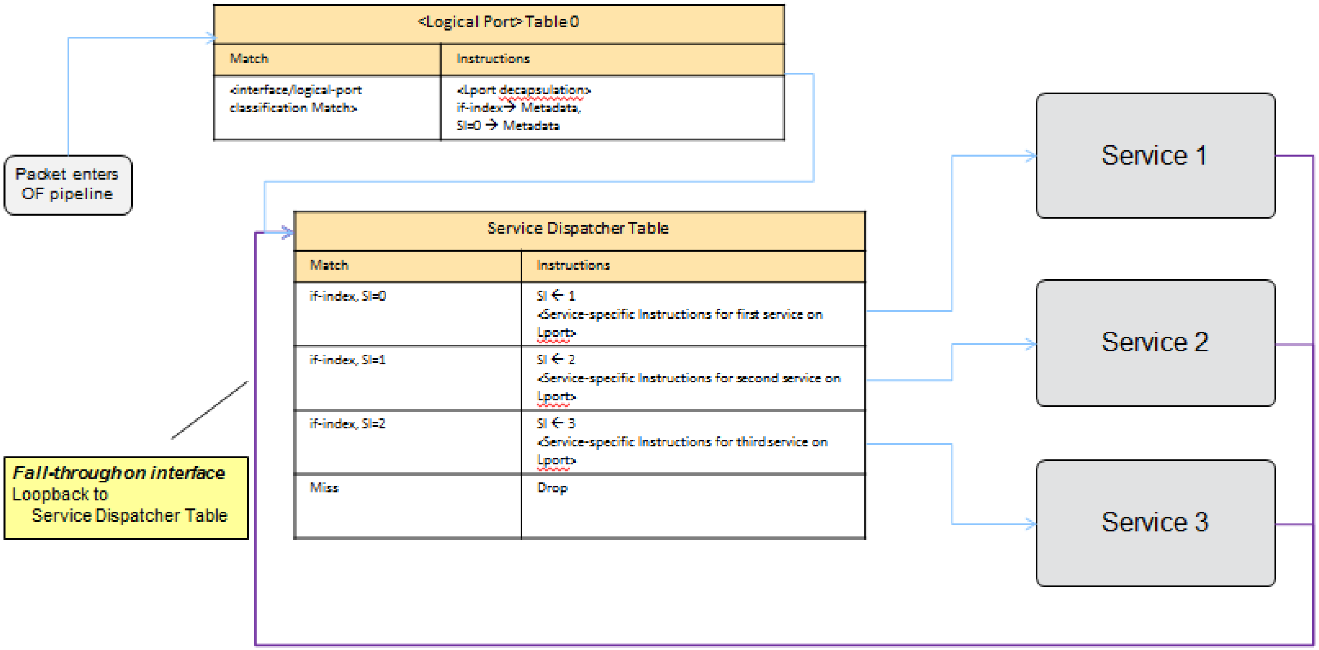
[[File:interface_hierarchy.png]| align:right]

- Service-Priority
- Service-mode(ingress/egress)
- Service-Name
- Service-Info
    1. (for service-type openflow-based)
    2. Flow-priority
    3. Instruction-list

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 priorities must be agreed upon by the participating applications to avoid conflict.

Depending upon the service-mode parameter service is applied to interface ingress on interface egress.

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. if the service-mode is ingress, ingress dispatcher is programmed or else if the service-mode-egress is configured egress-dispatcher table is programmed.



<Logical Port> Table 0

Match	Instructions

<interface/logical-port classification Match>	<Lport decapsulation>
	if-index→ Metadata,
	SI=0 → Metadata

Service Dispatcher Table

Match	Instructions
if-index, SI=0	SI ← 1 <Service-specific Instructions for first service on Lport>
if-index, SI=1	SI ← 2 <Service-specific Instructions for second service on Lport>
if-index, SI=2	SI ← 3 <Service-specific Instructions for third service on Lport>
Miss	Drop

Binding services on interface

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 interafce depending upon the a service priority. Service modules can specify openflow-rules to be applied on the packet belonging to the inetrface. Usually these rules include sending the packet to specific service table/pipeline. Service modules are responsible for sending the packet back (if not consumed) to service dispatcher table, for next service to process the packet.

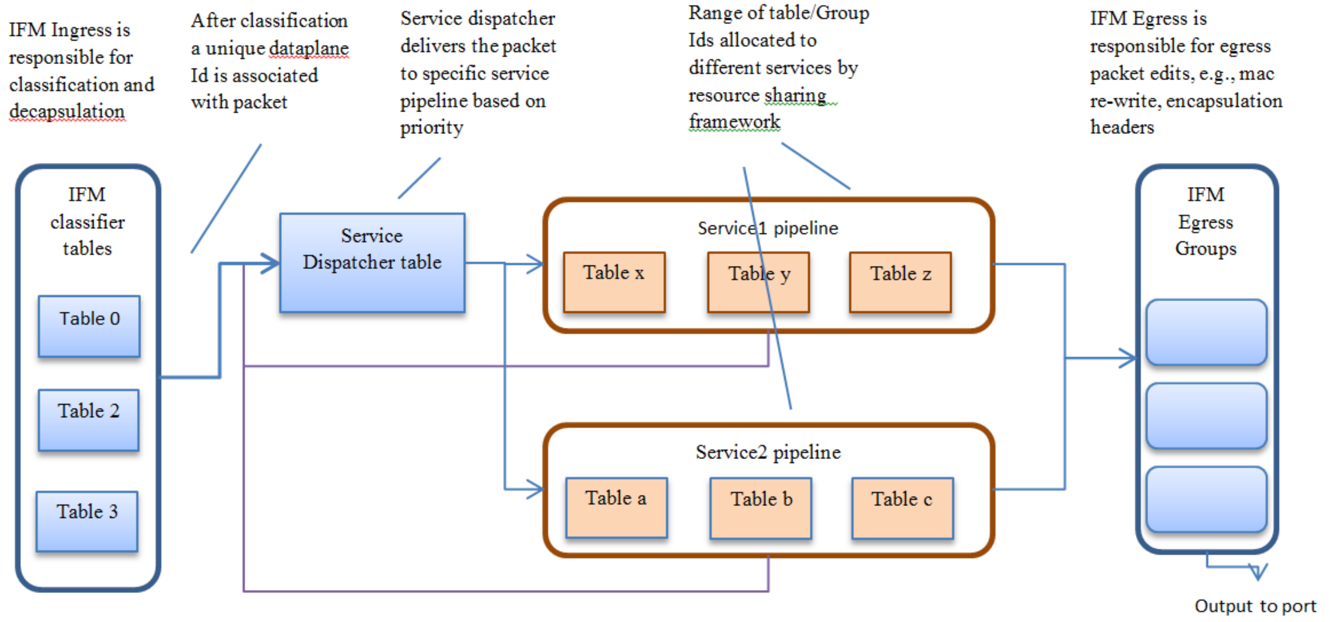
URL:/restconf/config/interface-service-bindings:service-bindings/

Sample JSON data

```
 "service-bindings": {
   "services-info": [
     {
       "interface-name": "4152de47-29eb-4e95-8727-2939ac03ef84",
       "bound-services": [
         {
           "service-name": "ELAN",
           "service-type": "interface-service-bindings:service-type-flow-based"
           "service-priority": 3,
           "flow-priority": 5,
           "flow-cookie": 134479872,
           "instruction": [
             {
               "order": 2,
               "go-to-table": {
                 "table_id": 50
               }
             },
             {
               "order": 1,
               "write-metadata": {
                 "metadata": 83953188864,
                 "metadata-mask": 1099494850560
               }
             }
           ],
         },
         {
          "service-name": "L3VPN",
          "service-type": "interface-service-bindings:service-type-flow-based"
          "service-priority": 2,
          "flow-priority": 10,
          "flow-cookie": 134217729,
          "instruction": [
             {
               "order": 2,
               "go-to-table": {
                 "table_id": 21
               }
             },
             {
               "order": 1,
               "write-metadata": {
                 "metadata": 100,
                 "metadata-mask": 4294967295
               }
             }
           ],
         }
       ]
     }
   ]
 }
```

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.


IFM Ingress is responsible for classification and decapsulation
After classification a unique dataplane Id is associated with packet
Service dispatcher delivers the packet to specific service pipeline based on priority
Range of table/Group Ids allocated to different services by resource sharing framework
IFM Egress is responsible for egress packet edits, e.g., mac re-write, encapsulation headers
IFM classifier tables
Table 0
Table 2
Table 3
Service Dispatcher table
Service1 pipeline
Table x
Table y
Table z
Service2 pipeline
Table a
Table b
Table c
IFM Egress Groups
Output to port

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
    1. Set service_index =0
- Update REG1 = egress if_index
    1. 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
    1. ServiceIndex=egress Service priority
    2. if_index in REG1 = if_index for egress interface
- Actions
    1. Increment service_index
    2. 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 RPCs

In addition to above defined configuration interfaces, IFM also provides several RPCs to access interface operational data and other helpful information. Interface Manger RPCs are defined in odl-interface-rpc.yang Following RPCs are available.

get-dpid-from-interface

This RPC is used to retrieve dpid/switch hosting the root port from given interface name

```
rpc get-dpid-from-interface {
    description "used to retrieve dpid from interface name";
    input {
        leaf intf-name {
            type string;
        }
    }
    output {
        leaf dpid {
            type uint64;
        }
    }
}
```

get-port-from-interface

This RPC is used to retrieve south bound port attributes from the interface name.

```
rpc get-port-from-interface {
     description "used to retrieve south bound port attributes from the interface name";
     input {
         leaf intf-name {
             type string;
         }
     }
     output {
         leaf dpid {
             type uint64;
         }
         leaf portno {
             type uint32;
         }
         leaf portname {
             type string;
         }
     }
 }
```

get-egress-actions-for-interface

This RPC is used to retrieve group actions to use from interface name.

```
 rpc get-egress-actions-for-interface {
     description "used to retrieve group actions to use from interface name";
     input {
         leaf intf-name {
             type string;
             mandatory true;
         }
         leaf tunnel-key {
             description "It can be VNI for VxLAN tunnel ifaces, Gre Key for GRE tunnels, etc.";
             type uint32;
             mandatory false;
         }
     }
     output {
         uses action:action-list;
     }
 }
```

get-egress-instructions-for-interface

This RPC is used to retrieve flow instructions to use from interface name.

```
rpc get-egress-instructions-for-interface {
     description "used to retrieve flow instructions to use from interface name";
     input {
         leaf intf-name {
             type string;
             mandatory true;
         }
         leaf tunnel-key {
             description "It can be VNI for VxLAN tunnel ifaces, Gre Key for GRE tunnels, etc.";
             type uint32;
             mandatory false;
         }
     }
     output {
         uses offlow:instruction-list;
     }
 }
```

get-endpoint-ip-for-dpn

This RPC is used to get the local ip of the tunnel/trunk interface on a particular DPN

```
rpc get-endpoint-ip-for-dpn {
    description "to get the local ip of the tunnel/trunk interface";
    input {
        leaf dpid {
            type uint64;
        }
    }
    output {
        leaf-list local-ips {
            type inet:ip-address;
        }
    }
}
```

get-interface-type

This RPC is used to get the type of the interface(vlan/vxlan or gre)

```
rpc get-interface-type {
description "to get the type of the interface(vlan/vxlan or gre)";
    input {
        leaf intf-name {
            type string;
        }
    }
    output {
        leaf interface-type {
            type identityref {
                base if:interface-type;
            }
        }
    }
}
```

get-tunnel-type

This RPC is used to get the type of the tunnel interface(vxlan or gre).

```
rpc get-tunnel-type {
 description "to get the type of the tunnel interface(vxlan or gre)";
     input {
         leaf intf-name {
             type string;
         }
     }
     output {
         leaf tunnel-type {
             type identityref {
                 base odlif:tunnel-type-base;
             }
         }
     }
 }
```

get-nodeconnector-id-from-interface

This RPC is used to get nodeconnector id associated with an interface.

```
rpc get-nodeconnector-id-from-interface {
 description "to get nodeconnector id associated with an interface";
     input {
         leaf intf-name {
             type string;
         }
     }
     output {
         leaf nodeconnector-id {
             type inv:node-connector-id;
         }
     }
 }
```

get-interface-from-if-index

This RPC is used to get interface associated with an if-index (dataplane interface id)

```
 rpc get-interface-from-if-index {
     description "to get interface associated with an if-index";
         input {
             leaf if-index {
                 type int32;
             }
         }
         output {
             leaf interface-name {
                 type string;
             }
         }
     }
```

create-terminating-service-actions

This RPC is used to create the tunnel termination service table entries

```
rpc create-terminating-service-actions {
 description "create the ingress terminating service table entries";
     input {
          leaf dpid {
              type uint64;
          }
          leaf tunnel-key {
              type uint64;
          }
          leaf interface-name {
              type string;
          }
          uses offlow:instruction-list;
     }
 }
```

remove-terminating-service-actions

This RPC is used to remove the tunnel termination service table entries

```
rpc remove-terminating-service-actions {
 description "remove the ingress terminating service table entries";
     input {
          leaf dpid {
              type uint64;
          }
          leaf interface-name {
              type string;
          }
          leaf tunnel-key {
              type uint64;
          }
     }
 }
```

ID Manager

Genius Id-Manager service provides interface to generate unique integer ID(s) for a particular key String, from a given pool of IDs with configured range. The key and generated ID mapping is persistent across cluster restarts.

ID Manager Data-Model/APIs

Creation of ID-Pool =

generating unique Id for a Key from particular pool

Resource Manager Framework

Resource Sharing Framework is based on Genius ID-Manager service. ID-Manager service manages resource/ID pools and allocates unique range of integer IDs for specific string keys. These key and ID/range mappings are persisted across controller restarts. The idea for resource sharing is to have common resource pools, managed by ID-Manager and allow applications to dynamically allocate a range of resource IDs for their use. The resource manager component of the framework creates resource Id pools for of-tables, groups, meters etc. Further, different applications can request for a number of resource-IDs from these pools. Same range is always returned across controller restarts.

Resource Framework Components

Resource pool config file contains available resource pool configs in a particular format which is understandable by the Resource Manager. Following is an example of Resource pool config file.

```
<poolName>,        <id-start>,         <id-end>
tableIdPool,         10,                 254
GroupIdPool,         100,                 1000
```

Pools should be created with resources available for dynamic allocations. Some of the resources are reserved for Genius Interface Manager, which manages port/interface ingress and egress functions and provides services to service modules.

Resource manager is a thin API wrapper layer on top of ID-Manager. It is responsible for creating different resource pools in ID-Manager, as specified in Resource pool config at the system startup. This is achieved using RPC provided by ID-Manager like following -

```
createIdPool{
  input {
     “pool-name” : “tableIdPool”,
     “low” : “10”,
     “high” : “220”
      }
 }
```

Additionally it exposes following RPCs for resource allocation and release. Following RPCs are provided

```
AllocateIdRange
input {
     “pool-name” : “tableIdPool”,
     “key” : “app1.zone1.table1”,
     “size” : “5” }
Returns --
output {
     “id-list” : “100”, “101”, “102”, “103”, “104”
}


rpc releaseResource {
       input {
            leaf resource-type {
               type identityref{
                   base resource-type-base;
               }
            }
            leaf id-key {
               type string;
            }
       }
}


rpc allocateResource {
       input {
            leaf resource-type {
               type identityref{
                   base resource-type-base;
               }
            }
            leaf id-key {
               type string;
            }
            leaf size {
               type uint32;
            }
       }
       output {
            leaf-list id-values{
               type uint32;
               ordered-by user;
            }
       }
   }
```

```
rpc getAvailableResources{
       input {
            leaf resource-type {
               type identityref{
                   base resource-type-base;
               }
            }
       }
       output{
            leaf total-available-id-count {
               type uint32;
            }
       }
   }


rpc getResourcePool{
       input{
            leaf resource-type {
               type identityref{
                   base resource-type-base;
               }
            }
       }
       output{
            list available-ids{
               uses available-resource-ids;
            }
            uses released-resource-ids;
       }
   }


grouping available-resource-ids {
       leaf start {
            type uint32;
       }
       leaf end {
            type uint32;
       }
   }


grouping released-resource-ids {
       list delayed-resource-entries {
           uses idmgr:delayed-id-entry;
       }
   }
```

ReleaseResource(poolName, key)