Workgroup: RIFT
Internet-Draft: draft-head-rift-auto-evpn-01
Published: 9 July 2021
Intended Status: Standards Track
Expires: 10 January 2022

Authors: J. Head, Ed.   T. Przygienda   W. Lin
         Juniper Networks   Juniper Networks   Juniper Networks

RIFT Auto-EVPN

Abstract

This document specifies procedures that allow an EVPN overlay to be fully and automatically provisioned when using RIFT as underlay by leveraging RIFT's no-touch ZTP architecture.

Status of This Memo

This Internet-Draft will expire on 10 January 2022.

Copyright Notice

Table of Contents

1. Introduction

RIFT is a protocol that focuses heavily on operational simplicity. [RIFT] natively supports Zero Touch Provisioning (ZTP) functionality that allows each node in an underlay network to automatically derive its place in the topology and configure itself accordingly when properly cabled. RIFT can also disseminate Key-Value information contained in Key-Value Topology Information Elements (KV-TIEs) [RIFT-KV]. These KV-TIEs can contain any information and therefore be used for any purpose. Leveraging RIFT to provision EVPN overlays without any need for configuration and leveraging KV capabilities to easily validate correct operation of such overlay without a single point of failure would provide significant benefit to operators in terms of simplicity and robustness of such a solution.

1.1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

2. Design Considerations

EVPN supports various service models, this document defines a method for the VLAN-Aware service model defined in [RFC7432]. Other service models may be considered in future revisions of this document.

Each model has its own set of requirements for deployment. For example, a functional BGP overlay is necessary to exchange EVPN NLRI regardless of the service model. Furthermore, the requirements are made up of individual variables, such as each node's loopback address and AS number for the BGP session. Some of these variables may be coordinated across each node in a network, but are ultimately locally significant (e.g. route distinguishers). Similarly, calculation of some variables will be local only to each device. RIFT contains currently enough topology information in each node to calculate all those necessary variables automatically.

Once the EVPN overlay is configured and becomes operational, RIFT Key-Value TIEs can be used to distribute state information to allow for validation of basic operational correctness without the need for further tooling.

3. System ID

The 64-bit RIFT System ID that uniquely identifies a node as defined in RIFT [RIFT].

4. Fabric ID

RIFT operates on variants of Clos substrate which are commonly called an IP Fabric. Since EVPN VLANs can be either contained within one fabric or span them, Auto-EVPN introduces the concept of a Fabric ID into RIFT.

This section describes an optional extension to LIE packet schema in the form of a 16-bit Fabric ID that identifies a nodes membership within a particular fabric. Auto-EVPN capable nodes MUST support this extension but MAY not advertise it when not participating in Auto-EVPN. A non-present Fabric ID and value of 0 is reserved as ANY_FABRIC and MUST NOT be used for any other purpose.

Fabric ID MUST be considered in existing adjacency FSM rules so nodes that support Auto-EVPN can interoperate with nodes that do not. The LIE validation is extended with following clause and if it is not met, miscabling should be declared:

```
(if fabric_id is not advertised by either node OR
 if fabric_id is identical on both nodes)
   AND
(if auto_evpn_version is not advertised by either node OR
 if auto_evpn_version is identical on both nodes)
```

The appendix details LIE (Appendix A.1.2) and Node-TIE (Appendix A.2.2) schema changes.

5. Auto-EVPN Device Roles

Auto-EVPN requires that each node understand its given role within the scope of the EVPN implementation so each node derives the necessary variables and provides the necessary overlay configuration. For example, a leaf node performing VXLAN gateway functions does not need to derive its own Cluster ID or learn one from the route reflector that it peers with.

5.1. All Participating Nodes

Not all nodes have to participate in Auto-EVPN, however if a node does assume an Auto-EVPN role, it MUST derive the following variables:

IPv6 Loopback Address
Unique IPv6 loopback address used in BGP sessions.

Router ID
The BGP Router ID.

Autonomous System Number
The ASN for IBGP sessions.

Cluster ID
The Cluster ID for Top-of-Fabric IBGP route reflection.

5.2. ToF Nodes as Route Reflectors

This section defines an Auto-EVPN role whereby some Top-of-Fabric nodes act as EVPN route reflectors. It is expected that route reflectors would establish IBGP sessions with leaf nodes in the same fabric. The typical route reflector requirements do not change, however determining which specific values to use requires further consideration. ToF nodes performing route reflector functionality MUST derive the following variables:

IPv6 RR Loopback Address
The source address for IBGP sessions with leaf nodes in case ToF won election for one of the route reflectors in the fabric.

IPv6 RR Acceptable Prefix Range
Range of addresses acceptable by the route reflector to form a IBGP session. This range covers ALL possible IPv6 Loopback Addresses derived by other Auto EVPN nodes in the current fabric and other Auto-EVPN RRs addresses.

5.3. Leaf Nodes

Leaf nodes derive their role from realizing they are at the bottom of the fabric, i.e. not having any southbound adjacencies. Alternately, a node can assume a leaf node if it has only southbound adjacencies to nodes with explicit LEAF_LEVEL to allow for scenarios where RIFT leaves do NOT participate in Auto-EVPN.

Leaf nodes MUST derive the following variables:

IPv6 RR Loopback Addresses
Addresses of the RRs present in the fabric. Those addresses are used to build BGP sessions to the RR.

EVIs
Leaf node derives all the necessary variables to instantiate EVIs with layer-2 and optionally layer-3 functionality.

If a leaf node is required to perform layer-2 VXLAN gateway functions, it MUST be capable of deriving the following types of variables:

Route Distinguisher
The route distinguisher corresponding to a MAC-VRF that uniquely identifies each node.

Route Target
The route target that corresponds to a MAC-VRF.

MAC VRF Name
This is an optional variable to provide a common MAC VRF name across all leaves.

Set of VLANs
Those are VLANs provisioned either within the fabric or allowing to stretch across fabrics.

For each VLAN derived in an EVI the following variables MUST be derived:

VLAN
The VLAN ID.

Name
This is an optional variable to provide a common VLAN name across all leaves.

VNI
The VNI that corresponds to the VLAN ID. This will contribute to the EVPN Type-2 route.

IRB
Optional variables of the IRB for the VLAN if the leaf performs layer-3 gateway function.

If a leaf node is required to perform layer-3 VXLAN gateway functions, it MUST additionally be capable of deriving the following types of variables:

IP Gateway MAC Address
The MAC address associated with IP gateway.

IP Gateway Subnetted Address
The IPv4 and/or IPv6 gateway address including its subnet length.

Type-5 EVPN IP Prefix with ToFs performing gateway functionality can also be derived and will be described in a future version of this document.

6. Auto-EVPN Variable Derivation

As previously mentioned, not all nodes are required to derive all variables in a given network (e.g. a transit spine node may not need to derive any or participate in Auto-EVPN). Additionally, all derived variables are derived from RIFT's FSM or ZTP mechanism so no additional flooding beside RIFT flooding is necessary for the functionality.

It is also important to mention that all variable derivation is in some way based on combinations of System ID, MAC-VRF ID, Fabric ID, EVI and VLAN and MUST comply precisely with calculation methods specified in the Auto-EVPN Variable Derivation section to allow interoperability between different implementations. All foundational code elements such as imports, constants, etc. are also mentioned there.

6.1. Auto-EVPN Version

This section describes extensions to both the RIFT LIE packet and Node-TIE schemas in the form of a 16-bit value that identifies the Auto-EVPN Version. Auto-EVPN capable nodes MUST support this extension, but MAY choose not to advertise it in LIEs and Node-TIEs when Auto-EVPN is not being utilized. The appendix describes LIE (Appendix A.1.1) and Node-TIE (Appendix A.2.1) schema changes in detail.

6.2. MAC-VRF ID

This section describes a variable MAC-VRF ID that uniquely identifies an instance of EVPN instance (EVI) and is used in variable derivation procedures. Each EVPN EVI MUST be associated with a unique MAC-VRF ID, this document does not specify a method for making that association or ensuring that they are coordinated properly across fabric(s).

6.3. Loopback Address

First and foremost, RIFT does not advertise anything more specific than the fabric default route in the southbound direction by default. However, Auto-EVPN nodes MUST advertise specific loopback addresses southbound to all other Auto-EVPN nodes so to establish MP-BGP reachability correctly in all scenarios.

Auto-EVPN nodes MUST derive a ULA-scoped IPv6 loopback address to be used as both the IBGP source address, as well as the VTEP source when VXLAN gateways are required. Calculation is done using the 6-bytes of reserved ULA space, the 2-byte Fabric ID, and the node's 8-byte System ID. Derivation of the System ID varies slightly depending upon the node's location/role in the fabric and will be described in subsequent sections.

6.3.1. Leaf Nodes as Gateways

Calculation is done using the 6-bytes of reserved ULA space, the 2-byte Fabric ID, and the node's 8-byte System ID.

In order for leaf nodes to derive IPv6 loopback addresses, algorithms shown in both auto_evpn_fidsidv6loopback (Figure 24) and auto_evpn_v6prefixfidsid2loopback (Figure 9) are required.

IPv4 addresses MAY be supported, but it should be noted that they have a higher likelihood of collision. The appendix contains the required auto_evpn_fidsid2v4loopback (Figure 23) algorithm to support IPv4 loopback derivation.

6.3.2. ToF Nodes as Route Reflectors

ToF nodes acting as route reflectors MUST derive their loopback address according to the specific section describing the algorithm. Calculation is done using the 6-bytes of reserved ULA space, the 2-byte Fabric ID, and the 8-byte System ID of each elected route reflector.

In order for the ToF nodes to derive IPv6 loopbacks, the algorithms shown in both auto_evpn_fidsidv6loopback (Figure 24) and auto_evpn_fidrrpref2rrloopback (Figure 10) are required.

In order for the ToF derive the necessary prefix range to facilitate peering requests from any leaf, the algorithm shown in "auto_evpn_fid2fabric_prefixes" (Figure 8) is required.

6.3.2.1. Route Reflector Election Procedures

Four Top-of-Fabric nodes MUST be elected as an IBGP route reflector. Each ToF performs the election independently based on system IDs of other ToFs in the fabric obtained via southbound reflection. The route reflector election procedures are defined as follows:

1. ToF node with the highest System ID.
2. ToF node with the lowest System ID.
3. ToF node with the 2nd highest System ID.
4. ToF node with the 2nd lowest System ID.

This ordering is necessary to prevent a single node with either the highest or lowest System ID from triggering changes to route reflector loopback addresses as it would result in all BGP sessions dropping.

For example, if two nodes, ToF01 and ToF02 with System IDs 002c6af5a281c000 and 002c6bf5788fc000 respectively, ToF02 would be elected due to it having the highest System ID of the ToFs (002c6bf5788fc000). If a ToF determines that it is elected as route reflector, it uses the knowledge of its position in the list to derive route reflector v6 loopback address.

The algorithm shown in "auto_evpn_sids2rrs" (Figure 6) is required to accomplish this.

Considerations for multiplane route reflector elections will be included in future revisions.

6.4. Autonomous System Number

Nodes in each fabric MUST derive a private autonomous system number based on its Fabric ID so that it is unique across the fabric.

The algorithm shown in auto_evpn_fid2private_AS (Figure 25) is required to derive the private ASN.

6.5. Router ID

Nodes MUST drive a Router ID that is based on both its System ID and Fabric ID so that it is unique to both.

The algorithm shown in auto_evpn_sidfid2bgpid (Figure 11) is required to derive the BGP Router ID.

6.6. Cluster ID

Route reflector nodes in each fabric MUST derive a cluster ID that is based on its Fabric ID so that it is unique across the fabric.

The algorithm shown in auto_evpn_fid2clusterid (Figure 26) is required to derive the BGP Cluster ID.

6.7. Route Target

Nodes hosting EVPN EVIs MUST derive a route target extended community based on the MAC-VRF ID for each EVI so that it is unique across the network. Route targets MUST be of type 0 as per RFC4360.

For example, if given a MAC-VRF ID of 1, the derived route target would be "target:1"

The algorithm shown in auto_evpn_evi2rt (Figure 12) is required to derive the Route Target community.

6.8. Route Distinguisher

Nodes hosting EVPN EVIs MUST derive a type-0 route distinguisher based on its System ID and Fabric ID so that it is unique per MAC-VRF and per node.

The algorithm shown in auto_evpn_sidfid2rd (Figure 18) is required to derive the Route Distinguisher.

6.9. EVPN MAC-VRF Services

It's obvious that applications utilizing Auto-EVPN overlay services may require a variety of layer-2 and/or layer-3 traffic considerations. Variables supporting these services are also derived based on some combination of MAC-VRF ID, Fabric ID, and other constant values. Integrated Routing and Bridging (IRB) gateway address derivation also leverages a set of constant RANDOMSEEDS (Figure 5) values that MUST be used to provide additional entropy.

In order to ensure that VLAN ID's don't collide, a single deployment SHOULD NOT exceed 3 fabrics with 3 EVIs where each EVI terminate 15 VLANs. The algorithms shown in auto_evpn_fidevivlansvlans2desc (Figure 16) and auto_evpn_vlan_description_table (Figure 15) are required to derive VLANs accordingly. An implementation MAY exceed this, but MUST indicate methods to ensure collision-free derivation and describe which VLANs are stretched across fabrics.

6.9.1. Untagged Traffic in Multiple Fabrics

This section defines methods to derive unique VLAN, VNI, MAC, and gateway address values for deployments where untagged traffic is stretched across multiple fabrics.

6.9.1.1. VLAN

Untagged traffic stretched across multiple fabrics MUST derive VLAN tags based on MAC-VRF ID in conjunction with a constant value of 1 (i.e. MAC-VRF ID + 1).

6.9.1.2. VNI

Untagged traffic stretched across multiple fabrics MUST derive VNIs based on MAC-VRF ID and Fabric ID in conjunction with a constant value. These VNIs MUST correspond to EVPN Type-2 routes.

The algorithm shown in auto_evpn_fidevivid2vni (Figure 14) is required to derive VNIs for Type-2 EVPN routes.

6.9.1.3. MAC Address

The MAC address MUST be a unicast address and also MUST be identical for any IRB gateways that belong to an individual bridge-domain across fabrics. The last 5-bytes MUST be a hash of the MAC-VRF ID and a constant value of 1 that is calculated using the previously mentioned random seed values.

The algorithm shown in auto_evpn_fidevividsid2mac (Figure 22) is required to derive MAC addresses.

6.9.1.4. IPv6 IRB Gateway Address

The derived IPv6 gateway address MUST be from a ULA-scoped range that will account for the first 6-bytes. The next 5-bytes MUST be the last bytes of the derived MAC address. Finally, the remaining 7-bytes MUST be ::0001.

The algorithm shown in auto_evpn_fidevividsid2v6subnet (Figure 21) is required to derive the IPv6 gateway address.

6.9.1.5. IPv4 IRB Gateway Address

The derived IPv4 gateway address MUST be from a RFC1918 range, which accounts for the first octet. The next octet MUST a hash of the MAC-VRF ID and a constant value of 1 that is calculated using the previously mentioned random seed values. Finally, the remaining 2 octets MUST be 0 and 1 respectively.

The algorithm shown in auto_evpn_v4prefixfidevividsid2v4subnet (Figure 19) is required to derive the IPv4 gateway address. It should be noted that there is a higher likelihood of address collisions when deriving IPv4 addresses.

6.9.2. Tagged Traffic in Multiple Fabrics

This section defines methods to derive unique VLAN, VNI, MAC, and gateway address values for deployments where tagged traffic is stretched across multiple fabrics.

6.9.2.1. VLAN

Tagged traffic stretched across multiple fabrics MUST derive VLAN tags based on MAC-VRF ID in conjunction with a constant value of 16 (i.e. MAC-VRF ID + 16).

6.9.2.2. VNI

Tagged traffic stretched across multiple fabrics MUST derive VNIs based on MAC-VRF ID and Fabric ID in conjunction with a constant value. These VNIs MUST correspond to EVPN Type-2 routes.

The algorithm shown in auto_evpn_fidevivid2vni (Figure 14) is required to derive VNIs for Type-2 EVPN routes.

6.9.2.3. MAC Address

The MAC address MUST be a unicast address and also MUST be identical for any IRB gateways that belong to an individual bridge-domain across fabrics. The last 5-bytes MUST be a hash of the MAC-VRF ID and a constant value of 1 that is calculated using the previously mentioned random seed values.

The algorithm shown in auto_evpn_fidevividsid2mac (Figure 22) is required to derive MAC addresses.

6.9.2.4. IPv6 IRB Gateway Address

The derived IPv6 gateway address MUST be from a ULA-scoped range that will account for the first 6-bytes. The next 5-bytes MUST be the last bytes of the derived MAC address. Finally, the remaining 7-bytes MUST be ::0001.

The algorithm shown in auto_evpn_fidevividsid2v6subnet (Figure 21) is required to derive the IPv6 gateway address.

6.9.2.5. IPv4 IRB Gateway Address

The derived IPv4 gateway address MUST be from a RFC1918 range, which accounts for the first octet. The next octet MUST a hash of the MAC-VRF ID and a constant value of 16 that is calculated using the previously mentioned random seed values. Finally, the remaining 2 octets MUST be 0 and 1 respectively.

The algorithm shown in auto_evpn_v4prefixfidevividsid2v4subnet (Figure 19) is required to derive the IPv4 gateway address. It should be noted that there is a higher likelihood of address collisions when deriving IPv4 addresses.

6.9.3. Tagged Traffic in a Single Fabric

This section defines a method to derive unique VLAN, VNI, MAC, and gateway address values for deployments where untagged traffic is contained within a single fabric.

6.9.3.1. VLAN

Tagged traffic contained to a single fabric MUST derive VLAN tags based on MAC-VRF ID and Fabric ID in conjunction with a constant value of 17 (i.e. MAC-VRF ID + Fabric ID + 17).

6.9.3.2. VNI

Tagged traffic contained to a single fabric MUST derive VNIs based on MAC-VRF ID and Fabric ID in conjunction with a constant value. These VNIs MUST correspond to EVPN Type-2 routes.

The algorithm shown in auto_evpn_fidevivid2vni (Figure 14) is required to derive VNIs for Type-2 EVPN routes.

6.9.3.3. MAC Address

The MAC address MUST be a unicast address and also MUST be identical for any IRB gateways that belong to an individual bridge-domain across fabrics. The last 5-bytes MUST be a hash of the MAC-VRF ID and a constant value of 1 that is calculated using the previously mentioned random seed values.

The algorithm shown in auto_evpn_fidevividsid2mac (Figure 22) is required to derive MAC addresses.

6.9.3.4. IPv6 IRB Gateway Address

The derived IPv6 gateway address MUST be from a ULA-scoped range, which accounts for the first 6-bytes. The next 5-bytes MUST be the last bytes of the derived MAC address. Finally, the remaining 7-bytes MUST be ::0001.

The algorithm shown in auto_evpn_fidevividsid2v6subnet (Figure 21) is required to derive the IPv6 gateway address.

6.9.3.5. IPv4 IRB Gateway Address

The derived IPv4 gateway address MUST be from a RFC1918 range, which accounts for the first octet. The next octet MUST a hash of the MAC-VRF ID and a constant value of 17 that is calculated using the previously mentioned random seed values. Finally, the remaining 2 octets MUST be 0 and 1 respectively.

The algorithm shown in auto_evpn_v4prefixfidevividsid2v4subnet (Figure 19) is required to derive the IPv4 gateway address. It should be noted that there is a higher likelihood of address collisions when deriving IPv4 addresses.

6.9.4. Traffic Routed to External Destinations

6.9.4.1. Route Distinguisher

Nodes hosting IP Prefix routes MUST derive a type-0 route distinguisher based on its System ID and Fabric ID so that it is unique per IP-VRF and per node.

The algorithm shown in auto_evpn_sidfid2rd (Figure 18) is required to derive the Route Target.

6.9.4.2. Route Target

Nodes hosting IP prefix routes MUST derive a route target extended community based on the MAC-VRF ID for each IP-VRF so that it is unique across the network. Route targets MUST be of type 0.

The algorithm shown in auto_evpn_evi2rt (Figure 12) is required to derive the Route Target community.

6.10. Auto-EVPN Analytics

Leaf nodes MAY optionally advertise analytics information about the Auto-EVPN fabric to ToF nodes using RIFT Key-Value TIEs. This may be advantageous in that overlay validation and troubleshooting activities can be performed on the ToF nodes.

This section requests suggested values from the RIFT Well-Known Key-Type Registry and describes their use for Auto-EVPN.

Name	Value	Description
Auto-EVPN Analytics MAC-VRF	3	Analytics describing a MAC-VRF on a particular node within a fabric.
Auto-EVPN Analytics Global	4	Analytics describing an Auto-EVPN node within a fabric.

Table 1: Requested RIFT Key Registry Values

The normative Thrift schema can be found in the appendix (Appendix A.4).

6.10.1. Auto-EVPN Global Analytics Key Type

This Key Type describes node level information within the context of the Auto-EVPN fabric. The System ID of the advertising leaf node MUST be used to differentiate the node among other nodes in the fabric.

The Auto-EVPN Global Key Type MUST be advertised with the RIFT Fabric ID encoded into the 3rd and 4th bytes of the Key Identifier.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Well-Known  |             Auto-EVPN (Global)                 |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     (Auto-EVPN Role,                                          |
|      Established BGP Peer Count,                              |
|      Total BGP Peer Count,)                                   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 1: Auto-EVPN Global Key-Value TIE

where:

Auto-EVPN Role:
The value indicating the node's Auto-EVPN role within the fabric.

0: Illegal value, MUST NOT be used.

1: Auto-EVPN Leaf Gateway

2: Auto-EVPN Top-of-Fabric Gateway

Established BGP Session Count:
A 16-bit integer indicating the number of BGP sessions in the Established state.

Total BGP Peer Count:
A 16-bit integer indicating the total number of possible BGP sessions on the local node, regardless of state.

6.10.2. Auto-EVPN MAC-VRF Key Type

This Key-Value structure contains information about a specific MAC-VRF within the Auto-EVPN fabric.

The Auto-EVPN MAC-VRF Key Type MUST be advertised with the Auto-EVPN MAC-VRF ID encoded into the 3rd and 4th bytes of the Key Identifier.

All values advertised in a MAC-VRF Key-Value TIE MUST represent only state of the local node.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Well-Known  |             Auto-EVPN (MAC-VRF)                |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     (Operational CE Interface Count,                          |
|      Total CE Interface Count,                                |
|      Operational IRB Interface Count,                         |
|      Total IRB Interface Count,                                |
|      EVPN Type-2 MAC Route Count,                             |
|      EVPN Type-2 MAC/IP Route Count,                          |
|      Configured VLAN Count,                                   |
|      MAC-VRF Name,                                            |
|      MAC-VRF Description,)                                    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 2: Auto-EVPN MAC-VRF Key-Value TIE

where:

Operational Customer Edge Interface Count:
A 16-bit integer indicating the number of CE interfaces associated with the MAC-VRF where both administrative and operational status are "up".

Total Customer Edge Interface Count:
A 16-bit integer indicating the total number of CE interfaces associated with the MAC-VRF regardless of interface status.

Operational IRB Interface Count:
A 16-bit integer indicating the number of IRB interfaces associated with the MAC-VRF where both administrative and operational status are "up".

Total IRB Interface Count:
A 16-bit integer indicating the total number of IRB interfaces associated with the MAC-VRF regardless of interface status.

EVPN Type-2 MAC Route Count:
A 32-bit integer indicating the total number of EVPN Type-2 MAC routes.

EVPN Type-2 MAC/IP Route Count:
A 32-bit integer indicating the total number of EVPN Type-2 MAC/IP routes.

VLAN Count:
A 16-bit integer indicating the total number configured VLANs.

MAC-VRF Name:
A string used to indicate the name of the MAC-VRF on the node.

MAC-VRF Description:
A string used to describe the MAC-VRF on the node, similar to that of an interface description.

7. Acknowledgements

The authors would like to thank Olivier Vandezande, Matthew Jones, and Michal Styszynski for their contributions.

8. Security Considerations

This document introduces no new security concerns to RIFT or other specifications referenced in this document.

9. References

9.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC7432] Sajassi, A., Aggarwal, R., Bitar, N., Isaac, A., Uttaro, J., Drake, J., and W. Henderickx, "BGP MPLS-Based Ethernet VPN", February 2015, <https://www.rfc-editor.org/info/rfc7432>.

[RIFT] Przygienda, T., Sharma, A., Thubert, P., Rijsman, B., and D. Afanasiev, "RIFT: Routing in Fat Trees", Work in Progress, draft-ietf-rift-rift-13, July 2021.

[RIFT-KV] Head, J. and T. Przygienda, "RIFT Keys Structure and Well-Known Registry in Key Value TIE", Work in Progress, draft-head-rift-kv-registry-01, July 2021.

Appendix A. Thrift Models

This section contains the normative Thrift models required to support Auto-EVPN. Per the main RIFT [RIFT] specification, all signed values MUST be interpreted as unsigned values.

A.1. RIFT LIE Schema

A.1.1. Auto-EVPN Version

```
struct LIEPacket {
...
   /** It provides optional version of EVPN ZTP as 256 * MAJOR + MINOR */
   26: optional i16                       auto_evpn_version;
...
```

A.1.2. Fabric ID

```
struct LIEPacket {
...
   /** It provides the optional ID of the configured fabric  */
   25: optional common.FabricIDType       fabric_id;
...
```

A.2. RIFT Node-TIE Schema

A.2.1. Auto-EVPN Version

```
struct NodeTIEElement {
...
   /** It provides optional version of EVPN ZTP as 256 * MAJOR + MINOR */
   13: optional i16                         auto_evpn_version;
...
```

A.2.2. Fabric ID

```
struct NodeTIEElement {
...
   /** It provides the optional ID of the Fabric configured */
   12: optional common.FabricIDType         fabric_id;
...
```

A.3. common_evpn.thrift

This section contains the normative Auto-EVPN Thrift schema.

```
/**
    Thrift file for common AUTO EVPN definitions for RIFT

    Copyright (c) Juniper Networks, Inc., 2016-
    All rights reserved.
*/

namespace py common_evpn
namespace rs models

include "common.thrift"
include "encoding.thrift"
include "statistics.thrift"

const common.FabricIDType   default_fabric_id     = 1
const i8                    default_evis          = 3
const i8                    default_vlans_per_evi = 7

typedef i32       RouterIDType
typedef i32       ASType
typedef i32       ClusterIDType

struct EVPNAnyRole {
    1: required   common.IPv6Address                    v6_loopback,
    2: required   common.IPv6Address                    type5_v6_loopbac
    3: required   common.IPv4Address                    type5_v4_loopbac
    4: required   RouterIDType                          bgp_router_id,
    5: required   ASType                                autonomous_syste
    6: required   ClusterIDType                         cluster_id,
    /** prefixes to be redistributed north */
    7: required   set<common.IPPrefixType>              redistribute_nor
    /** prefixes to be redistributed south */
    8: required   set<common.IPPrefixType>              redistribute_sou
    /** group name for evpn auto overlay */
    9: required   string                                bgp_group_name,
    /** fabric prefixes to be advertised in rift instead of default */
   10: required   set<common.IPPrefixType>              fabric_prefixes,
}

struct PartialEVPNEVI {
    // route target per RFC4360
    1: required   CommunityType                         rt_target,
    2: required   RTDistinguisherType                   rt_distinguisher
    3: required   RTDistinguisherType                   rt_type5_disting
    5: required   string                                mac_vrf_name,
    6: required   VNIType                               type5_vni,
}

struct EVPNRRRole {
    2: required   common.IPv6Address                    v6_rr_addr_loopb
    3: required   common.IPv6PrefixType                 v6_peers_allowed
    4: required   map<MACVRFNumberType, PartialEVPNEVI> evis,
}

typedef i64         RTDistinguisherType
typedef i64         RTTargetType
typedef i16         MACVRFNumberType

typedef i16         VLANIDType
typedef binary      MACType

typedef i16         UnitType

struct IRBType {
    1: required   string                                name,
    2: required   UnitType                              unit,
    /// constant
    3: required   MACType                               mac,
    /// contains address of the gateway as well
    4: optional   common.IPv6PrefixType                 v6_subnet,
    /// contains address of the gateway as well
    5: optional   common.IPv4PrefixType                 v4_prefix,
}

typedef i32      VNIType

struct VLANType {
    1: optional   VLANIDType                            id,
    2: required   string                                name,
    3: optional   IRBType                               irb,
    5: optional   bool                                  stretched = fals
    6: optional   bool                                  is_native = fals
}

struct CEInterfaceType {
    2: optional   common.IEEE802_1ASTimeStampType       moved_to_ce,
    // we may not be able to obtain it in case of internal errors
    3: optional   string                                platform_interfa
}

typedef i64       CommunityType

struct EVPNEVI {
    // route target per RFC4360
    1: required   CommunityType                         rt_target,
    2: required   RTDistinguisherType                   rt_distinguisher
    3: required   RTDistinguisherType                   rt_type5_disting
    4: required   string                                mac_vrf_name,
    // fabric unique 24 bits VNI on non-stretch, otherwise unique across
    5: required   map<VNIType, VLANType>                vlans,
    6: required   VNIType                               type5_vni,
}

struct EVPNLeafRole {
    1: required   set<common.IPv6Address>               rrs,
    2: required   map<MACVRFNumberType, EVPNEVI>        evis,
    3: optional   map<common.LinkIDType,
                      CEInterfaceType>                  ce_interfaces,

    5: optional   binary                                leaf_unique_lacp
    6: optional   binary                                fabric_unique_la
}

/// structure to indicate EVPN roles assumed and their variables for
/// external platform to configure itself accordingly. Presence of
/// according structure indicates that the role is assumed.
struct EVPNRoles {
    1: required  EVPNAnyRole                            generic,
    2: optional  EVPNRRRole                             route_reflector,
    3: optional  EVPNLeafRole                           leaf,
}

const common.TimeIntervalInSecType          default_leaf_delay = 120
const common.TimeIntervalInSecType          default_interface_ce_delay =
/// default delay before EVPNZTP FSM starts to compute anything
const common.TimeIntervalInSecType          default_evpnztp_startup_dela
```

A.4. auto_evpn_kv.thrift

This section contains the normative Auto-EVPN Analytics Thrift schema.

```
include "common.thrift"

namespace py auto_evpn_kv
namespace rs models

/** We don't need the full role structure, only an indication of the nod
enum AutoEVPNRole {
    ILLEGAL            = 0,
    auto_evpn_leaf_erb = 1,
    auto_evpn_tof_gw   = 2,
}

enum   KVTypes {
    OUI       = 1,
    WellKnown = 2,
}

const i8            AutoEVPNWellKnownKeyType  = 1
typedef i32         AutoEVPNKeyIdentifier
typedef i16         AutoEVPNCounterType
typedef i32         AutoEVPNLongCounterType

const i8            GlobalAutoEVPNTelemetryKV = 4
const i8            AutoEVPNTelemetryKV       = 3

/** Per the according RIFT draft the key comes from the well known space
    Part of the key is used as Fabric-ID.

    1st     byte  MUST be = "Well-Known"
    2nd     byte  MUST be = "Global Auto-EVPN Telemetry KV",
    3rd/4th bytes MUST be = FabricIDType
*/
struct AutoEVPNTelemetryGlobalKV {
    /** Only values that the ToF cannot derive itself should be flooded.
    1: required   set<AutoEVPNRole>            auto_evpn_roles,

    /** Established BGP peer count (for Auto-EVPN)
    2: optional   AutoEVPNCounterType          established_bgp_peer_coun

    /** Total BGP peer count (for Auto-EVPN)
    3: optional   AutoEVPNCounterType          total_bgp_peer_count,
}

/** Per the according RIFT draft the key comes from the well known space
    Part of the key is used as MAC-VRF number.

    1st     byte  MUST be = "Well-Known"
    2nd     byte  MUST be = indicates "Auto-EVPN Telemetry KV",
    3rd/4th bytes MUST be = MACVRFNumberType
*/
```

```
struct AutoEVPNTelemetryMACVRFKV {
    /** Active CE interface count (up/up)
    1: optional   AutoEVPNCounterType          active_ce_interfaces,

    /** Total CE interface count
    2: optional   AutoEVPNCounterType          total_ce_interfaces,

    /** Active IRB interface count (up/up)
    3: optional   AutoEVPNCounterType          active_irb_interfaces,

    /** Total IRB interface count
    4: optional   AutoEVPNCounterType          total_irb_interfaces,

    /** Local EVPN Type-2 MAC route count
    5: optional  AutoEVPNLongCounterType       local_evpn_type2_mac_rout

    /** Local EVPN Type-2 MAC/IP route count
    6: optional  AutoEVPNLongCounterType       local_evpn_type2_mac_ip_r

    /** number of configured VLANs */
    7: optional  i16                           configured_vlans,

    /** optional human readable name */
    8: optional  string                        name,

    /** optional human readable string describing the MAC-VRF */
    9: optional  string                        description,
}
```

Figure 3: Auto-EVPN Key-Value Thrift Schema

Appendix B. Auto-EVPN Variable Derivation

```
use std::cell::{RefCell, RefMut};
use std::cmp::{max, min};
use std::collections::{BTreeMap, BTreeSet, HashMap};
use std::fmt::Debug;
use std::net::{Ipv4Addr, Ipv6Addr};
use std::str::FromStr;
use itertools::interleave;
use itertools::Itertools;
use rayon::slice::ParallelSliceMut;

use foundation::models::common::{FabricIDType, IPv6PrefixType};
use foundation::models::common::LevelType;
use foundation::models::common::HierarchyIndications;
use foundation::models::common::IPPrefixType;
use foundation::models::common::IPv4Address;
use foundation::models::common::IPv4PrefixType;
use foundation::models::common::IPv6Address;
use foundation::models::common::LEAF_LEVEL;
use foundation::models::common_services::ServiceErrorType;
use foundation::models::common_evpn::{DEFAULT_EVIS, DEFAULT_VLANS_PER_EV
                                          DEFAULT_EVPNZTP_STARTUP_DELAY,
                                          DEFAULT_LEAF_DELAY, EVPNAnyRol
                                          EVPNRoles, EVPNRRRole, UnitTyp
use foundation::models::common_evpn::CEInterfaceType;
use foundation::models::common_evpn::CommunityType;
use foundation::models::common_evpn::EVPNEVI;
use foundation::models::common_evpn::IRBType;
use foundation::models::common_evpn::MACVRFNumberType;
use foundation::models::common_evpn::PartialEVPNEVI;
use foundation::models::common_evpn::RTDistinguisherType;
use foundation::models::common_evpn::VLANIDType;
use foundation::models::common_evpn::VLANType;
use foundation::models::common_evpn::VNIType;
use ILLEGAL_SYSTEM_I_D;
use NodeCapabilities;
use UnsignedSystemID;
```

Figure 4: auto_evpn_imports

```
/// indicates how many RRs we're computing in EVPN ZTP
pub const MAX_AUTO_EVPN_RRS: usize = 3;
/// indicates the fabric has no ID, used in computations to omit effects
pub const NO_FABRIC_ID: FabricIDType = 0;
/// invalid MACVRF number, MACVRFs start from 1
pub const NO_MACVRF: MACVRFNumberType = 0;

/// unique v6 prefix for all nodes starts with this
pub const AUTO_EVPN_V6PREF: &str = "FD00:A1";
/// how many bytes in a v6pref for RRs
pub const AUTO_EVPN_V6PREFLEN: usize = 8 * 3;
/// unique v6 prefix for route reflector purposes starts like this
pub const AUTO_EVPN_V6RRPREF: &str = "FD00:A2";
/// unique v6 prefix for type-5 purposes starts like this
pub const AUTO_EVPN_V6T5PREF: &str = "FD00:A3";
/// unique v6 prefix for IRB prefix purposes
pub const AUTO_EVPN_V6IRBPREF: &str = "FD00";
/// unique v6 prefix first byte for v6 IRB prefix purposes
pub const AUTO_EVPN_V6IRBPREFFIRSTBYTE: u8 = 0xA4;
/// unique v4 prefix for IRB purposes
pub const AUTO_EVPN_V4IRBPREF: &str = "10";
/// 3 bytes of prefix type and then we have fabric ID after that
pub const AUTO_EVPN_V6_FABPREFIXLEN: usize = 8 + 8 + 8 + 16;

/// per RFC magic
const RT_TARGET_HIGH: CommunityType = 0;
const RT_TARGET_LOW: CommunityType = 0;

/// first available VLAN number
pub const FIRST_VLAN: VLANIDType = 1;
// maximum vlan number one less than maximum to use as bitmask
pub const MAX_VLAN: VLANIDType = 4095;
/// constant VLAN shift
pub const FIRST_VLAN_SHIFT: VLANIDType = 16;
/// NATIVE VLAN number
pub const NATIVE_VLAN: VLANIDType = 1;

/// abstract description of VLAN properties for a derived VLAN
pub struct VLANDescription {
    pub vlan_id: VLANIDType,
    pub name: String,
    /// can this VLAN be stretched across multiple fabrics
    pub stretchable: bool,
    pub native: bool,
}

/// maximum number of VLANs per MACVRF
pub const MAX_VLANS_PER_EVI: usize = 15;
pub type VLANStretchableType = bool;
pub type VLANNativeType = bool;

pub const EXTRATYPE5_RD_DISTINGUISHER: u32 = 0xffff_ffff;

/// high bits of type 5 VNI
const TYPE5VNIHIGH: VNIType = 0x0080_0000;
/// bitmask for type 2 VNI
const TYPE2VNIMASK: VNIType = 0x00ff_ffff ^ TYPE5VNIHIGH;

/// random seeds used in several algorithms to increase entropy
pub const RANDOMSEEDS: [u64; 4] = [
    27008318799u64,
    67438371571,
    37087353685,
    88675895388,
];
```

Figure 5: auto_evpn_const_structs_type

```
pub(crate) fn auto_evpn_sids2rrs(mut v: Vec<UnsignedSystemID>) -> Vec<Un
    v.par_sort_unstable();
    let r = if v.len() > 2 {
        let mut s = v.split_off(v.len() / 2);
        s.reverse();
        interleave(v.into_iter(), s.into_iter()).collect()
    } else {
        v
    };

    r
}
```

Figure 6: auto_evpn_sids2rrs

```
pub(crate) fn auto_evpn_v62octets(a: Ipv6Addr) -> Vec<u8> {
    a.octets().iter().cloned().collect()
}
```

Figure 7: auto_evpn_v62octets

```
/// fabric prefixes derived instead of advertising default on the fabric
/// for default route on ToF or leaves
pub fn auto_evpn_fid2fabric_prefixes(fid: FabricIDType) -> Result<Vec<IP
    vec![
        (auto_evpn_fidsidv6loopback(fid, ILLEGAL_SYSTEM_I_D as _), AUTO_
        (auto_evpn_fidrrpref2rrloopback(fid, ILLEGAL_SYSTEM_I_D as _), A
    ]
        .into_iter()
        .map(|(p, _)|
            match p {
                Ok(_) => Ok(
                    IPPrefixType::Ipv6prefix(
                        IPv6PrefixType {
                            address: auto_evpn_v62octets(p?),
                            prefixlen: AUTO_EVPN_V6PREFLEN as _,
                        })),
                Err(e) => Err(e),
            }
        )
        .collect::<Result<Vec<_>, _>>()
}
```

Figure 8: auto_evpn_fid2fabric_prefixes

```
/// local address with encoded fabric ID and system ID for collision fre
/// for several different prefixes.
pub fn auto_evpn_v6prefixfidsid2loopback(v6pref: &str, fid: FabricIDType
                                         sid: UnsignedSystemID) -> Resul
    assert!(fid != 0);
    let a = format!("{}{:02X}::{}",
                    v6pref,
                    fid as u16,
                    sid.to_ne_bytes()
                        .iter()
                        .chunks(2)
                        .into_iter()
                        .map(|chunk|
                            chunk.fold(0u16, |v, n| (v << 8) | *n as u16
                        .map(|v| format!("{:04X}", v))
                        .collect::<Vec<_>>()
                        .into_iter()
                        .join(":")
    );

    Ipv6Addr::from_str(&a)
        .map_err(|_| ServiceErrorType::INTERNALRIFTERROR)
}
```

Figure 9: auto_evpn_v6prefixfidsid2loopback

```
/// auto evpn V6 loopback for RRs
pub fn auto_evpn_fidrrpref2rrloopback(fid: FabricIDType,
                                      preference: u8) -> Result<Ipv6Addr
    auto_evpn_v6prefixfidsid2loopback(AUTO_EVPN_V6RRPREF, fid, (1 + pref
}
```

Figure 10: auto_evpn_fidrrpref2rrloopback

```
/// auto evpn BGP router ID
pub fn auto_evpn_sidfid2bgpid(fid: FabricIDType, sid: UnsignedSystemID)
    assert!(fid != 0);
    let hs: u32 = ((sid & 0xffff_ffff_0000_0000) >> 32) as _;
    let mut ls: u32 = (sid & 0xffff_ffff) as _;
    ls = ls.rotate_right(7) ^ (fid as u32).rotate_right(13);
    max(1, hs ^ ls) // never a 0
}
```

Figure 11: auto_evpn_sidfid2bgpid

```
/// route target bytes are type0/0 and then add EVI
pub fn auto_evpn_evi2rt(evi: MACVRFNumberType) -> CommunityType {
    let wideevi = (evi + 1) as CommunityType;

    (RT_TARGET_HIGH << (64 - 8)) | (RT_TARGET_LOW << 64 - 16) |
        ((wideevi) << 17) |
        ((wideevi))
}
```

Figure 12: auto_evpn_evi2rt

```
/// type-5 VNI for an EVI
pub fn auto_evpn_fidevi2type5vni(fid: FabricIDType, evi: MACVRFNumberTyp
    TYPE5VNIHIGH | auto_evpn_fidevivid2vni(fid, evi, 0)
}
```

Figure 13: auto-evpn_fidevi2type5vni

```
/// type-2 VNI for a specific VLAN
pub fn auto_evpn_fidevivid2vni(fid: FabricIDType, evi: MACVRFNumberType,
    let rfid = fid as i32;
    let revi = evi as i32;
    let rvlan = vlanid as i32;
// mask out high bits, VNI is only 24 bits
    TYPE2VNIMASK &
        (
            rfid.rotate_left(16) ^
                revi.rotate_left(12) ^
                rvlan
        )
}
```

Figure 14: auto_evpn_fidevivid2vni

```
/// maximum VLANs per EVI supported by auto evpn when deriving
pub fn auto_evpn_vlan_description_table<'a>(vlans: usize)
                                            -> Result<&'a [(VLANIDType,
    // up to 15 vlans can be activated
    const VLANSARRAY: [(i16, bool, bool); MAX_VLANS_PER_EVI] = [
        (NATIVE_VLAN, true, true, ),
        (FIRST_VLAN_SHIFT, true, false, ),
        (FIRST_VLAN_SHIFT + 1, true, false, ),
        (FIRST_VLAN_SHIFT + 2, true, false, ),
        (FIRST_VLAN_SHIFT + 3, false, false, ),
        (FIRST_VLAN_SHIFT + 4, false, false, ),
        (FIRST_VLAN_SHIFT + 5, false, false, ),
        (FIRST_VLAN_SHIFT + 6, false, false, ),
        (FIRST_VLAN_SHIFT + 7, false, false, ),
        (FIRST_VLAN_SHIFT + 8, false, false, ),
        (FIRST_VLAN_SHIFT + 9, false, false, ),
        (FIRST_VLAN_SHIFT +10, false, false, ),
        (FIRST_VLAN_SHIFT +11, false, false, ),
        (FIRST_VLAN_SHIFT +12, false, false, ),
        (FIRST_VLAN_SHIFT +13, false, false, ),
    ];

    if vlans > VLANSARRAY.len() {
        return Err(ServiceErrorType::INVALIDPARAMETERVALUE)
    }

    Ok(&VLANSARRAY[..vlans])
}
```

Figure 15: auto_evpn_vlan_description_table

```
/// delivers the vlan description that can be used to generate vlans for
/// specific fabric ID and a MACVRF number
pub fn auto_evpn_fidevivlansvlans2desc(fid: FabricIDType, macvrf: MACVRF
                                       vlans: usize) -> Vec<VLANDescript

    assert!(NO_MACVRF != macvrf);

    // abstract description of derived VLANs
    let vlan_table = auto_evpn_vlan_description_table(vlans)
        .expect("vlan table in AUTO EVPN incorrect");

    let vlanshift = vlan_table
        .iter()
        .map(|(vl, _, _)| *vl as usize)
        .max()
        .expect("vlan table in AUTO EVPN incorrect")
        .checked_next_power_of_two()
        .expect("vlan table in AUTO EVPN incorrect");

    assert!(vlan_table.len() < FIRST_VLAN_SHIFT as _);

    vlan_table
        .iter()
        .map(move |(vid, stretch, native_)| {
            let stretchedfid = if !stretch {
                fid
            } else {
                NO_FABRIC_ID
            };

            let mut vlan_id = *vid ^ stretchedfid
                .rotate_left(max(16, vlanshift as u32 + 8)) as VLANIDTyp
            // leave space for VLANs in the encoding
            vlan_id ^= macvrf.rotate_left(vlanshift as _) as VLANIDType;

            vlan_id %= MAX_VLAN;
            vlan_id = max(1, vlan_id);

            VLANDescription {
                vlan_id: vlan_id as _,
                name: format!("V{}", vlan_id),
                stretchable: *stretch,
                native: *native_,
            }
        })
        .collect()
}
```

Figure 16: auto_evpn_fidevivlansvlans2desc

```
/// IRB interface number.
/// fid/evi combination shifted up to not interfere with the VLAN-ID
/// and then add the VLAN-ID
pub fn auto_evpn_fidevivid2irb(fid: FabricIDType, evi: MACVRFNumberType,

    assert!(NO_MACVRF != evi);

    let mut v = (fid as UnitType ^ evi.rotate_left(4) as UnitType) << (1

    v = 1 + v.wrapping_add(vid) % MAX_VLAN;
    v % (UnitType::MAX - 1)
}
```

Figure 17: auto_evpn_fidevivid2irb

```
/// route distinguisher derivation
pub fn auto_evpn_sidfid2rd(sid: UnsignedSystemID, fid: FabricIDType, ext
    // generate type 0 route distinguisher, first 2 bytes 0 and then 6 b
    assert!(fid != NO_FABRIC_ID);
    // shift the 2 bytes we loose
    let convsid = sid as RTDistinguisherType;
    let hs = ((sid & 0xffff_0000_0000_0000) >> 32) as RTDistinguisherTyp
    let mut ls: RTDistinguisherType = convsid & 0x0000_ffff_ffff_ffff;
    ls ^= hs;
    ls ^= (fid as RTDistinguisherType).rotate_left(16);
    ls ^= extra as RTDistinguisherType;
    ls
}
```

Figure 18: auto_evpn_sidfid2rd

```
/// v4 subnet derivation
pub fn auto_evpn_v4prefixfidevividsid2v4subnet(v4pref: &str, fid: Fabric
                                               evi: MACVRFNumberType, vi
                                               sid: UnsignedSystemID) ->

    assert!(NO_MACVRF != evi);

    // fid can be 0 for stretched v4subnets
    let mut sub = evi.to_ne_bytes().iter()
        .fold((RANDOMSEEDS[0] & 0xff) as u8, |r, e| r.rotate_left(1) ^ e
    sub ^= fid.to_ne_bytes().iter()
        .fold((RANDOMSEEDS[1] & 0xff) as u8, |r, e| r.rotate_left(2) ^ e
    sub ^= vid.to_ne_bytes().iter()
        .fold((RANDOMSEEDS[2] & 0xff) as u8, |r, e| r.rotate_left(3) ^ e

    let subnet = sub % 254; // make sure we don't show multicast subnet

    let _host = sid.to_ne_bytes().iter()
        .fold(0u16, |r, e| r.rotate_left(3) ^ e.rotate_right(3) as u16);

    let a = format!("{}.{}.{}.{}",
                    v4pref,
                    subnet,
                    0,
                    1,
    );

    Ok(
        IPv4PrefixType {
            address: Ipv4Addr::from_str(&a)
                .map_err(|_| {
                    ServiceErrorType::INTERNALRIFTERROR
                })?
                .octets()
                .iter()
                .fold(0u32, |v, nv| v << 8 | (*nv as u32)) as IPv4Addres
            ,
            prefixlen: 16,
        }
    )
}
```

Figure 19: auto_evpn_v4prefixfidevividsid2v4subnet

```
/// generic v6 bytes derivation used for different purposes
pub fn auto_evpn_v6hash(fid: FabricIDType, evi: MACVRFNumberType, vid: V
                        -> [u8; 8] {

    let mut sub = evi.to_ne_bytes().iter()
        .fold(RANDOMSEEDS[3], |r, e| r.rotate_left(6) ^ e.rotate_right(4
    sub ^= fid.to_ne_bytes().iter()
        .fold(RANDOMSEEDS[0], |r, e| r.rotate_left(6) ^ e.rotate_right(4
    sub ^= vid as u64;
    sub ^= sid;

    sub.to_ne_bytes()
}
```

Figure 20: auto_evpn_v6hash

```
pub fn auto_evpn_fidevividsid2v6subnet(fid: FabricIDType, evi: MACVRFNum
                                       vid: VLANIDType,
                                       sid: UnsignedSystemID) -> Result<

    assert!(NO_MACVRF != evi);

    let sb = auto_evpn_v6hash(fid, evi, vid, sid);

    let a = format!("{}:{:02X}{:02X}:{:02X}{:02X}:{:02X}{:02X}::1",
                    AUTO_EVPN_V6IRBPREF,
                    AUTO_EVPN_V6IRBPREFFIRSTBYTE,
                    sb[3] ^ sb[0],
                    sb[4] ^ sb[1],
                    sb[5] ^ sb[2],
                    sb[6],
                    sb[7],
    );

    Ok(IPv6PrefixType {
        address: Ipv6Addr::from_str(
            &a)
            .map_err(|_| {
                ServiceErrorType::INTERNALRIFTERROR
            })?
            .octets()
            .to_vec(),
        prefixlen: 64,
    })
}
```

Figure 21: auto_evpn_fidevividsid2v6subnet

```
/// MAC address derivation for IRB
pub fn auto_evpn_fidevividsid2mac(fid: FabricIDType, evi: MACVRFNumberTy
                                  vid: VLANIDType, sid: UnsignedSystemID

    let sb = auto_evpn_v6hash(fid, evi, vid, sid);

    vec![0x02,
         sb[3] ^ sb[0],
         sb[4] ^ sb[1],
         sb[5] ^ sb[2],
         sb[6],
         sb[7],
    ]
}
```

Figure 22: auto_evpn_fidevividsid2mac

```
/// v4 loopback address derivation for every node in auto-evpn
pub fn auto_evpn_fidsid2v4loopback(fid: FabricIDType, sid: UnsignedSyste
    let mut derived = sid.to_ne_bytes().iter()
        .fold(0 as IPv4Address, |p, e| (p << 4) ^ (*e as IPv4Address));
    derived ^= fid as IPv4Address;
    // use the byte we loose for entropy
    derived ^= derived >> 24;
    // and sanitize for loopback range
    derived &= 0x00ff_ffff;

    let m = ((127 as IPv4Address) << 24) | derived;
    m as _
}
```

Figure 23: auto_evpn_fidsid2v4loopback

```
/// V6 loopback derivation for every node in auto-evpn
pub fn auto_evpn_fidsidv6loopback(fid: FabricIDType,
                                  sid: UnsignedSystemID) -> Result<Ipv6A
    auto_evpn_v6prefixfidsid2loopback(AUTO_EVPN_V6PREF, fid, sid)
}
```

Figure 24: auto_evpn_fidsidv6loopback

```
#[allow(non_snake_case)]
pub fn auto_evpn_fid2private_AS(fid: FabricIDType) -> u32 {
    assert!(fid != NO_FABRIC_ID);
    // range 4200000000-4294967294
    const DIFF: u32 = 4_294_967_294 - 4_200_000_000;
    64496 + ((fid as u32) << 3) % DIFF
}
```

Figure 25: auto_evpn_fid2private_AS

```
pub fn auto_evpn_fid2clusterid(fid: FabricIDType) -> u32 {
    auto_evpn_fid2private_AS(fid)
}
```

Figure 26: auto_evpn_fid2clusterid

Authors' Addresses

Jordan Head (editor)
Juniper Networks
1137 Innovation Way
Sunnyvale, CA
United States of America

Email: jhead@juniper.net

Tony Przygienda
Juniper Networks
1137 Innovation Way
Sunnyvale, CA
United States of America

Email: prz@juniper.net

Wen Lin
Juniper Networks
10 Technology Park Drive
Westford, MA
United States of America

Email: wlin@juniper.net