RIFT J. Head, Ed.
Internet-Draft T. Przygienda
Intended status: Standards Track W. Lin
Expires: 26 August 2021 Juniper Networks
22 February 2021

RIFT Auto-EVPN
draft-head-rift-auto-evpn-00

Abstract

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

Status of This Memo

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). 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 KV TIEs can be used to distribute state information to allow for validation of basic operational correctness without need for further tooling.

3. System ID

The 64-bit RIFT System ID that uniquely identifies a node as defined in [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 but when they do they do assume EVPN roles and MUST derive according 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 Adresses
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 Appendix section to allow interoperability between different implementations.

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.

IPv4 addresses MAY be supported, but it should be noted that they have a higher likelihood of collision.

The required algorithm can be found in the appendix (Appendix A.3.3).

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.

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.

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.

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 required algorithm for 2-byte ASNs can be found in the appendix (Appendix A.3.4).

6.5. 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. Implementations MAY choose to simply use the AS number as the cluster ID.

The required algorithm can be found in the appendix (Appendix A.3.5).

6.6. 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 required algorithm can be found in the appendix (Appendix A.3.6).

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 required algorithm can be found in the appendix (Appendix A.3.7).

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 required algorithm can be found in the appendix (Appendix A.3.8).

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 "random seed" values to provide additional entropy.

The required derivation procedures can be found in the appendix (Appendix A.3).

6.9.1. Untagged Traffic in Multiple Fabrics

This section defines a 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.

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.

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.

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.

6.9.2. Tagged Traffic in Multiple Fabrics

This section defines a 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.

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.

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.

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.

6.9.3. Tagged Traffic in a Single Fabric

This section defines a methods 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.

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.

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.

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.

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 required algorithm can be found in the appendix (Appendix A.3.8).

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 required algorithm can be found in the appendix (Appendix A.3.7).

7. Acknowledgements

TBD

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-12, May 2020.

Appendix A. Appendix

A.1. RIFT LIE Schema

A.1.1. Auto-EVPN Version

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

A.1.2. Fabric ID

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

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. Variable Derivation

A.3.1. Random Seed Values

To be provided in future version of this document.

A.3.2. Fabric ID

To be provided in future version of this document.

A.3.3. Loopback Address

To be provided in future version of this document.

A.3.4. Autonomous System Number

To be provided in future version of this document.

A.3.5. Cluster ID

To be provided in future version of this document.

A.3.6. Router ID

To be provided in future version of this document.

A.3.7. Route Target

To be provided in future version of this document.

A.3.8. Route Distinguisher

To be provided in future version of this document.

A.3.9. VLAN

To be provided in future version of this document.

A.3.10. VNI

To be provided in future version of this document.

A.3.11. Gateway (MAC)

To be provided in future version of this document.

A.3.12. Gateway (IPv6)

To be provided in future version of this document.

A.3.13. Gateway (IPv4)

To be provided in future version of this document.

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

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: 26 August 2021

J. Head
A. Przygienda
Juniper Networks

22 February 2021

RIFT Keys Structure and Well-Known Registry in Key Value TIE
draft-head-rift-kv-registry-00

Abstract

Routing in Fat-Trees RIFT [RIFT] allows for key/value pairs to be advertised within Key-Value Topology Information Elements (KV TIEs). The data contained within these KV TIEs can be used for any imaginable purpose. This document defines the various Key Types (i.e. Well-Known, OUI, and Experimental) and a method to structure corresponding values.

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].

Status of This Memo

Copyright Notice

Table of Contents

1. Description

Routing in Fat-Trees (RIFT [RIFT]) allows for key/value pairs to be advertised within Key-Value Topology Information Elements (KV TIEs). There are no restrictions placed on the type of data that is contained in KV TIEs nor what the data is used for. It could contain a simple string or even Thrift encoded data. However, the KV elements SHOULD NOT be used to carry topology information used by RIFT itself to perform distributed computations.

This document defines a Key Type Registry to maintain Well-Known and vendor specific Key Types in order to simplify interoperability between implementations and eliminate the risk of collision for future implementations. An Experimental Key Type is additionally defined.

2. Key-Value Pair Structure

Figure 1 illustrates the generic Key-Value Pair structure.

```
 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
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Key-Type    |                 Key Identifier                |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       Values (variable)                       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 1: Generic Key-Value Structure

where:

Key-Type:
A 1-byte value that identifies the Key Type. It MUST be a reserved value from the Key Value Type Registry that is defined later in this document.

Key Identifier:
A 3-byte value that identifies the specific Key and describes the structure of the contained values.

Values:
A variable length value that contains data associated with the Key. It SHOULD contain 1 or more elements. Whether the collection of elements allows duplicates and/or is ordered is governed by the particular key identifier.

2.1. Well-Known Key Type

This section reserves a value in the Key Type Registry to indicate Well-Known Key Types that all implementations SHOULD support.

As shown in Figure 2, the Key-Type will be used to identify that the Key Type is Well-Known. The Key Identifier will be used to identify the specific Key and describe the structure of the contained values.

```
 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
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      TBD2     |         Well-Known Key Identifier             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                 Well-Known Values (variable)                  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 2: Well-Known Key Type

2.2. OUI Key Type

This section reserves a value in the Key Type Registry to indicate an OUI (vendor-specific) Key Type that any implementation MAY support.

As shown in Figure 3, the Key-Type will be used to identify the Key Type as OUI. The Key Identifier MUST use an organization's reserved OUI space to indicate the Key and value structure.

```
 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
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      TBD3     |        Organizationally Unique Identifier     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|               Vendor Specific Values (variable)               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 3: OUI Key Type

2.3. Experimental Key Type

This section reserves a value in the Key Type Registry to indicate an Experimental Key Type.

As shown in Figure 4, the Key-Type will be used to identify the Key Type as Experimental. The Key Identifier will be used to identify the specific experimental Key and describe the structure of the contained values.

```
 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
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      TBD1     |          Experimental Key Identifier          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                 Experimental Values (variable)                |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 4: Experimental Key Type

3. IANA Considerations

This section requests that IANA help govern Key Types via the usual IANA registry procedures as per [RFC8126].

All values not suggested are available for assignment. The allocation of new values MUST be done via "Expert Review" procedures.

3.1. Key Type Registry

This section defines the Key Type Registry that is used to identify a specific Key Type. It also suggests values for Experimental, Well-Known, and OUI Key Types.

The range of valid values is 1 - 255.

0 is an illegal value and MUST NOT be allocated to or used by any implementation. It MUST be ignored on reception.

3.1.1. Requested Entries

Key Type	Value	Description
Experimental	TBD1	Indicates that the Key is Experimental.
Well-Known	TBD2	Indicates that the Key is Well-Known.
OUI	TBD3	Indicates that the Key is OUI.

Table 1

3.2. Experimental Key Type

This value indicates that a specific key is Experimental.

The range of valid values is 1 - 16777215 (2^24-1).

0 is an illegal value and MUST NOT be allocated to or used by any implementation. It MUST be ignored on reception.

3.2.1. Requested Entries

Experimental Key	Identifier	Description
Illegal	0	Not allowed.

Table 2

3.3. Well-Known Key Type

This value indicates that a specific key is Well-Known.

The range of valid values is 1 - 16777215 (2^24-1).

0 is an illegal value and MUST not be allocated to or used by any implementation. It MUST be ignored on reception.

3.3.1. Requested Entries

Well-Known Key	Identifier	Description
Illegal	0	Not allowed.
MAC/IP Binding	TBD1	To be defined.
FAM Security Roll-Over Key	TBD2	To be defined.

Table 3

3.4. OUI Key Type

This value indicates a specific OUI Key using an organization's reserved OUI space.

The range of valid values is 1 - 16777215 (2^24-1).

0 is an illegal value and MUST NOT be allocated to or used by any implementation. It MUST be ignored on reception.

3.4.1. Requested Entries

OUI Key	Identifier	Description
Illegal	0	Not allowed.

Table 4

4. Security Considerations

This document introduces no new security concerns to RIFT or other specifications referenced in this document given that the TIEs that carry KV pairs are already extensively secured by the RIFT [RIFT] specification itself.

5. Acknowledgements

To be provided.

6. 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>.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", June 2017, <https://www.rfc-editor.org/info/rfc8126>.

[RIFT] Przygienda, T., Sharma, A., Thubert, P., Rijsman, B., and D. Afanasiev, "RIFT: Routing in Fat Trees", Work in Progress, draft-ietf-rift-rift-12, May 2020.

Authors' Addresses

Jordan Head
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

RIFT WG Yuehua. Wei, Ed.
Internet-Draft Zheng. Zhang
Intended status: Informational ZTE Corporation
Expires: 24 July 2021 Dmitry. Afanasiev
Yandex
Tom. Verhaeg
Juniper Networks
Jaroslaw. Kowalczyk
Orange Polska
P. Thubert
Cisco Systems
20 January 2021

RIFT Applicability
draft-ietf-rift-applicability-04

Abstract

This document discusses the properties, applicability and operational considerations of RIFT in different network scenarios. It intends to provide a rough guide how RIFT can be deployed to simplify routing operations in Clos topologies and their variations.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

This document intends to explain the properties and applicability of "Routing in Fat Trees" [RIFT] in different deployment scenarios and highlight the operational simplicity of the technology compared to traditional routing solutions. It also documents special considerations when RIFT is used with or without overlays, with or without controllers, corrects topology mis-cablings, and node or link failures.

2. Problem Statement of Routing in Modern IP Fabric Fat Tree Networks

Clos [CLOS] and fat tree [FATTREE] topologies have gained prominence in today's networking, primarily as a result of the paradigm shift towards a centralized data-center based architecture that deliver a majority of computation and storage services.

Today's current routing protocols were geared towards a network with an irregular topology and low degree of connectivity originally. When they are applied to fat tree topologies:

* They tend to need extensive configuration or provisioning during bring up and re-dimensioning.

* Spine and leaf nodes have the entire network topology and routing information which is in fact not needed on the leaf nodes during normal operation.

* Significant Link State PDUs (LSPs) flooding duplication between spine nodes and leaf nodes occurs during network bring up and topology updates. It consumes both spine and leaf nodes' CPU and link bandwidth resources.

3. Applicability of RIFT to Clos IP Fabrics

Further content of this document assumes that the reader is familiar with the terms and concepts used in OSPF [RFC2328] and IS-IS [ISO10589-Second-Edition] link-state protocols. The sections of RIFT [RIFT] outline the requirements of routing in IP fabrics and RIFT protocol concepts.

3.1. Overview of RIFT

RIFT is a dynamic routing protocol for Clos and fat tree network topologies. It defines a link-state protocol when "pointing north" and path-vector protocol when "pointing south".

It floods flat link-state information northbound only so that each level obtains the full topology of levels south of it. That information is never flooded east-west or back south again. So a top tier node has full set of prefixes from the Shortest Path First (SPF) calculation.

In the southbound direction, the protocol operates like a "fully summarizing, unidirectional" path vector protocol or rather a distance vector with implicit split horizon. Routing information, normally just the default route, propagates one hop south and is 're-advertised' by nodes at next lower level.

```
                +-----------+          +-----------+
                |    ToF    |          |    ToF    |         LEVEL 2
      +         +-----+--+--+          +-+--+------+
      |         |     |  |  |          | |  |      |      ^
      +         |     |  |  +-------------------------+   |
      Distance  |  +-------------------+ |  |      |  |   |
      Vector    |  |  |  |               |  |      |  |   +
      South     |  |  |  |      +--------+  |      |  |   Link-state
      +         |  |  |  |      |           |      |  |   Flooding
      |         |  |  +-------------+       |      |  |   North
      v         |  |     |      |   |       |      |  |   +
              +-+--+-+   +------+   +-------+   +--+--+-+ |
              |SPINE |   |SPINE |   | SPINE |   | SPINE | |  LEVEL 1
      +       ++----++   ++---+-+   +--+--+-+   ++----+-+ |
      +        |    |     |   |        |  |      |    |   |     ^ N
      Distance |    +-------+ |        |  +--------+  |   |     |   E
      Vector   |          | | |        |         | |  |   |  +------>
      South    |  +-------+ | |        | +-------+ |  |   |     |
      +        |  |         | |        | |         |  |   |     +
      v       ++--++      +-+-++      ++-+-+     +-+--++  +
              |LEAF|      |LEAF|      |LEAF|     |LEAF |     LEVEL 0
              +----+      +----+      +----+     +-----+
```

Figure 1: Rift overview

A spine node has only information necessary for its level, which is all destinations south of the node based on SPF calculation, default route, and potential disaggregated routes.

RIFT combines the advantage of both link-state and distance vector:

* Fastest possible convergence

* Automatic detection of topology

* Minimal routes/info on tors

* High degree of ECMP

* Fast de-commissioning of nodes

* Maximum Propagation speed with flexible prefixes in an update

And RIFT eliminates the disadvantages of link-state or distance vector:

* Reduced and balanced flooding

* Automatic neighbor detection

So there are two types of link-state database which are "north representation" North Topology Information Elements (N-TIEs) and "south representation" South Topology Information Elements (S-TIEs). The N-TIEs contain a link-state topology description of lower levels and S-TIEs carry simply default routes for the lower levels.

There are more advantages unique to RIFT listed below which could be understood if you read the details of RIFT [RIFT].

* True ZTP

* Minimal blast radius on failures

* Can utilize all paths through fabric without looping

* Automatic disaggregation on failures

* Simple leaf implementation that can scale down to servers

* Key-Value store

* Horizontal links used for protection only

* Supports non-equal cost multipath and can replace MC-LAG

* Optimal flooding reduction and load-balancing

3.2. Applicable Topologies

Albeit RIFT is specified primarily for "proper" Clos or "fat tree" structures, it already supports Points of Delivery (PoD) concepts which are strictly speaking not found in original Clos concepts.

Further, the specification explains and supports operations of multi-plane Clos variants where the protocol relies on set of rings to allow the reconciliation of topology view of different planes as most desirable solution making proper disaggregation viable in case of failures. These observations hold not only in case of RIFT but also in the generic case of dynamic routing on Clos variants with multiple planes and failures in bi-sectional bandwidth, especially on the leafs.

3.2.1. Horizontal Links

RIFT is not limited to pure Clos divided into PoD and multi-planes but supports horizontal links below the top of fabric level. Those links are used only as routes of last resort northbound when a spine loses all northbound links or cannot compute a default route through them.

A possible configuration is a "ring" of horizontal links at a level. In presence of such a "ring" in any level (except Top of Fabric (ToF) level) neither North SPF (N-SPF) nor South SPF (S-SPF) will provide a "ring-based protection" scheme since such a computation would have to deal necessarily with breaking of "loops" in Dijkstra sense; an application for which RIFT is not intended.

A full-mesh connectivity between nodes on the same level can be employed and that allows N-SPF to provide for any node loosing all its northbound adjacencies (as long as any of the other nodes in the level are northbound connected) to still participate in northbound forwarding.

3.2.2. Vertical Shortcuts

Through relaxations of the specified adjacency forming rules, RIFT implementations can be extended to support vertical "shortcuts" as proposed by e.g. [I-D.white-distoptflood]. The RIFT specification itself does not provide the exact details since the resulting solution suffers from either much larger blast radius with increased flooding volumes or in case of maximum aggregation routing bow-tie problems.

3.2.3. Generalizing to any Directed Acyclic Graph

RIFT is an anisotropic routing protocol, meaning that it has a sense of direction (northbound, southbound, east-west) and that it operates differently depending on the direction.

* Northbound, RIFT operates as a link-state IGP, whereby the control packets are reflooded first all the way north and only interpreted later. All the individual fine grained routes are advertised.

* Southbound, RIFT operates as a distance vector IGP, whereby the control packets are flooded only one hop, interpreted, and the consequence of that computation is what gets flooded one more hop south. In the most common use-cases, a ToF node can reach most of the prefixes in the fabric. If that is the case, the ToF node advertises the fabric default and disaggregates the prefixes that it cannot reach. On the other hand, a ToF node that can reach only a small subset of the prefixes in the fabric will preferably advertise those prefixes and refrain from aggregating.

  In the general case, what gets advertised south is in more details:

  1. A fabric default that aggregates all the prefixes that are reachable within the fabric, and that could be a default route or a prefix that is dedicated to this particular fabric.

  2. The loopback addresses of the northbound nodes, e.g., for inband management.

  3. The disaggregated prefixes for the dynamic exceptions to the fabric default, advertised to route around the black hole that may form.

* East-west routing can optionally be used, with specific restrictions. It is useful in particular when a sibling has access to the fabric default but this node does not.

A Directed Acyclic Graph (DAG) provides a sense of north (the direction of the DAG) and of south (the reverse), which can be used to apply RIFT. For the purpose of RIFT, an edge in the DAG that has only incoming vertices is a ToF node.

There are a number of caveats though:

* The DAG structure must exist before RIFT starts, so there is a need for a companion protocol to establish the logical DAG structure.

* A generic DAG does not have a sense of east and west. The operation specified for east-west links and the southbound reflection between nodes are not applicable.

* In order to aggregate and disaggregate routes, RIFT requires that all the ToF nodes share the full knowledge of the prefixes in the fabric. This can be achieved with a ring as suggested by the RIFT main specification, by some preconfiguration, or using a synchronization with a common repository where all the active prefixes are registered.

3.3. Use Cases

3.3.1. Data Center Fabrics

RIFT is largely driven by demands and hence ideally suited for applying in data center (DC) IP fabrics underlay routing, vast majority of which seem to be currently (and for the foreseeable future) Clos architectures. It significantly simplifies operation and deployment of such fabrics as described in Section 4 for environments compared to extensive proprietary provisioning and operational solutions.

3.3.2. Metro Fabrics

The demand for bandwidth is increasing steadily, driven primarily by environments close to content producers (server farms connection via DC fabrics) but in proximity to content consumers as well. Consumers are often clustered in metro areas with their own network architectures that can benefit from simplified, regular Clos structures and hence RIFT.

3.3.3. Building Cabling

Commercial edifices are often cabled in topologies that are either Clos or its isomorphic equivalents. The Clos can grow rather high with many floors. That presents a challenge for traditional routing protocols (except BGP and by now largely phased-out PNNI) which do not support an arbitrary number of levels which RIFT does naturally. Moreover, due to the limited sizes of forwarding tables in network elements of building cabling，the minimum FIB size RIFT maintains under normal conditions is cost-effective in terms of hardware and operational costs.

3.3.4. Internal Router Switching Fabrics

It is common in high-speed communications switching and routing devices to use fabrics when a crossbar is not feasible due to cost, head-of-line blocking or size trade-offs. Normally such fabrics are not self-healing or rely on 1:/+1 protection schemes but it is conceivable to use RIFT to operate Clos fabrics that can deal effectively with interconnections or subsystem failures in such module. RIFT is neither IP specific and hence any link addressing connecting internal device subnets is conceivable.

3.3.5. CloudCO

The Cloud Central Office (CloudCO) is a new stage of telecom Central Office. It takes the advantage of Software Defined Networking (SDN) and Network Function Virtualization (NFV) in conjunction with general purpose hardware to optimize current networks. The following figure illustrates this architecture at a high level. It describes a single instance or macro-node of cloud CO. An Access I/O module faces a Cloud CO access node, and the Customer Premises Equipments (CPEs) behind it. A Network I/O module is facing the core network. The two I/O modules are interconnected by a leaf and spine fabric. [TR-384]

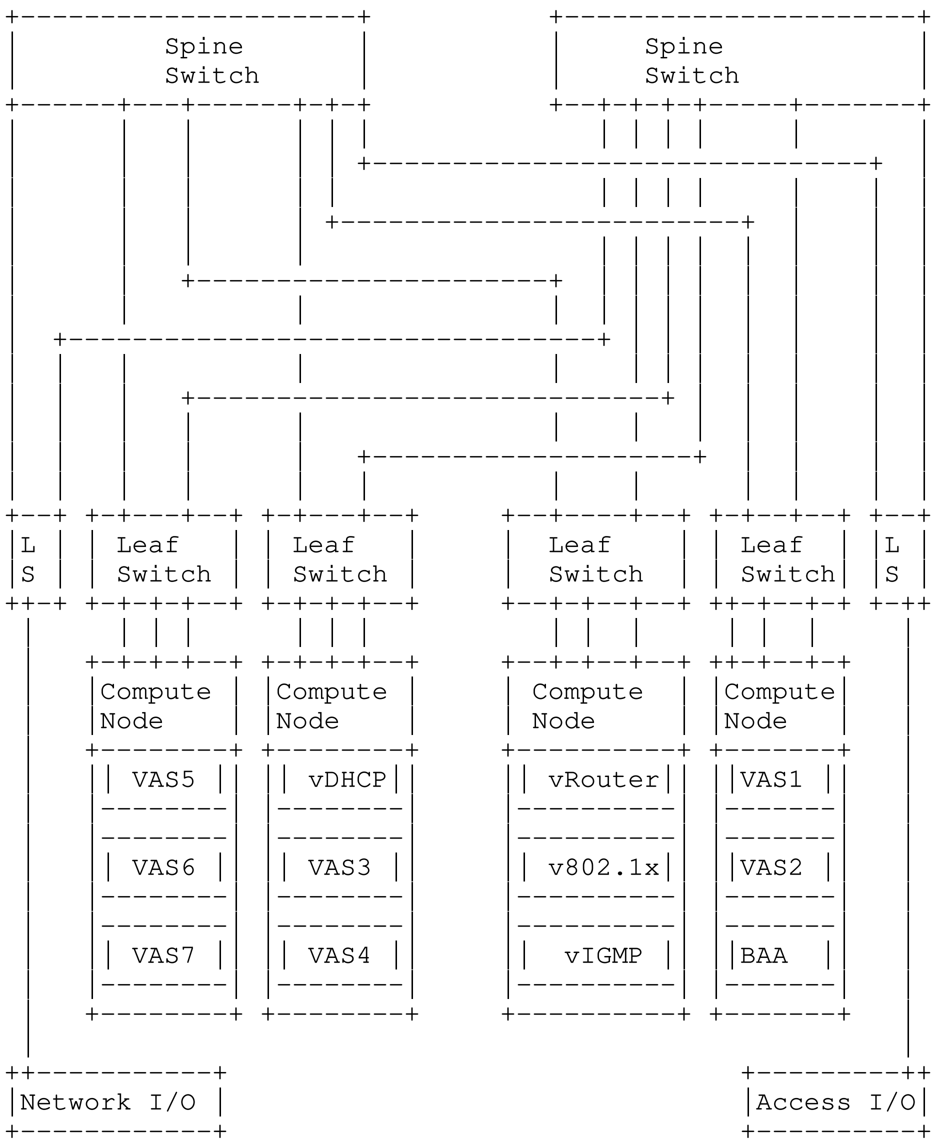

Figure 2: An example of CloudCO architecture

The Spine-Leaf architecture deployed inside CloudCO meets the network requirements of adaptable, agile, scalable and dynamic.

4. Deployment Considerations

RIFT presents the opportunity for organizations building and operating IP fabrics to simplify their operation and deployments while achieving many desirable properties of a dynamic routing on such a substrate:

* RIFT only foods routing information to the devices that absolutely need it. RIFT design follows minimum blast radius and minimum necessary epistemological scope philosophy which leads to good scaling properties while delivering maximum reactiveness.

* RIFT allows for extensive Zero Touch Provisioning within the protocol. In its most extreme version RIFT does not rely on any specific addressing and for IP fabric can operate using IPv6 ND [RFC4861] only.

* RIFT has provisions to detect common IP fabric mis-cabling scenarios.

* RIFT negotiates automatically BFD per link allowing this way for IP and micro-BFD [RFC7130] to replace Link Aggregation Groups (LAGs) which do hide bandwidth imbalances in case of constituent failures. Further automatic link validation techniques similar to [RFC5357] could be supported as well.

* RIFT inherently solves many difficult problems associated with the use of traditional routing topologies with dense meshes and high degrees of ECMP by including automatic bandwidth balancing, flood reduction and automatic disaggregation on failures while providing maximum aggregation of prefixes in default scenarios.

* RIFT reduces FIB size towards the bottom of the IP fabric where most nodes reside and allows with that for cheaper hardware on the edges and introduction of modern IP fabric architectures that encompass e.g. server multi-homing.

* RIFT provides valley-free routing and with that is loop free. This allows the use of any such valley-free path in bi-sectional fabric bandwidth between two destination irrespective of their metrics which can be used to balance load on the fabric in different ways.

* RIFT includes a key-value distribution mechanism which allows for many future applications such as automatic provisioning of basic overlay services or automatic key roll-overs over whole fabrics.

* RIFT is designed for minimum delay in case of prefix mobility on the fabric.

* Many further operational and design points collected over many years of routing protocol deployments have been incorporated in RIFT such as fast flooding rates, protection of information lifetimes and operationally easily recognizable remote ends of links and node names.

4.1. South Reflection

South reflection is a mechanism that South Node TIEs are "reflected" back up north to allow nodes in same level without East-west links to "see" each other.

For example, Spine111\Spine112\Spine121\Spine122 reflects Node S-TIEs from ToF21 to ToF22 separately. Respectively, Spine111\Spine112\Spine121\Spine122 reflects Node S-TIEs from ToF22 to ToF21 separately. So ToF22 and ToF21 see each other's node information as level 2 nodes.

In an equivalent fashion, as the result of the south reflection between Spine121-Leaf121-Spine122 and Spine121-Leaf122-Spine122, Spine121 and Spine 122 knows each other at level 1.

4.2. Suboptimal Routing on Link Failures

```
                  +--------+          +--------+
                  | ToF21  |          |  ToF22 |                LEVEL 2
                  ++--+-+-++          ++-+--+-++
                   |  | | |            | |  | +
                   |  | | |            | |  | linkTS8
     +-------------+  | +-+linkTS3+-+  | |  | +--------------+
     |                |   |         |  | |  +               |
     |    +----------------------------+ |  linkTS7         |
     |    |           |   |         +    +  +               |
     |    |           |   +-------+linkTS4+------------+    |
     |    |           |             +    +  |          |    |
     |    |           |     +------------+--+          |    |
     |    |           |     |       |  linkTS6         |    |
   +-+----+-+         +-----+--+   ++--------+       +-+----+-+
   |Spine111|         |Spine112|   |Spine121 |       |Spine122| LEVEL 1
   +-+---+--+         +----+---+   +-+---+---+       +-+---+--+
     |   |            |    |         |   |             |   |
     |   +--------------+  |         +   ++XX+linkSL6+---+ +
     |                | |  |      linkSL5              | | linkSL8
     |   +------------+ |  |         +   +---+linkSL7+-+ | +
     |   |              |  |         |   |               | |
   +-+---+-+         +--+--+-+     +-+---+-+          +--+-+--+
   |Leaf111|         |Leaf112|     |Leaf121|          |Leaf122| LEVEL 0
   +-+-----+         ++------+     +-----+-+          +-+-----+
     +                +                  +              +
   Prefix111         Prefix112     Prefix121          Prefix122
```

Figure 3: Suboptimal routing upon link failure use case

As shown in Figure 3, as the result of the south reflection between Spine121-Leaf121-Spine122 and Spine121-Leaf122-Spine122, Spine121 and Spine 122 knows each other at level 1.

Without disaggregation mechanism, when linkSL6 fails, the packet from leaf121 to prefix122 will probably go up through linkSL5 to linkTS3 then go down through linkTS4 to linkSL8 to Leaf122 or go up through linkSL5 to linkTS6 then go down through linkTS4 and linkSL8 to Leaf122 based on pure default route. It’s the case of suboptimal routing or bow-tieing.

With disaggregation mechanism, when linkSL6 fails, Spine122 will detect the failure according to the reflected node S-TIE from Spine121. Based on the disaggregation algorithm provided by RIFT, Spine122 will explicitly advertise prefix122 in Disaggregated Prefix S-TIE PrefixesElement(prefix122, cost 1). The packet from leaf121 to prefix122 will only be sent to linkSL7 following a longest-prefix match to prefix 122 directly then go down through linkSL8 to Leaf122 .

4.3. Black-Holing on Link Failures

```
                   +--------+          +--------+
                   | ToF 21 |          | ToF 22 |                LEVEL 2
                   ++-+--+-++          ++-+--+-++
                    | |  | |            | |  | +
                    | |  | |            | |  | linkTS8
     +--------------+ |  +-+linkTS3+X+  | |  | +--------------+
     linkTS1          |    |         |  | |  +                |
     +    +-----------------------------+ |  linkTS7          |
     |    |           +    |         +    +  +                |
     |    |      linkTS2   +-------+linkTS4+X+----------+     |
     |    +           +              +    +  |          |     |
     |   linkTS5      +-+    +------------+--+          |     |
     |    +             |    |       |  linkTS6         |     |
   +-+----+-+         +-+----+-+    ++-------+        +-+-----++
   |Spine111|         |Spine112|    |Spine121|        |Spine122| LEVEL 1
   +-+---+--+         ++----+--+    +-+---+--+        +-+---+--+
     |   |             |    |         |   |             |   |
     +   +---------------+  |         +   +---+linkSL6+---+ +
     linkSL1           | |  |      linkSL5              | | linkSL8
     +   +--+linkSL3+--+ |  |         +   +---+linkSL7+-+ | +
     |   |               |  |         |   |               | |
   +-+---+-+          +--+--+-+     +-+---+-+          +--+-+---+
   |Leaf111|          |Leaf112|     |Leaf121|          |Leaf122| LEVEL 0
   +-+-----+          ++------+     +-----+-+          +-+-----+
     +                 +                  +              +
   Prefix111          Prefix112     Prefix121          Prefix122
```

Figure 4: Black-holing upon link failure use case

This scenario illustrates a case when double link failure occurs and with that black-holing can happen.

Without disaggregation mechanism, when linkTS3 and linkTS4 both fail, the packet from leaf111 to prefix122 would suffer 50% black-holing based on pure default route. The packet supposed to go up through linkSL1 to linkTS1 then go down through linkTS3 or linkTS4 will be dropped. The packet supposed to go up through linkSL3 to linkTS2 then go down through linkTS3 or linkTS4 will be dropped as well. It’s the case of black-holing.

With disaggregation mechanism, when linkTS3 and linkTS4 both fail, ToF22 will detect the failure according to the reflected node S-TIE of ToF21 from Spine111\Spine112. Based on the disaggregation algorithm provided by RITF, ToF22 will explicitly originate an S-TIE with prefix 121 and prefix 122, that is flooded to spines 111, 112, 121 and 122.

The packet from leaf111 to prefix122 will not be routed to linkTS1 or linkTS2. The packet from leaf111 to prefix122 will only be routed to linkTS5 or linkTS7 following a longest-prefix match to prefix122.

4.4. Zero Touch Provisioning (ZTP)

Each RIFT node may operate in zero touch provisioning (ZTP) mode. It has no configuration (unless it is a ToF at the top of the topology or it is desired to confine it to leaf role w/o leaf-2-leaf procedures). In such case RIFT will fully configure the node's level after it is attached to the topology.

The most important component for ZTP is the automatic level derivation procedure. All the ToF nodes are explicitly marked with TOP_OF_FABRIC flag which are initial 'seeds' needed for other ZTP nodes to derive their level in the topology. The derivation of the level of each node happens then based on Link Information Elements (LIEs) received from its neighbors whereas each node (with possibly exceptions of configured leafs) tries to attach at the highest possible point in the fabric. This guarantees that even if the diffusion front reaches a node from "below" faster than from "above", it will greedily abandon already negotiated level derived from nodes topologically below it and properly peer with nodes above.

4.5. Mis-cabling Examples

```
        +----------------+              +-----------------+
        |     ToF21      |       +------+      ToF22      |   LEVEL 2
        +-------+----+---+       |      +----+---+--------+
        |       |    |   |       |      |    |   |        |
        |       |    |   +----------------------------+   |
        |   +---------------------------+    |   |    |   |
        |   |   |    |           |           |   |    |   |
        |   |   |    |   +-----------------------+    |   |
        |   |   +------------------------+   |        |   |
        |   |        |   |       |       |   |        |   |
      +-+---+--+   +-+---+--+    |    +--+---+-+  +--+---+-+
      |Spine111|   |Spine112|    |    |Spine121|  |Spine122| LEVEL 1
      +-+---+--+   ++----+--+    |    +--+---+-+  +-+----+-+
        |   |       |    |       |       |   |       |    |
        |   +---------+  |     link-M    |   +---------+  |
        |           | |  |       |       |           | |  |
        |   +-------+ |  |       |       |   +-------+ |  |
        |   |         |  |       |       |   |         |  |
      +-+---+-+    +--+--+-+     |     +-+---+-+    +--+--+-+
      |Leaf111|    |Leaf112+-----+     |Leaf121|    |Leaf122| LEVEL 0
      +-------+    +-------+           +-------+    +-------+
```

Figure 5: A single plane mis-cabling example

Figure 5 shows a single plane mis-cabling example. It's a perfect fat tree fabric except link-M connecting Leaf112 to ToF22.

The RIFT control protocol can discover the physical links automatically and be able to detect cabling that violates fat tree topology constraints. It reacts accordingly to such mis-cabling attempts, at a minimum preventing adjacencies between nodes from being formed and traffic from being forwarded on those mis-cabled links. Leaf112 will in such scenario use link-M to derive its level (unless it is leaf) and can report links to Spine111 and Spine112 as mis-cabled unless the implementations allows horizontal links.

Figure 6 shows a multiple plane mis-cabling example. Since Leaf112 and Spine121 belong to two different PoDs, the adjacency between Leaf112 and Spine121 can not be formed. link-W would be detected and prevented.

```
      +-------+    +-------+           +-------+    +-------+
      |ToF  A1|    |ToF  A2|           |ToF  B1|    |ToF  B2| LEVEL 2
      +-------+    +-------+           +-------+    +-------+
      |       |    |       |           |       |    |       |
      |       |    |       +-----------------+ |    |       |
      |       +--------------------------+   | |    |       |
      |            |                   | |   | |    |       |
      |     +------+                   | |   | +------+     |
      |     |        +-----------------+ |   |      | |     |
      |     |        |   +--------------------------+ |     |
      |  A  |        | B |               | A |        |  B  |
      +-----+--+   +-+---+--+         +--+---+-+   +--+-----+
      |Spine111|   |Spine112|     +---+Spine121|   |Spine122| LEVEL 1
      +-+---+--+   ++----+--+     |   +--+---+-+   +-+----+-+
        |   |       |    |        |      |   |       |    |
        |   +---------+  |        |      |   +---------+  |
        |           | |  |      link-W   |           | |  |
        |   +-------+ |  |        |      |   +-------+ |  |
        |   |         |  |        |      |   |         |  |
      +-+---+-+    +--+--+-+      |    +-+---+-+    +--+--+-+
      |Leaf111|    |Leaf112+------+    |Leaf121|    |Leaf122| LEVEL 0
      +-------+    +-------+           +-------+    +-------+
     +--------PoD#1----------+       +---------PoD#2---------+
```

Figure 6: A multiple plane mis-cabling example

RIFT provides an optional level determination procedure in its Zero Touch Provisioning mode. Nodes in the fabric without their level configured determine it automatically. This can have possibly counter-intuitive consequences however. One extreme failure scenario is depicted in Figure 7 and it shows that if all northbound links of spine11 fail at the same time, spine11 negotiates a lower level than Leaf11 and Leaf12.

To prevent such scenario where leafs are expected to act as switches, LEAF_ONLY flag can be set for Leaf111 and Leaf112. Since level -1 is invalid, Spine11 would not derive a valid level from the topology in Figure 7. It will be isolated from the whole fabric and it would be up to the leafs to declare the links towards such spine as mis-cabled.

```
           +-------+    +-------+        +-------+    +-------+
           |ToF  A1|    |ToF  A2|        |ToF  A1|    |ToF  A2|
           +-------+    +-------+        +-------+    +-------+
           |       |    |       |                |            |
           |    +-------+       |                |            |
           +    +  |            |  ====>         |            |
           X    X  +------+     |                +------+     |
           +    +         |     |                       |     |
           +----+--+    +-+-----+                     +-+-----+
           |Spine11|    |Spine12|                     |Spine12|
           +-+---+-+    ++----+-+                     ++----+-+
             |   |       |    |                        |    |
             |   +---------+  |                        |    |
             |           | |  |                        |    |
             |   +-------+ |  |                +-------+    |
             |   |         |  |                |            |
           +-+---+-+    +--+--+-+        +-----+-+    +-----+-+
           |Leaf111|    |Leaf112|        |Leaf111|    |Leaf112|
           +-------+    +-------+        +-+-----+    +-+-----+
                                           |            |
                                           |   +--------+
                                           |   |
                                         +-+---+-+
                                         |Spine11|
                                         +-------+
```

Figure 7: Fallen spine

4.6. Positive vs. Negative Disaggregation

Disaggregation is the procedure whereby [RIFT] advertises a more specific route southwards as an exception to the aggregated fabric-default north. Disaggregation is useful when a prefix within the aggregation is reachable via some of the parents but not the others at the same level of the fabric. It is mandatory when the level is the ToF since a ToF node that cannot reach a prefix becomes a black hole for that prefix. The hard problem is to know which prefixes are reachable by whom.

In the general case, [RIFT] solves that problem by interconnecting the ToF nodes. So the ToF nodes can exchange the full list of prefixes that exist in the fabric and figure when a ToF node lacks reachability and to existing prefix. This requires additional ports at the ToF, typically 2 ports per ToF node to form a ToF-spanning ring. [RIFT] also defines the southbound reflection procedure that enables a parent to explore the direct connectivity of its peers, meaning their own parents and children; based on the advertisements received from the shared parents and children, it may enable the parent to infer the prefixes its peers can reach.

When a parent lacks reachability to a prefix, it may disaggregate the prefix negatively, i.e., advertise that this parent can be used to reach any prefix in the aggregation except that one. The Negative Disaggregation signaling is simple and functions transitively from ToF to top-of-pod (ToP) and then from ToP to Leaf. But it is hard for a parent to figure which prefix it needs to disaggregate, because it does not know what it does not know; it results that the use of a spanning ring at the ToF is required to operate the Negative Disaggregation. Also, though it is only an implementation problem, the programmation of the FIB is complex compared to normal routes, and may incur recursions.

The more classical alternative is, for the parents that can reach a prefix that peers at the same level cannot, to advertise a more specific route to that prefix. This leverages the normal longest prefix match in the FIB, and does not require a special implementation. But as opposed to the Negative Disaggregation, the Positive Disaggregation is difficult and inefficient to operate transitively.

Transitivity is not needed to a grandchild if all its parents received the Positive Disaggregation, meaning that they shall all avoid the black hole; when that is the case, they collectively build a ceiling that protects the grandchild. But until then, a parent that received a Positive Disaggregation may believe that some peers are lacking the reachability and readvertise too early, or defer and maintain a black hole situation longer than necessary.

In a non-partitioned fabric, all the ToF nodes see one another through the reflection and can figure if one is missing a child. In that case it is possible to compute the prefixes that the peer cannot reach and disaggregate positively without a ToF-spanning ring. The ToF nodes can also ascertain that the ToP nodes are connected each to at least a ToF node that can still reach the prefix, meaning that the transitive operation is not required.

The bottom line is that in a fabric that is partitioned (e.g., using multiple planes) and/or where the ToP nodes are not guaranteed to always form a ceiling for their children, it is mandatory to use the Negative Disaggregation. On the other hand, in a highly symmetrical and fully connected fabric, (e.g., a canonical Clos Network), the Positive Disaggregation methods allows to save the complexity and cost associated to the ToF-spanning ring.

Note that in the case of Positive Disaggregation, the first ToF node(s) that announces a more-specific route attracts all the traffic for that route and may suffer from a transient incast. A ToP node that defers injecting the longer prefix in the FIB, in order to receive more advertisements and spread the packets better, also keeps on sending a portion of the traffic to the black hole in the meantime. In the case of Negative Disaggregation, the last ToF node(s) that injects the route may also incur an incast issue; this problem would occur if a prefix that becomes totally unreachable is disaggregated, but doing so is mostly useless and is not recommended.

4.7. Mobile Edge and Anycast

When a physical or a virtual node changes its point of attachement in the fabric from a previous-leaf to a next-leaf, new routes must be installed that supersede the old ones. Since the flooding flows northwards, the nodes (if any) between the previous-leaf and the common parent are not immediately aware that the path via previous-leaf is obsolete, and a stale route may exist for a while. The common parent needs to select the freshest route advertisement in order to install the correct route via the next-leaf. This requires that the fabric determines the sequence of the movements of the mobile node.

On the one hand, a classical sequence counter provides a total order for a while but it will eventually wrap. On the other hand, a timestamp provides a permanent order but it may miss a movement that happens too quickly vs. the granularity of the timing information. It is not envisioned in the short term that the average fabric supports a Precision Time Protocol [IEEEstd1588], and the precision that may be available with the Network Time Protocol [RFC5905], in the order of 100 to 200ms, may not be necessarily enough to cover, e.g., the fast mobility of a Virtual Machine.

Section 4.3.3. "Mobility" of [RIFT] specifies an hybrid method that combines a sequence counter from the mobile node and a timestamp from the network taken at the leaf when the route is injected. If the timestamps of the concurrent advertisements are comparable (i.e., more distant than the precision of the timing protocol), then the timestamp alone is used to determine the relative freshness of the routes. Otherwise, the sequence counter from the mobile node, if available, is used. One caveat is that the sequence counter must not wrap within the precision of the timing protocol. Another is that the mobile node may not even provide a sequence counter, in which case the mobility itself must be slower than the precision of the timing.

Mobility must not be confused with anycast. In both cases, a same address is injected in RIFT at different leaves. In the case of mobility, only the freshest route must be conserved, since mobile node changed its point of attachment for a leaf to the next. In the case of anycast, the node may be either multihomed (attached to multiple leaves in parallel) or reachable beyond the fabric via multiple routes that are redistributed to different leaves; either way, in the case of anycast, the multiple routes are equally valid and should be conserved. Without further information from the redistributed routing protocol, it is impossible to sort out a movement from a redistribution that happens asynchronously on different leaves. [RIFT] expects that anycast addresses are advertised within the timing precision, which is typically the case with a low-precision timing and a multihomed node. Beyond that time interval, RIFT interprets the lag as a mobility and only the freshest route is retained.

When using IPv6 [RFC8200], RIFT suggests to leverage "Registration Extensions for IPv6 over Low-Power Wireless Personal Area Network (6LoWPAN) Neighbor Discovery (ND)" [RFC8505] as the IPv6 ND interaction between the mobile node and the leaf. This provides not only a sequence counter but also a lifetime and a security token that may be used to protect the ownership of an address [RFC8928]. When using [RFC8505], the parallel registration of an anycast address to multiple leaves is done with the same sequence counter, whereas the sequence counter is incremented when the point of attachement changes. This way, it is possible to differentiate a mobile node from a multihomed node, even when the mobility happens within the timing precision. It is also possible for a mobile node to be multihomed as well, e.g., to change only one of its points of attachement.

4.8. IPv4 over IPv6

RIFT allows advertising IPv4 prefixes over IPv6 RIFT network. IPv6 Address Family (AF) configures via the usual Neighbor Discovery (ND) mechanisms and then V4 can use V6 nexthops analogous to [RFC5549]. It is expected that the whole fabric supports the same type of forwarding of address families on all the links. RIFT provides an indication whether a node is v4 forwarding capable and implementations are possible where different routing tables are computed per address family as long as the computation remains loop-free.

```
                           +-----+        +-----+
                +---+---+  | ToF |        | ToF |
                    ^      +--+--+        +-----+
                    |      |  |           |     |
                    |      |  +-------------+   |
                    |      |     +---------+|   |
                    +      |     |          |   |
                   V6      +-----+        +-+---+
                Forwarding |Spine|        |Spine|
                    +      +--+--+        +-----+
                    |      |  |           |     |
                    |      |  +-------------+   |
                    |      |     +---------+|   |
                    |      |     |          |   |
                    v      +-----+        +-+---+
                +---+---+  |Leaf |        | Leaf|
                           +--+--+        +--+--+
                              |              |
                 IPv4 prefixes|              |IPv4 prefixes
                              |              |
                          +---+----+     +---+----+
                          |   V4   |     |   V4   |
                          | subnet |     | subnet |
                          +--------+     +--------+
```

Figure 8: IPv4 over IPv6

4.9. In-Band Reachability of Nodes

RIFT doesn't precondition that nodes of the fabric have reachable addresses. But the operational purposes to reach the internal nodes may exist. Figure 9 shows an example that the network management station (NMS) attaches to leaf1.

```
+-------+      +-------+
| ToF1  |      | ToF2  |
++---- ++      ++-----++
 |     |        |     |
 |     +----------+   |
 |     +--------+ |   |
 |     |          |   |
++-----++      +--+---++
|Spine1 |      |Spine2 |
++-----++      ++-----++
 |     |        |     |
 |     +----------+   |
 |     +--------+ |   |
 |     |          |   |
++-----++      +--+---++
| Leaf1 |      | Leaf2 |
+---+---+      +-------+
    |
    |NMS
```

Figure 9: In-Band reachability of node

If NMS wants to access Leaf2, it simply works. Because loopback address of Leaf2 is flooded in its Prefix North TIE.

If NMS wants to access Spine2, it simply works too. Because spine node always advertises its loopback address in the Prefix North TIE. NMS may reach Spine2 from Leaf1-Spine2 or Leaf1-Spine1-ToF1/ ToF2-Spine2.

If NMS wants to access ToF2, ToF2's loopback address needs to be injected into its Prefix South TIE. This TIE must be seen by all nodes at the level below - the spine nodes in Figure 9 - that must form a ceiling for all the traffic coming from below (south). Otherwise, the traffic from NMS may follow the default route to the wrong ToF Node, e.g., ToF1.

In a fully connected ToF, in case of failure between ToF2 and spine nodes, ToF2's loopback address must be disaggregated recursively all the way to the leaves.

In a partitioned ToF, a TOF node is only reachable within its Plane, and the disaggregation to the leaves is also required. A possible alternative is to use the ring that interconnects the ToF nodes to transmit packets between them for their loopback addresses only. The idea is that this is mostly control traffic and should not alter the load balancing properties of the fabric.

4.10. Dual Homing Servers

Each RIFT node may operate in Zero Touch Provisioning (ZTP) mode. It has no configuration (unless it is a Top-of-Fabric at the top of the topology or the must operate in the topology as leaf and/or support leaf-2-leaf procedures) and it will fully configure itself after being attached to the topology.

```
                 +---+         +---+         +---+
                 |ToF|         |ToF|         |ToF|      ToF
                 +---+         +---+         +---+
                 |   |         |   |         |   |
                 |   +----------------+      |   |
                 |             |   |  |      |   |
                 |          +----------------+   |
                 |          |  |   |  |          |
                 +----------+--+   +--+----------+
                 |     ToR1    |   |     ToR2    |      Spine
                 +--+------+---+   +--+-------+--+
             +---+  |      |   |   |  |       |  +---+
             |      |      |   |   |  |       |      |
             |   +-----------------+  |       |      |
             |   |  |   +-------------+       |      |
             +   |  +   |  |   |-----------------+   |
             X   |  X   |  +--------x-----+   |  X   |
             +   |  +   |                 |   |  +   |
             +---+  +---+                 +---+  +---+
             |   |  |   |                 |   |  |   |
             +---+  +---+  ...............+---+  +---+
             SV(1) SV(2)                 SV(n+1) SV(n)  Leaf
```

Figure 10: Dual-homing servers

In the single plane, the worst condition is disaggregation of every other servers at the same level. Suppose the links from ToR1 (Top of Rack) to all the leaves become not available. All the servers' routes are disaggregated and the FIB of the servers will be expanded with n-1 more specific routes.

Sometimes, people may prefer to disaggregate from ToR to servers from start on, i.e. the servers have couple tens of routes in FIB from start on beside default routes to avoid breakages at rack level. Full disaggregation of the fabric could be achieved by configuration supported by RIFT.

4.11. Fabric With A Controller

There are many different ways to deploy the controller. One possibility is attaching a controller to the RIFT domain from ToF and another possibility is attaching a controller from the leaf.

```
                                     +------------+
                                     | Controller |
                                     ++----------++
                                      |          |
                                      |          |
                                 +----++        ++----+
                     -------     | ToF |        | ToF |
                        |        +--+--+        +-----+
                        |        |  |           |     |
                        |        |  +-------------+   |
                        |        |     +--------+ |   |
                        |        |     |          |   |
                                 +-----+        +-+---+
                    RIFT domain  |Spine|        |Spine|
                                 +--+--+        +-----+
                        |        |  |           |     |
                        |        |  +-------------+   |
                        |        |     +--------+ |   |
                        |        |     |          |   |
                        |        +-----+        +-+---+
                     -------     |Leaf |        | Leaf|
                                 +-----+        +-----+
```

Figure 11: Fabric with a controller

4.11.1. Controller Attached to ToFs

If a controller is attaching to the RIFT domain from ToF, it usually uses dual-homing connections. The loopback prefix of the controller should be advertised down by the ToF and spine to leaves. If the controller loses link to ToF, make sure the ToF withdraw the prefix of the controller(use different mechanisms).

4.11.2. Controller Attached to Leaf

If the controller is attaching from a leaf to the fabric, no special provisions are needed.

4.12. Internet Connectivity With Underlay

If global addressing is running without overlay, an external default route needs to be advertised through rift fabric to achieve internet connectivity. For the purpose of forwarding of the entire rift fabric, an internal fabric prefix needs to be advertised in the South Prefix TIE by ToF and spine nodes.

4.12.1. Internet Default on the Leaf

In case that an internet access request comes from a leaf and the internet gateway is another leaf, the leaf node as the internet gateway needs to advertise a default route in its Prefix North TIE.

4.12.2. Internet Default on the ToFs

In case that an internet access request comes from a leaf and the internet gateway is a ToF, the ToF and spine nodes need to advertise a default route in the Prefix South TIE.

4.13. Subnet Mismatch and Address Families

```
+--------+                     +--------+
|        |  LIE         LIE    |        |
|   A    | +---->       <----+ |   B    |
|        +---------------------+        |
+--------+                     +--------+
   X/24                           Y/24
```

Figure 12: subnet mismatch

LIEs are exchanged over all links running RIFT to perform Link (Neighbor) Discovery. A node MUST NOT originate LIEs on an address family if it does not process received LIEs on that family. LIEs on same link are considered part of the same negotiation independent on the address family they arrive on. An implementation MUST be ready to accept TIEs on all addresses it used as source of LIE frames.

As shown in the above figure, without further checks adjacency of node A and B may form, but the forwarding between node A and node B may fail because subnet X mismatches with subnet Y.

To prevent this a RIFT implementation should check for subnet mismatch just like e.g. ISIS does. This can lead to scenarios where an adjacency, despite exchange of LIEs in both address families may end up having an adjacency in a single AF only. This is a consideration especially in Section 4.8 scenarios.

4.14. Anycast Considerations

```
                                 + traffic
                                 |
                                 v
                          +------+------+
                          |     ToF     |
                          +---+-----+---+
                          |   |     |   |
             +------------+   |     |   +------------+
             |                |     |                |
         +---+---+    +-------+     +-------+    +---+---+
         |       |    |       |     |       |    |       |
         |Spine11|    |Spine12|     |Spine21|    |Spine22| LEVEL 1
         +-+---+-+    ++----+-+     +-+---+-+    ++----+-+
           |   |       |    |         |   |       |    |
           |   +---------+  |         |   +---------+  |
           |           | |  |         |           | |  |
           |   +-------+ |  |         |   +-------+ |  |
           |   |         |  |         |   |         |  |
         +-+---+-+    +--+--+-+     +-+---+-+    +--+--+-+
         |       |    |       |     |       |    |       |
         |Leaf111|    |Leaf112|     |Leaf121|    |Leaf122| LEVEL 0
         +-+-----+    ++------+     +-----+-+    +-----+-+
           +           +                  +      ^     |
         PrefixA      PrefixB         PrefixA    | PrefixC
                                                 |
                                                 + traffic
```

Figure 13: Anycast

If the traffic comes from ToF to Leaf111 or Leaf121 which has anycast prefix PrefixA. RIFT can deal with this case well. But if the traffic comes from Leaf122, it arrives Spine21 or Spine22 at level 1. But Spine21 or Spine22 doesn’t know another PrefixA attaching Leaf111. So it will always get to Leaf121 and never get to Leaf111. If the intension is that the traffic should been offloaded to Leaf111, then use policy guided prefixes defined in "Routing in Fat Trees" [RIFT].

4.15. IoT Applicability

The design of RIFT inherits from RPL [RFC6550] the anisotropic design of a default route upwards (northwards); it also inherits the capability to inject external host routes at the Leaf level using Wireless ND (WiND) [RFC8505][RFC8928] between a RIFT-agnostic host and a RIFT router. Both the RPL and the RIFT protocols are meant for large scale, and WiND enables device mobility at the edge the same way in both cases.

The main difference between RIFT and RPL is that with RPL, there's a single Root, whereas RIFT has many ToF nodes. The adds huge capabilities for leaf-2-leaf ECMP paths, but additional complexity with the need to disaggregate. Also RIFT uses Link State flooding northwards, and is not designed for low-power operation.

Still nothing prevents that the IP devices connected at the Leaf are IoT (Internet of Things) devices, which typically expose their address using WiND - which is an upgrade from 6LoWPAN ND [RFC6775].

A network that serves high speed/ high power IoT devices should typically provide deterministic capabilities for applications such as high speed control loops or movement detection. The Fat Tree is highly reliable, and in normal condition provides an equilatent multipath operation; but the ECMP doesn't provide hard guarantees for either delivery or latency. As long as the fabric is non-blocking the result is the same; but there can be load unbalances resulting in incast and possibly congestion loss that will prevent the delivery within bounded latency.

This could be alleviated with Packet Replication, Elimination and Reordering (PREOF) [RFC8655] leaf-2-leaf but PREOF is hard to provide at the scale of all flows, and the replication may increase the probability of the overload that it attempts to solve.

Note that the load balancing is not RIFT's problem, but it is key to serve IoT adequately.

5. Security Considerations

This document presents applicability of RIFT. As such, it does not introduce any security considerations. However, there are a number of security concerns at [RIFT].

6. Contributors

The following people (listed in alphabetical order) contributed significantly to the content of this document and should be considered co-authors:

Tony Przygienda

Juniper Networks

1194 N. Mathilda Ave

Sunnyvale, CA 94089

US

Email: prz@juniper.net

7. Normative References

[ISO10589-Second-Edition]
International Organization for Standardization, "Intermediate system to Intermediate system intra-domain routeing information exchange protocol for use in conjunction with the protocol for providing the connectionless-mode Network Service (ISO 8473)", November 2002.

[TR-384] Broadband Forum Technical Report, "TR-384 Cloud Central Office Reference Architectural Framework", January 2018.

[RFC2328] Moy, J., "OSPF Version 2", STD 54, RFC 2328, DOI 10.17487/RFC2328, April 1998, <https://www.rfc-editor.org/info/rfc2328>.

[RFC4861] Narten, T., Nordmark, E., Simpson, W., and H. Soliman, "Neighbor Discovery for IP version 6 (IPv6)", RFC 4861, DOI 10.17487/RFC4861, September 2007, <https://www.rfc-editor.org/info/rfc4861>.

[RFC5357] Hedayat, K., Krzanowski, R., Morton, A., Yum, K., and J. Babiarz, "A Two-Way Active Measurement Protocol (TWAMP)", RFC 5357, DOI 10.17487/RFC5357, October 2008, <https://www.rfc-editor.org/info/rfc5357>.

[RFC7130] Bhatia, M., Ed., Chen, M., Ed., Boutros, S., Ed., Binderberger, M., Ed., and J. Haas, Ed., "Bidirectional Forwarding Detection (BFD) on Link Aggregation Group (LAG) Interfaces", RFC 7130, DOI 10.17487/RFC7130, February 2014, <https://www.rfc-editor.org/info/rfc7130>.

[RFC5549] Le Faucheur, F. and E. Rosen, "Advertising IPv4 Network Layer Reachability Information with an IPv6 Next Hop", RFC 5549, DOI 10.17487/RFC5549, May 2009, <https://www.rfc-editor.org/info/rfc5549>.

[RFC6550] Winter, T., Ed., Thubert, P., Ed., Brandt, A., Hui, J., Kelsey, R., Levis, P., Pister, K., Struik, R., Vasseur, JP., and R. Alexander, "RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks", RFC 6550, DOI 10.17487/RFC6550, March 2012, <https://www.rfc-editor.org/info/rfc6550>.

[RFC6775] Shelby, Z., Ed., Chakrabarti, S., Nordmark, E., and C. Bormann, "Neighbor Discovery Optimization for IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs)", RFC 6775, DOI 10.17487/RFC6775, November 2012, <https://www.rfc-editor.org/info/rfc6775>.

[RFC8655] Finn, N., Thubert, P., Varga, B., and J. Farkas, "Deterministic Networking Architecture", RFC 8655, DOI 10.17487/RFC8655, October 2019, <https://www.rfc-editor.org/info/rfc8655>.

[RIFT] Przygienda, T., Sharma, A., Thubert, P., Rijsman, B., and D. Afanasiev, "RIFT: Routing in Fat Trees", Work in Progress, Internet-Draft, draft-ietf-rift-rift-12, 26 May 2020, <https://tools.ietf.org/html/draft-ietf-rift-rift-12>.

[I-D.white-distoptflood] White, R., Hegde, S., and S. Zandi, "IS-IS Optimal Distributed Flooding for Dense Topologies", Work in Progress, Internet-Draft, draft-white-distoptflood-04, 27 July 2020, <https://tools.ietf.org/html/draft-white-distoptflood-04>.

8. Informative References

[IEEEstd1588]
IEEE standard for Information Technology, "IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems", <https://standards.ieee.org/standard/1588-2019.html>.

[CLOS] Yuan, X., "On Nonblocking Folded-Clos Networks in Computer Communication Environments", IEEE International Parallel & Distributed Processing Symposium, 2011.

[FATTREE] Leiserson, C. E., "Fat-Trees: Universal Networks for Hardware-Efficient Supercomputing", 1985.

[RFC5905] Mills, D., Martin, J., Ed., Burbank, J., and W. Kasch, "Network Time Protocol Version 4: Protocol and Algorithms Specification", RFC 5905, DOI 10.17487/RFC5905, June 2010, <https://www.rfc-editor.org/info/rfc5905>.

[RFC8200] Deering, S. and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification", STD 86, RFC 8200, DOI 10.17487/RFC8200, July 2017, <https://www.rfc-editor.org/info/rfc8200>.

[RFC8505] Thubert, P., Ed., Nordmark, E., Chakrabarti, S., and C. Perkins, "Registration Extensions for IPv6 over Low-Power Wireless Personal Area Network (6LoWPAN) Neighbor Discovery", RFC 8505, DOI 10.17487/RFC8505, November 2018, <https://www.rfc-editor.org/info/rfc8505>.

[RFC8928] Thubert, P., Ed., Sarikaya, B., Sethi, M., and R. Struik, "Address-Protected Neighbor Discovery for Low-Power and Lossy Networks", RFC 8928, DOI 10.17487/RFC8928, November 2020, <https://www.rfc-editor.org/info/rfc8928>.

Authors' Addresses

Yuehua Wei (editor)
ZTE Corporation
No.50, Software Avenue
Nanjing
210012
China

Email: wei.yuehua@zte.com.cn

Zheng Zhang
ZTE Corporation
No.50, Software Avenue
Nanjing
210012
China

Email: zhang.zheng@zte.com.cn

Dmitry Afanasiev
Yandex

Email: fl0w@yandex-team.ru

Tom Verhaeg
Juniper Networks

Email: tverhaeg@juniper.net

Jaroslaw Kowalczyk
Orange Polska

Email: jaroslaw.kowalczyk2@orange.com

Pascal Thubert
Cisco Systems, Inc
Building D
45 Allee des Ormes - BP1200
06254 MOUGINS - Sophia Antipolis
France

Phone: +33 497 23 26 34
Email: pthubert@cisco.com

RIFT Working Group A. Przygienda, Ed.
Internet-Draft Juniper
Intended status: Standards Track A. Sharma
Expires: November 27, 2020 Comcast
P. Thubert
Cisco
Bruno. Rijsman
Individual
Dmitry. Afanasiev
Yandex
May 26, 2020

RIFT: Routing in Fat Trees
draft-ietf-rift-rift-12

Abstract

This document defines a specialized, dynamic routing protocol for Clos and fat-tree network topologies optimized towards minimization of configuration and operational complexity. The protocol

- o deals with no configuration, fully automated construction of fat-tree topologies based on detection of links,
- o minimizes the amount of routing state held at each level,
- o automatically prunes and load balances topology flooding exchanges over a sufficient subset of links,
- o supports automatic disaggregation of prefixes on link and node failures to prevent black-holing and suboptimal routing,
- o allows traffic steering and re-routing policies,
- o allows loop-free non-ECMP forwarding,
- o automatically re-balances traffic towards the spines based on bandwidth available and finally
- o provides mechanisms to synchronize a limited key-value data-store that can be used after protocol convergence to e.g. bootstrap higher levels of functionality on nodes.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on November 27, 2020.

Copyright Notice

Table of Contents

1. Authors

This work is a product of a list of individuals which are all to be considered major contributors independent of the fact whether their name made it to the limited boilerplate author's list or not.

Tony Przygienda, Ed. Juniper Networks	Alankar Sharma Comcast	Pascal Thubert Cisco
Bruno Rijsman Individual	Ilya Vershkov Mellanox	Dmitry Afanasiev Yandex
Don Fedyk Individual	Alia Atlas Individual	John Drake Juniper

Table 1: RIFT Authors

2. Introduction

Clos [CLOS] and Fat-Tree [FATTREE] topologies have gained prominence in today's networking, primarily as result of the paradigm shift towards a centralized data-center based architecture that is poised to deliver a majority of computation and storage services in the future. Today's current routing protocols were geared towards a network with an irregular topology and low degree of connectivity originally but given they were the only available options, consequently several attempts to apply those protocols to Clos have been made. Most successfully BGP [RFC4271] [RFC7938] has been extended to this purpose, not as much due to its inherent suitability but rather because the perceived capability to easily modify BGP and the immanent difficulties with link-state [DIJKSTRA] based protocols to optimize topology exchange and converge quickly in large scale densely meshed topologies. The incumbent protocols precondition normally extensive configuration or provisioning during bring up and re-dimensioning. This tends to be viable only for a set of organizations with according networking operation skills and budgets. For many IP fabric builders a desirable protocol would be one that auto-configures itself and deals with failures and mis-configurations with a minimum of human intervention only. Such a solution would allow local IP fabric bandwidth to be consumed in a 'standard component' fashion, i.e. provision it much faster and operate it at much lower costs than today, much like compute or storage is consumed already.

In looking at the problem through the lens of data center requirements, RIFT addresses challenges in IP fabric routing not through an incremental modification of either a link-state (distributed computation) or distance-vector (diffused computation) but rather a mixture of both, colloquially best described as "link-state towards the spine" and "distance vector towards the leaves". In other words, "bottom" levels are flooding their link-state information in the "northern" direction while each node generates under normal conditions a "default route" and floods it in the "southern" direction. This type of protocol allows naturally for highly desirable aggregation. Alas, such aggregation could blackhole traffic in cases of misconfiguration or while failures are being resolved or even cause partial network partitioning and this has to be addressed by some adequate mechanism. The approach RIFT takes is described in Section 4.2.5 and is basically based on automatic, sufficient disaggregation of prefixes in case of link and node failures.

For the visually oriented reader, Figure 1 presents a first level simplified view of the resulting information and routes on a RIFT fabric. The top of the fabric is holding in its link-state database the nodes below it and the routes to them. In the second row of the database table we indicate that partial information of other nodes in the same level is available as well. The details of how this is achieved will be postponed for the moment. When we look at the "bottom" of the fabric, the leaves, we see that the topology is basically empty and they only hold a load balanced default route to the next level under normal conditions.

The balance of this document details a dedicated IP fabric routing protocol, fills in the specification details and ultimately includes resulting security considerations.

```
   .                                  [A,B,C,D]
   .                                  [E]
   .             +-----+      +-----+
   .             |  E  |      |  F  | A/32 @ [C,D]
   .             +-+-+-+      +-+-+-+ B/32 @ [C,D]
   .               | |          | |   C/32 @ C
   .               | |    +-----+ |   D/32 @ D
   .               | |    |       |
   .               | +------+     |
   .               |      | |     |
   .       [A,B] +-+---+  | | +---+-+ [A,B]
   .       [D]   |  C  +--+ +-+  D  | [C]
   .             +-+-+-+      +-+-+-+
   .  0/0  @ [E,F] | |          | |   0/0  @ [E,F]
   .  A/32 @ A     | |    +-----+ |   A/32 @ A
   .  B/32 @ B     | |    |       |   B/32 @ B
   .               | +------+     |
   .               |      | |     |
   .             +-+---+  | | +---+-+
   .             |  A  +--+ +-+  B  |
   . 0/0 @ [C,D] +-----+      +-----+ 0/0 @ [C,D]
```

Figure 1: RIFT Information Distribution

2.1. 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 8174 [RFC8174].

3. Reference Frame

3.1. Terminology

This section presents the terminology used in this document. It is assumed that the reader is thoroughly familiar with the terms and concepts used in OSPF [RFC2328] and IS-IS [ISO10589-Second-Edition], [ISO10589] as well as the according graph theoretical concepts of shortest path first (SPF) [DIJKSTRA] computation and DAGs.

Crossbar: Physical arrangement of ports in a switching matrix without implying any further scheduling or buffering disciplines.

Clos/Fat Tree: This document uses the terms Clos and Fat Tree interchangeably whereas it always refers to a folded spine-and-leaf topology with possibly multiple Points of Delivery (PoDs) and one or multiple Top of Fabric (ToF) planes. Several modifications such as leaf-2-leaf shortcuts and multiple level shortcuts are possible and described further in the document.

Directed Acyclic Graph (DAG): A finite directed graph with no directed cycles (loops). If links in Clos are considered as either being all directed towards the top or vice versa, each of such two graphs is a DAG.

Folded Spine-and-Leaf: In case Clos fabric input and output stages are analogous, the fabric can be "folded" to build a "superspine" or top which we will call Top of Fabric (ToF) in this document.

Level: Clos and Fat Tree networks are topologically partially ordered graphs and 'level' denotes the set of nodes at the same height in such a network, where the bottom level (leaf) is the level with lowest value. A node has links to nodes one level down and/or one level up. Under some circumstances, a node may have links to nodes at the same level. As footnote: Clos terminology uses often the concept of "stage" but due to the folded nature of the Fat Tree we do not use it to prevent misunderstandings.

Superspine vs. Aggregation and Spine vs. Edge/Leaf: Traditional level names in 5-stages folded Clos for Level 2, 1 and 0 respectively. We normalize this language to talk about top-of-fabric (ToF), top-of-pod (ToP) and leaves.

Zero Touch Provisioning (ZTP): Optional RIFT mechanism which allows to derive node levels automatically based on minimum configuration (only ToF property has to be provisioned on according nodes).

Point of Delivery (PoD): A self-contained vertical slice or subset of a Clos or Fat Tree network containing normally only level 0 and level 1 nodes. A node in a PoD communicates with nodes in other PoDs via the Top-of-Fabric. We number PoDs to distinguish them and use PoD #0 to denote "undefined" PoD.

Top of PoD (ToP): The set of nodes that provide intra-PoD communication and have northbound adjacencies outside of the PoD, i.e. are at the "top" of the PoD.

Top of Fabric (ToF): The set of nodes that provide inter-PoD communication and have no northbound adjacencies, i.e. are at the "very top" of the fabric. ToF nodes do not belong to any PoD and are assigned "undefined" PoD value to indicate the equivalent of "any" PoD.

Spine: Any nodes north of leaves and south of top-of-fabric nodes. Multiple layers of spines in a PoD are possible.

Leaf: A node without southbound adjacencies. Its level is 0 (except cases where it is deriving its level via ZTP and is running without LEAF_ONLY which will be explained in Section 4.2.7).

Top-of-fabric Plane or Partition: In large fabrics top-of-fabric switches may not have enough ports to aggregate all switches south of them and with that, the ToF is 'split' into multiple independent planes. Introduction and Section 4.1.2 explains the concept in more detail. A plane is subset of ToF nodes that see each other through south reflection or E-W links.

Radix: A radix of a switch is basically number of switching ports it provides. It's sometimes called fanout as well.

North Radix: Ports cabled northbound to higher level nodes.

South Radix: Ports cabled southbound to lower level nodes.

South/Southbound and North/Northbound (Direction): When describing protocol elements and procedures, we will be using in different situations the directionality of the compass. I.e., 'south' or 'southbound' mean moving towards the bottom of the Clos or Fat Tree network and 'north' and 'northbound' mean moving towards the top of the Clos or Fat Tree network.

Northbound Link: A link to a node one level up or in other words, one level further north.

Southbound Link: A link to a node one level down or in other words, one level further south.

East-West Link: A link between two nodes at the same level. East-West links are normally not part of Clos or "fat-tree" topologies.

Leaf shortcuts (L2L): East-West links at leaf level will need to be differentiated from East-West links at other levels.

Routing on the host (RotH): Modern data center architecture variant where servers/leaves are multi-homed and consecutively participate in routing.

Northbound representation: Subset of topology information flooded towards higher levels of the fabric.

Southbound representation: Subset of topology information sent towards a lower level.

South Reflection: Often abbreviated just as "reflection" it defines a mechanism where South Node TIEs are "reflected" from the level south back up north to allow nodes in the same level without E-W links to "see" each other's node TIEs.

TIE: This is an acronym for a "Topology Information Element". TIEs are exchanged between RIFT nodes to describe parts of a network such as links and address prefixes, in a fashion similar to ISIS LSPs or OSPF LSAs. A TIE has always a direction and a type. We will talk about North TIEs (sometimes abbreviated as N-TIEs) when talking about TIEs in the northbound representation and South-TIEs (sometimes abbreviated as S-TIEs) for the southbound equivalent. TIEs have different types such as node and prefix TIEs.

Node TIE: This stands as acronym for a "Node Topology Information Element" that contains all adjacencies the node discovered and information about node itself. Node TIE should NOT be confused with a North TIE since "node" defines the type of TIE rather than its direction.

Prefix TIE: This is an acronym for a "Prefix Topology Information Element" and it contains all prefixes directly attached to this node in case of a North TIE and in case of South TIE the necessary default routes the node advertises southbound.

Key Value TIE: A South TIE that is carrying a set of key value pairs [DYNAMO]. It can be used to distribute information in the southbound direction within the protocol.

TIDE: Topology Information Description Element, equivalent to CSNP in ISIS.

TIRE: Topology Information Request Element, equivalent to PSNP in ISIS. It can both confirm received and request missing TIEs.

De-aggregation/Disaggregation: Process in which a node decides to advertise more specific prefixes Southwards, either positively to attract the corresponding traffic, or negatively to repel it. Disaggregation is performed to prevent black-holing and suboptimal routing to the more specific prefixes.

LIE: This is an acronym for a "Link Information Element", largely equivalent to HELLOs in IGPs and exchanged over all the links between systems running RIFT to form three way adjacencies.

Flood Repeater (FR): A node can designate one or more northbound neighbor nodes to be flood repeaters. The flood repeaters are responsible for flooding northbound TIEs further north. They are similar to MPR in OSLR. The document sometimes calls them flood leaders as well.

Bandwidth Adjusted Distance (BAD): Each RIFT node can calculate the amount of northbound bandwidth available towards a node compared to other nodes at the same level and can modify the route distance accordingly to allow for the lower level to adjust their load balancing towards spines.

Overloaded: Applies to a node advertising `overload` attribute as set. The semantics closely follow the meaning of the same attribute in [ISO10589-Second-Edition].

Interface: A layer 3 entity over which RIFT control packets are exchanged.

Three-Way Adjacency: RIFT tries to form a unique adjacency over an interface and exchange local configuration and necessary ZTP information. An adjacency is only advertised in node TIEs and used for computations after it achieved three-way state, i.e. both routers reflected each other in LIEs including relevant security information. LIEs before three-way state is reached may carry ZTP related information already.

Bi-directional Adjacency: Bidirectional adjacency is an adjacency where nodes of both sides of the adjacency advertised it in the node TIEs with the correct levels and system IDs. Bi-directionality is used to check in different algorithms whether the link should be included.

Neighbor: Once a three-way adjacency has been formed a neighborship relationship contains the neighbor's properties. Multiple adjacencies can be formed to a remote node via parallel interfaces but such adjacencies are NOT sharing a neighbor structure. Saying "neighbor" is thus equivalent to saying "a three-way adjacency".

Cost: The term signifies the weighted distance between two neighbors.

Distance: Sum of costs (bound by infinite distance) between two nodes.

Shortest-Path First (SPF): A well-known graph algorithm attributed to Dijkstra that establishes a tree of shortest paths from a source to destinations on the graph. We use SPF acronym due to its familiarity as general term for the node reachability calculations RIFT can employ to ultimately calculate routes of which Dijkstra algorithm is one.

North SPF (N-SPF): A reachability calculation that is progressing northbound, as example SPF that is using South Node TIEs only. Normally it progresses a single hop only and installs default routes.

South SPF (S-SPF): A reachability calculation that is progressing southbound, as example SPF that is using North Node TIEs only.

Security Envelope RIFT packets are flooded within an authenticated security envelope that allows to protect the integrity of information a node accepts.

3.2. Topology

```
               ^ N      +--------+          +--------+
Level 2        |        |ToF   21|          |ToF   22|
           E <-*-> W    ++-+--+-++          ++-+--+-++
               |         | |  | |            | |  | |
             S v      P111/2  P121/2         | |  | |
                         ^ ^  ^ ^            | |  | |
                         | |  | |            | |  | |
          +--------------+ |  +-----------+  | |  | +---------------+
          |                |    |         |  | |  |                 |
         South +-----------------------------+ |  |                 ^
          |    |           |    |         |    |  |             All TIEs
          0/0  0/0        0/0   +-----------------------------+     |
          v    v           v              |    |  |           |     |
          |    |           +-+    +<-0/0----------+           |     |
          |    |             |    |       |    |              |     |
        +-+----++ optional +-+----++     ++----+-+           ++-----++
Level 1 |       | E/W link |       |     |       |           |       |
        |Spin111+----------+Spin112|     |Spin121|           |Spin122|
        +-+---+-+          ++----+-+     +-+---+-+           ++---+--+
          |   |             |   South      |   |              |   |
          |   +---0/0--->-----+ 0/0        |   +----------------+ |
         0/0                | |  |         |                  | | |
          |   +---<-0/0-----+ |  v         |   +--------------+ | |
          v   |               |  |         |   |                | |
        +-+---+-+          +--+--+-+     +-+---+-+          +---+-+-+
Level 0 |       |  (L2L)   |       |     |       |          |       |
        |Leaf111+~~~~~~~~~~+Leaf112|     |Leaf121|          |Leaf122|
        +-+-----+          +-+---+-+     +--+--+-+          +-+-----+
          +                  +    \        /   +              +
          Prefix111   Prefix112    \      /   Prefix121    Prefix122
                                  multi-homed
                                    Prefix
        +---------- PoD 1 ---------+     +---------- PoD 2 ---------+
```

Figure 2: A Three Level Spine-and-Leaf Topology

```
.+--------+  +--------+  +--------+  +--------+
.|ToF   A1|  |ToF   B1|  |ToF   B2|  |ToF   A2|
.++-+-----+  ++-+-----+  ++-+-----+  ++-+-----+
. | |         | |         | |         | |
. | |         | |         | +---------------+
. | |         | |         |           | |   |
. | |         | +-------------------------+ |
. | |         |           |           | | | |
. | +-----------------------+         | | | |
. |           |           | |         | | | |
. |           | +---------+ | +---------+ | |
. |           | |           | |       |   | |
. | +---------------------------------+   | |
. | |         | |           | |           | |
.++-+-----+  ++-+-----+  +--+-+---+  +----+-+-+
.|Spine111|  |Spine112|  |Spine121|  |Spine122|
.+-+---+--+  ++----+--+  +-+---+--+  ++---+---+
.  |   |      |    |       |   |      |   |
.  |   +--------+  |       |   +--------+ |
.  |          | |  |       |          | | |
.  |   -------+ |  |       |   +------+ | |
.  |   |        |  |       |   |        | |
.+-+---+-+   +--+--+-+   +-+---+-+  +---+-+-+
.|Leaf111|   |Leaf112|   |Leaf121|  |Leaf122|
.+-------+   +-------+   +-------+  +-------+
```

Figure 3: Topology with Multiple Planes

We will use topology in Figure 2 (called commonly a fat tree/network in modern IP fabric considerations [VAHDAT08] as homonym to the original definition of the term [FATTREE]) in all further considerations. This figure depicts a generic "single plane fat-tree" and the concepts explained using three levels apply by induction to further levels and higher degrees of connectivity. Further, this document will deal also with designs that provide only sparser connectivity and "partitioned spines" as shown in Figure 3 and explained further in Section 4.1.2.

4. RIFT: Routing in Fat Trees

We present here a detailed outline of a protocol optimized for Routing in Fat Trees (RIFT) that in most abstract terms has many properties of a modified link-state protocol [RFC2328][ISO10589-Second-Edition] when distributing information northbound and distance vector [RFC4271] protocol when distributing information southbound. While this is an unusual combination, it does quite naturally exhibit the desirable properties we seek.

4.1. Overview

4.1.1. Properties

The most singular property of RIFT is that it floods flat link-state information northbound only so that each level obtains the full topology of levels south of it. Link-State information is, with some exceptions, never flooded East-West or back South again. Exceptions like south reflection is explained in detail in Section 4.2.5.1 and east-west flooding at ToF level in multi-plane fabrics is outlined in Section 4.1.2. In southbound direction, the protocol operates like a "fully summarizing, unidirectional" path vector protocol or rather a distance vector with implicit split horizon. Routing information, normally just the default route, propagates one hop south and is 're-advertised' by nodes at next lower level. However, RIFT uses flooding in the southern direction as well to avoid the overhead of building an update per adjacency. We omit describing the East-West direction for the moment.

Those information flow constraints create not only an anisotropic protocol (i.e. the information is not distributed "evenly" or "clumped" but summarized along the N-S gradient) but also a "smooth" information propagation where nodes do not receive the same information from multiple directions at the same time. Normally, accepting the same reachability on any link, without understanding its topological significance, forces tie-breaking on some kind of distance metric. And such tie-breaking leads ultimately in hop-by-hop forwarding to shortest paths only. In contrast to that, RIFT, under normal conditions, does not need to tie-break same reachability information from multiple directions. Its computation principles (south forwarding direction is always preferred) leads to valley-free forwarding behavior. And since valley free routing is loop-free, it can use all feasible paths which is another highly desirable property if available bandwidth should be utilized to the maximum extent possible.

To account for the "northern" and the "southern" information split the link state database is partitioned accordingly into "north representation" and "south representation" TIEs. In simplest terms the North TIEs contain a link state topology description of lower levels and and South TIEs carry simply default routes towards the level above. This oversimplified view will be refined gradually in following sections while introducing protocol procedures and state machines at the same time.

4.1.2. Generalized Topology View

This section will shed some light on the topologies RIFT addresses, including multi plane fabrics and their implications. Readers that are only interested in single plane designs, i.e. all top-of-fabric nodes being topologically equal and initially connected to all the switches at the level below them, can skip the rest of Section 4.1.2 and resulting Section 4.2.5.2 as well.

It is quite difficult to visualize multi plane design, which are effectively multi-dimensional switching matrices. To cope with that, we will introduce a methodology allowing us to depict the connectivity in two-dimensional pictures. Further, we will leverage the fact that we are dealing basically with stacked crossbar fabrics where ports align "on top of each other" in a regular fashion.

A word of caution to the reader; at this point it should be observed that the language used to describe Clos variations, especially in multi-plane designs, varies widely between sources. This description follows the terminology introduced in Section 3.1. It is unavoidable to have it present to be able to follow the rest of this section correctly.

4.1.2.1. Terminology

This section describes the terminology and acronyms used in the rest of the text.

P: Denotes the number of PoDs in a topology.

S: Denotes the number of ToF nodes in a topology.

K: Denotes the number of ports in radix of a switch pointing north or south. Further, K_LEAF denotes number of ports pointing south, i.e. towards leaves, and K_TOP for number of ports pointing north towards a higher spine level. To simplify the visual aids, notations and further considerations, K will be mostly set to Radix/2.

ToF Plane: Set of ToFs that are aware of each other by means of south reflection. We number planes by capital letters, e.g. plane A.

N: Denote the number of independent ToF planes in a topology.

R: Denotes a redundancy factor, i.e. number of connections a spine has towards a ToF plane. In single plane design K_TOP is equal to R.

Fallen Leaf: A fallen leaf in a plane Z is a switch that lost all connectivity northbound to Z.

4.1.2.2. Clos as Crossed Crossbars

The typical topology for which RIFT is defined is built of P number of PoDs and connected together by S number of ToF nodes. A PoD node has K number of ports (also called Radix). We consider half of them (K=Radix/2) as connecting host devices from the south, and the other half connecting to interleaved PoD Top-Level switches to the north. Ratio K can be chosen differently without loss of generality when port speeds differ or the fabric is oversubscribed but K=R/2 allows for more readable representation whereby there are as many ports facing north as south on any intermediate node. We represent a node hence in a schematic fashion with ports "sticking out" to its north and south rather than by the usual real-world front faceplate designs of the day.

Figure 4 provides a view of a leaf node as seen from the north, i.e. showing ports that connect northbound. For lack of a better symbol, we have chosen to use the "o" as ASCII visualisation of a single port. In this example, K_LEAF has 6 ports. Observe that the number of PoDs is not related to Radix unless the ToF Nodes are constrained to be the same as the PoD nodes in a particular deployment.

```
       Top view
        +---+
        |   |
        | o |     e.g., Radix = 12, K_LEAF = 6
        |   |
        | o |
        |   |      -------------------------
        | o ------- Physical Port (Ethernet) ----+
        |   |      -------------------------     |
        | o |                                    |
        |   |                                    |
        | o |                                    |
        |   |                                    |
        | o |                                    |
        |   |                                    |
        +---+                                    |

          ||             ||      ||      ||      ||      ||      ||
        +----+       +------------------------------------------------+
        |    |       |                                                |
        +----+       +------------------------------------------------+
          ||             ||      ||      ||      ||      ||      ||
              Side views
```

Figure 4: A Leaf Node, K_LEAF=6

The Radix of a PoD’s top node may be different than that of the leaf node. Though, more often than not, a same type of node is used for both, effectively forming a square (K*K). In general case, we could have switches with K_TOP southern ports on nodes at the top of the PoD which are not necessarily the same as K_LEAF. For instance, in the representations below, we pick a 6 port K_LEAF and a 8 port K_TOP. In order to form a crossbar, we need K_TOP Leaf Nodes as illustrated in Figure 5.

```
             +---+  +---+  +---+  +---+  +---+  +---+  +---+  +---+
             |   |  |   |  |   |  |   |  |   |  |   |  |   |  |   |
             | o |  | o |  | o |  | o |  | o |  | o |  | o |  | o |
             |   |  |   |  |   |  |   |  |   |  |   |  |   |  |   |
             | o |  | o |  | o |  | o |  | o |  | o |  | o |  | o |
             |   |  |   |  |   |  |   |  |   |  |   |  |   |  |   |
             | o |  | o |  | o |  | o |  | o |  | o |  | o |  | o |
             |   |  |   |  |   |  |   |  |   |  |   |  |   |  |   |
             | o |  | o |  | o |  | o |  | o |  | o |  | o |  | o |
             |   |  |   |  |   |  |   |  |   |  |   |  |   |  |   |
             | o |  | o |  | o |  | o |  | o |  | o |  | o |  | o |
             |   |  |   |  |   |  |   |  |   |  |   |  |   |  |   |
             | o |  | o |  | o |  | o |  | o |  | o |  | o |  | o |
             |   |  |   |  |   |  |   |  |   |  |   |  |   |  |   |
             +---+  +---+  +---+  +---+  +---+  +---+  +---+  +---+
```

Figure 5: Southern View of a PoD, K_TOP=8

As further visualized in Figure 6 the K_TOP Leaf Nodes are fully interconnected with the K_LEAF PoD-top nodes, providing connectivity that can be represented as a crossbar when "looked at" from the north. The result is that, in the absence of a failure, a packet entering the PoD from the north on any port can be routed to any port in the south of the PoD and vice versa. And that is precisely why it makes sense to talk about a "switching matrix".

```
                                      E<-*->W

          +---+  +---+  +---+  +---+  +---+  +---+  +---+  +---+
          |   |  |   |  |   |  |   |  |   |  |   |  |   |  |   |
        +---------------------------------------------------------+
        |   o      o      o      o      o      o      o      o    |
        +---------------------------------------------------------+
        +---------------------------------------------------------+
        |   o      o      o      o      o      o      o      o    |
        +---------------------------------------------------------+
        +---------------------------------------------------------+
        |   o      o      o      o      o      o      o      o    |
        +---------------------------------------------------------+
        +---------------------------------------------------------+
        |   o      o      o      o      o      o      o      o    |
        +---------------------------------------------------------+
        +---------------------------------------------------------+
        |   o      o      o      o      o      o      o      o    |<-+
        +---------------------------------------------------------+  |
        +---------------------------------------------------------+  |
        |   o      o      o      o      o      o      o      o    |  |
        +---------------------------------------------------------+  |
          |   |  |   |  |   |  |   |  |   |  |   |  |   |  |   |    |
          +---+  +---+  +---+  +---+  +---+  +---+  +---+  +---+    |
                     ^                                              |
                     |                                              |
                     |     __________       _____________________   |
                     +----- Leaf Node       PoD top Node (Spine) --+
                           __________       _____________________
```

Figure 6: Northern View of a PoD's Spines, K_TOP=8

Side views of this PoD is illustrated in Figure 7 and Figure 8.

```
                      Connecting to Spine

       ||      ||      ||      ||      ||      ||      ||      ||
  +----------------------------------------------------------------+   N
  |                    PoD top Node seen sideways                  |   ^
  +----------------------------------------------------------------+   |
       ||      ||      ||      ||      ||      ||      ||      ||      *
    +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+    |
    |     | |     | |     | |     | |     | |     | |     | |     |    v
    +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+    S
       ||      ||      ||      ||      ||      ||      ||      ||

                           Connecting to Client nodes
```

Figure 7: Side View of a PoD, K_TOP=8, K_LEAF=6

```
                   Connecting to Spine

           ||      ||      ||      ||      ||      ||
        +-----+ +-----+ +-----+ +-----+ +-----+ +-----+             N
        |     | |     | |     | |     | |     | |   PoD top Nodes   ^
        +-----+ +-----+ +-----+ +-----+ +-----+ +-----+             |
           ||      ||      ||      ||      ||      ||               *
      +-------------------------------------------------+             |
      |              Leaf seen sideways                 |             v
      +-------------------------------------------------+             S
           ||      ||      ||      ||      ||      ||

                   Connecting to Client nodes
```

Figure 8: Other Side View of a PoD, K_TOP=8, K_LEAF=6, 90o turn in E-W Plane

As next step, let us observe that a resulting PoD can be abstracted as a bigger node with a number K of K_POD= K_TOP * K_LEAF, and the design can recurse.

It will be critical at this point that, before progressing further, the concept and the picture of "crossed crossbars" is clear. Else, the following considerations might be difficult to comprehend.

To continue, the PoDs are interconnected with each other through a Top-of-Fabric (ToF) node at the very top or the north edge of the fabric. The resulting ToF is NOT partitioned if, and only if (IIF), every PoD top level node (spine) is connected to every ToF Node.

This topology is also referred to as a single plane configuration and is quite popular due to its simplicity. In order to reach a 1:1 connectivity ratio between the ToF and the leaves, it results that there are K_TOP ToF nodes, because each port of a ToP node connects to a different ToF node, and K_LEAF ToP nodes for the same reason. Consequently, it will take (P * K_LEAF) ports on a ToF node to connect to each of the K_LEAF ToP nodes of the P PoDs, as shown in Figure 9.

```
        [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] <-----+
         |   |   |   |   |   |   |   |        |
      [=================================]     |     -----------
         |   |   |   |   |   |   |   |        +----- Top-of-Fabric
        [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]       +----- Node      -------+
                                              |     -----------       |
                                              |                       v
        +-+ +-+ +-+ +-+ +-+ +-+ +-+ +-+ <-----+                      +-+
        | | | | | | | | | | | | | | | |                              | |
      [ |o| |o| |o| |o| |o| |o| |o| |o| ]                            | |
      [ |o| |o| |o| |o| |o| |o| |o| |o| ] -------------------------  | |
      [ |o| |o| |o| |o| |o| |o| |o| |o<--- Physical Port (Ethernet)  | |
      [ |o| |o| |o| |o| |o| |o| |o| |o| ] -------------------------  | |
      [ |o| |o| |o| |o| |o| |o| |o| |o| ]                            | |
      [ |o| |o| |o| |o| |o| |o| |o| |o| ]                            | |
        | | | | | | | | | | | | | | | |                              | |
      [ |o| |o| |o| |o| |o| |o| |o| |o| ]                            | |
      [ |o| |o| |o| |o| |o| |o| |o| |o| ]      --------------        | |
      [ |o| |o| |o| |o| |o| |o| |o| |o| ] <---  PoD top level        | |
      [ |o| |o| |o| |o| |o| |o| |o| |o| ]       node (Spine)  ---+   | |
      [ |o| |o| |o| |o| |o| |o| |o| |o| ]      --------------    |   | |
      [ |o| |o| |o| |o| |o| |o| |o| |o| ]                        |   | |
        | | | | | | | | | | | | | | | |  -+           +-   +-+   v   | |
      [ |o| |o| |o| |o| |o| |o| |o| |o| ] |           |  --| |--[ ]--| |
      [ |o| |o| |o| |o| |o| |o| |o| |o| ] |   -----   |  --| |--[ ]--| |
      [ |o| |o| |o| |o| |o| |o| |o| |o| ] +--- PoD ---+  --| |--[ ]--| |
      [ |o| |o| |o| |o| |o| |o| |o| |o| ] |   -----   |  --| |--[ ]--| |
      [ |o| |o| |o| |o| |o| |o| |o| |o| ] |           |  --| |--[ ]--| |
      [ |o| |o| |o| |o| |o| |o| |o| |o| ] |           |  --| |--[ ]--| |
        | | | | | | | | | | | | | | | |  -+           +-   +-+       | |
        | | | | | | | | | | | | | | | |                              | |
        +-+ +-+ +-+ +-+ +-+ +-+ +-+ +-+                              +-+
```

Figure 9: Fabric Spines and TOFs in Single Plane Design, 3 PoDs

The top view can be collapsed into a third dimension where the hidden depth index is representing the PoD number. We can then show one PoD as a class of PoDs and hence save one dimension in our representation. The Spine Node expands in the depth and the vertical dimensions, whereas the PoD top level Nodes are constrained, in horizontal dimension. A port in the 2-D representation represents effectively the class of all the ports at the same position in all the PoDs that are projected in its position along the depth axis. This is shown in Figure 10.

```
         / / / / / / / / / / / / / / / /
        / / / / / / / / / / / / / / / /
       / / / / / / / / / / / / / / / /
      / / / / / / / / / / / / / / / /   ]
     +-+ +-+ +-+ +-+ +-+ +-+ +-+ +-+   ]]
     | | | | | | | | | | | | | | | |  ]   ---------------------------
   [ |o| |o| |o| |o| |o| |o| |o| |o| ] <-- PoD top level node (Spine)
   [ |o| |o| |o| |o| |o| |o| |o| |o| ]    ---------------------------
   [ |o| |o| |o| |o| |o| |o| |o| |o| ]]]]
   [ |o| |o| |o| |o| |o| |o| |o| |o| ]]]     ^^
   [ |o| |o| |o| |o| |o| |o| |o| |o| ]]     //  PoDs
   [ |o| |o| |o| |o| |o| |o| |o| |o| ]     // (in depth)
     | |/| |/| |/| |/| |/| |/| |/| |/     //
     +-+ +-+ +-+/+-+/+-+ +-+ +-+ +-+     //
             ^
             |     ----------------
             +----- Top-of-Fabric Node
                   ----------------
```

Figure 10: Collapsed Northern View of a Fabric for Any Number of PoDs

As simple as single plane deployment is it introduces a limit due to the bound on the available radix of the ToF nodes that has to be at least P * K_LEAF. Nevertheless, we will see that a distinct advantage of a connected or non-partitioned Top-of-Fabric is that all failures can be resolved by simple, non-transitive, positive disaggregation (i.e. nodes advertising more specific prefixes with the default to the level below them that is however not propagated further down the fabric) as described in Section 4.2.5.1 . In other words; non-partitioned ToF nodes can always reach nodes below or withdraw the routes from PoDs they cannot reach unambiguously. And with this, positive disaggregation can heal all failures and still allow all the ToF nodes to see each other via south reflection. Disaggregation will be explained in further detail in Section 4.2.5.

In order to scale beyond the "single plane limit", the Top-of-Fabric can be partitioned by a N number of identically wired planes where N is an integer divider of K_LEAF. The 1:1 ratio and the desired symmetry are still served, this time with (K_TOP * N) ToF nodes, each of (P * K_LEAF / N) ports. N=1 represents a non-partitioned Spine and N=K_LEAF is a maximally partitioned Spine. Further, if R is any integer divisor of K_LEAF, then N=K_LEAF/R is a feasible number of planes and R a redundancy factor. If proves convenient for deployments to use a radix for the leaf nodes that is a power of 2 so they can pick a number of planes that is a lower power of 2. The example in Figure 11 splits the Spine in 2 planes with a redundancy factor R=3, meaning that there are 3 non-intersecting paths between any leaf node and any ToF node. A ToF node must have, in this case, at least 3*P ports, and be directly connected to 3 of the 6 PoD-ToP nodes (spines) in each PoD.

```
        +---+  +---+  +---+  +---+  +---+  +---+  +---+  +---+
      +-|   |--|   |--|   |--|   |--|   |--|   |--|   |--|   |-+
      | | o |  | o |  | o |  | o |  | o |  | o |  | o |  | o | |
      +-|   |--|   |--|   |--|   |--|   |--|   |--|   |--|   |-+
      +-|   |--|   |--|   |--|   |--|   |--|   |--|   |--|   |-+
      | | o |  | o |  | o |  | o |  | o |  | o |  | o |  | o | |
      +-|   |--|   |--|   |--|   |--|   |--|   |--|   |--|   |-+
      +-|   |--|   |--|   |--|   |--|   |--|   |--|   |--|   |-+
      | | o |  | o |  | o |  | o |  | o |  | o |  | o |  | o | |
      +-|   |--|   |--|   |--|   |--|   |--|   |--|   |--|   |-+
        +---+  +---+  +---+  +---+  +---+  +---+  +---+  +---+

      Plane 1
     ----------- . ------------ . ------------ . ------------ . --------
      Plane 2

        +---+  +---+  +---+  +---+  +---+  +---+  +---+  +---+
      +-|   |--|   |--|   |--|   |--|   |--|   |--|   |--|   |-+
      | | o |  | o |  | o |  | o |  | o |  | o |  | o |  | o | |
      +-|   |--|   |--|   |--|   |--|   |--|   |--|   |--|   |-+
      +-|   |--|   |--|   |--|   |--|   |--|   |--|   |--|   |-+
      | | o |  | o |  | o |  | o |  | o |  | o |  | o |  | o | |
      +-|   |--|   |--|   |--|   |--|   |--|   |--|   |--|   |-+
      +-|   |--|   |--|   |--|   |--|   |--|   |--|   |--|   |-+
      | | o |  | o |  | o |  | o |  | o |  | o |  | o |  | o | |
      +-|   |--|   |--|   |--|   |--|   |--|   |--|   |--|   |-+
        +---+  +---+  +---+  +---+  +---+  +---+  +---+  +---+
                 ^
                 |
                 |      ----------------
                 +----- Top-of-Fabric node
                        "across" depth
                        ----------------
```

Figure 11: Northern View of a Multi-Plane ToF Level, K_LEAF=6, N=2

At the extreme end of the spectrum it is even possible to fully partition the spine with N = K_LEAF and R=1, while maintaining connectivity between each leaf node and each Top-of-Fabric node. In that case the ToF node connects to a single Port per PoD, so it appears as a single port in the projected view represented in Figure 12. The number of ports required on the Spine Node is more or equal to P, the number of PoDs.

```
       Plane 1
         +---+  +---+  +---+  +---+  +---+  +---+  +---+  +---+  -+
       +-|   |--|   |--|   |--|   |--|   |--|   |--|   |--|   |-+ |
       | | o |  | o |  | o |  | o |  | o |  | o |  | o |  | o | | |
       +-|   |--|   |--|   |--|   |--|   |--|   |--|   |--|   |-+ |
         +---+  +---+  +---+  +---+  +---+  +---+  +---+  +---+    |
      ----------- . ------------------- . ------------ . -------- |
         +---+  +---+  +---+  +---+  +---+  +---+  +---+  +---+    |
       +-|   |--|   |--|   |--|   |--|   |--|   |--|   |--|   |-+ |
       | | o |  | o |  | o |  | o |  | o |  | o |  | o |  | o | | |
       +-|   |--|   |--|   |--|   |--|   |--|   |--|   |--|   |-+ |
         +---+  +---+  +---+  +---+  +---+  +---+  +---+  +---+    |
      ----------- . ------------ . ---- . ------------ . -------- |
         +---+  +---+  +---+  +---+  +---+  +---+  +---+  +---+    |
       +-|   |--|   |--|   |--|   |--|   |--|   |--|   |--|   |-+ |
       | | o |  | o |  | o |  | o |  | o |  | o |  | o |  | o | | |
       +-|   |--|   |--|   |--|   |--|   |--|   |--|   |--|   |-+ |
         +---+  +---+  +---+  +---+  +---+  +---+  +---+  +---+    |
      ----------- . ------------ . ------------------- . -------- +<-+
         +---+  +---+  +---+  +---+  +---+  +---+  +---+  +---+    |  |
       +-|   |--|   |--|   |--|   |--|   |--|   |--|   |--|   |-+ |  |
       | | o |  | o |  | o |  | o |  | o |  | o |  | o |  | o | | |  |
       +-|   |--|   |--|   |--|   |--|   |--|   |--|   |--|   |-+ |  |
         +---+  +---+  +---+  +---+  +---+  +---+  +---+  +---+    |  |
      ----------- . ------------ . ------------ . ---- . -------- |  |
         +---+  +---+  +---+  +---+  +---+  +---+  +---+  +---+    |  |
       +-|   |--|   |--|   |--|   |--|   |--|   |--|   |--|   |-+ |  |
       | | o |  | o |  | o |  | o |  | o |  | o |  | o |  | o | | |  |
       +-|   |--|   |--|   |--|   |--|   |--|   |--|   |--|   |-+ |  |
         +---+  +---+  +---+  +---+  +---+  +---+  +---+  +---+    |  |
      ----------- . ------------ . ------------ . --------------- |  |
         +---+  +---+  +---+  +---+  +---+  +---+  +---+  +---+    |  |
       +-|   |--|   |--|   |--|   |--|   |--|   |--|   |--|   |-+ |  |
       | | o |  | o |  | o |  | o |  | o |  | o |  | o |  | o | | |  |
       +-|   |--|   |--|   |--|   |--|   |--|   |--|   |--|   |-+ |  |
         +---+  +---+  +---+  +---+  +---+  +---+  +---+  +---+  -+  |
       Plane 6    ^                                                  |
                  |                                                  |
                  |     ----------------           -------------     |
                  +-----  ToF       Node           Class of PoDs  ---+
                        ----------------           -------------
```

Figure 12: Northern View of a Maximally Partitioned ToF Level, R=1

4.1.3. Fallen Leaf Problem

As mentioned earlier, RIFT exhibits an anisotropic behavior tailored for fabrics with a North / South orientation and a high level of interleaving paths. A non-partitioned fabric makes a total loss of connectivity between a Top-of-Fabric node at the north and a leaf node at the south a very rare but yet possible occasion that is fully healed by positive disaggregation as described in Section 4.2.5.1. In large fabrics or fabrics built from switches with low radix, the ToF ends often being partitioned in planes which makes the occurrence of having a given leaf being only reachable from a subset of the ToF nodes more likely to happen. This makes some further considerations necessary.

We define a "Fallen Leaf" as a leaf that can be reached by only a subset, but not all, of Top-of-Fabric nodes due to missing connectivity. If R is the redundancy factor, then it takes at least R breakages to reach a "Fallen Leaf" situation.

In a maximally partitioned fabric, the redundancy factor is R= 1, so any breakage in the fabric may cause one or more fallen leaves. However, not all cases require disaggregation. The following cases do not require particular action in such scenario:

> If a southern link on a node goes down, then connectivity through that node is lost for all nodes south of it. There is no need to disaggregate since the connectivity to this node is lost for all spine nodes in a same fashion.

> If a ToF Node goes down, then northern traffic towards it is routed via alternate ToF nodes in the same plane and there is no need to disaggregate routes.

In a general manner, the mechanism of non-transitive positive disaggregation is sufficient when the disaggregating ToF nodes collectively connect to all the ToP nodes in the broken plane. This happens in the following case:

> If the breakage is the last northern link from a ToP node to a ToF node going down, then the fallen leaf problem affects only The ToF node, and the connectivity to all the nodes in the PoD is lost from that ToF node. This can be observed by other ToF nodes within the plane where the ToP node is located and positively disaggregated within that plane.

On the other hand, there is a need to disaggregate the routes to Fallen Leaves in a transitive fashion, all the way to the other leaves in the following cases:

- If the breakage is the last northern link from a leaf node within a plane (there is only one such link in a maximally partitioned fabric) that goes down, then connectivity to all unicast prefixes attached to the leaf node is lost within the plane where the link is located. Southern Reflection by a leaf node, e.g., between ToP nodes, if the PoD has only 2 levels, happens in between planes, allowing the ToP nodes to detect the problem within the PoD where it occurs and positively disaggregate. The breakage can be observed by the ToF nodes in the same plane through the North flooding of TIEs from the ToP nodes. The ToF nodes however need to be aware of all the affected prefixes for the negative, possibly transitive disaggregation to be fully effective (i.e. a node advertising in control plane that it cannot reach a certain more specific prefix than default whereas such disaggregation must in extreme condition propagate further down southbound). The problem can also be observed by the ToF nodes in the other planes through the flooding of North TIEs from the affected leaf nodes, together with non-node North TIEs which indicate the affected prefixes. To be effective in that case, the positive disaggregation must reach down to the nodes that make the plane selection, which are typically the ingress leaf nodes. The information is not useful for routing in the intermediate levels.

- If the breakage is a ToP node in a maximally partitioned fabric - in which case it is the only ToP node serving the plane in that PoD - goes down, then the connectivity to all the nodes in the PoD is lost within the plane where the ToP node is located. Consequently, all leaves of the PoD fall in this plane. Since the Southern Reflection between the ToF nodes happens only within a plane, ToF nodes in other planes cannot discover fallen leaves in a different plane. They also cannot determine beyond their local plane whether a leaf node that was initially reachable has become unreachable. As the breakage can be observed by the ToF nodes in the plane where the breakage happened, the ToF nodes in the plane need to be aware of all the affected prefixes for the negative disaggregation to be fully effective. The problem can also be observed by the ToF nodes in the other planes through the flooding of North TIEs from the affected leaf nodes, if there are only 3 levels and the ToP nodes are directly connected to the leaf nodes, and then again it can only be effective it is propagated transitively to the leaf, and useless above that level.

For the sake of easy comprehension let us roll the abstractions back into a simple example and observe that in Figure 3 the loss of link Spine 122 to Leaf 122 will make Leaf 122 a fallen leaf for Top-of-Fabric plane B. Worse, if the cabling was never present in first place, plane B will not even be able to know that such a fallen leaf exists. Hence partitioning without further treatment results in two grave problems:

- o Leaf 111 trying to route to Leaf 122 MUST choose Spine 111 in plane A as its next hop since plane B will inevitably blackhole the packet when forwarding using default routes or do excessive bow tying. This information must be in its routing table.

- o Any kind of "flooding" or distance vector trying to deal with the problem by distributing host routes will be able to converge only using paths through leaves. The flooding of information on Leaf 122 would have to go up to Top-of-Fabric A and then "loopback" over other leaves to ToF B leading in extreme cases to traffic for Leaf 122 when presented to plane B taking an "inverted fabric" path where leaves start to serve as TOFs, at least for the duration of a protocol's convergence.

4.1.4. Discovering Fallen Leaves

As illustrated later, and without further proof, the way to deal with fallen leaves in multi-plane designs, when aggregation is used, is that RIFT requires all the ToF nodes to share the same north topology database. This happens naturally in single plane design by the means of northbound flooding and south reflection but needs additional considerations in multi-plane fabrics. To satisfy this RIFT, in multi-plane designs, relies at the ToF level on ring interconnection of switches in multiple planes. Other solutions are possible but they either need more cabling or end up having much longer flooding paths and/or single points of failure.

In detail, by reserving two ports on each Top-of-Fabric node it is possible to connect them together by interplane bi-directional rings as illustrated in Figure 13. The rings will be used to exchange full north topology information between planes. All ToFs having same north topology allows by the means of transitive, negative disaggregation described in Section 4.2.5.2 to efficiently fix any possible fallen leaf scenario. Somewhat as a side-effect, the exchange of information fulfills the ask to present full view of the fabric topology at the Top-of-Fabric level, without the need to collate it from multiple points by additional complexity of technologies like [RFC7752].

```
           +---+  +---+  +---+  +---+  +---+  +---+  +--------+
           |   |  |   |  |   |  |   |  |   |  |   |  |        |
           |      |      |      |      |      |      |        |
         +-o-+  +-o-+  +-o-+  +-o-+  +-o-+  +-o-+  +-o-+      |
       +-|   |--|   |--|   |--|   |--|   |--|   |--|   |-+    |
       | | o |  | o |  | o |  | o |  | o |  | o |  | o | |    | Plane A
       +-|   |--|   |--|   |--|   |--|   |--|   |--|   |-+    |
         +-o-+  +-o-+  +-o-+  +-o-+  +-o-+  +-o-+  +-o-+      |
          |      |      |      |      |      |      |         |
         +-o-+  +-o-+  +-o-+  +-o-+  +-o-+  +-o-+  +-o-+      |
       +-|   |--|   |--|   |--|   |--|   |--|   |--|   |-+    |
       | | o |  | o |  | o |  | o |  | o |  | o |  | o | |    | Plane B
       +-|   |--|   |--|   |--|   |--|   |--|   |--|   |-+    |
         +-o-+  +-o-+  +-o-+  +-o-+  +-o-+  +-o-+  +-o-+      |
           |      |      |      |      |      |      |        |
                               ...                            |
           |      |      |      |      |      |      |        |
         +-o-+  +-o-+  +-o-+  +-o-+  +-o-+  +-o-+  +-o-+      |
       +-|   |--|   |--|   |--|   |--|   |--|   |--|   |-+    |
       | | o |  | o |  | o |  | o |  | o |  | o |  | o | |    | Plane X
       +-|   |--|   |--|   |--|   |--|   |--|   |--|   |-+    |
         +-o-+  +-o-+  +-o-+  +-o-+  +-o-+  +-o-+  +-o-+      |
           |      |      |      |      |      |      |        |
           |   |  |   |  |   |  |   |  |   |  |   |  |        |
           +---+  +---+  +---+  +---+  +---+  +---+  +--------+
    Rings    1      2      3      4      5      6      7
```

Figure 13: Connecting Top-of-Fabric Nodes Across Planes by Rings

4.1.5. Addressing the Fallen Leaves Problem

One consequence of the "Fallen Leaf" problem is that some prefixes attached to the fallen leaf become unreachable from some of the ToF nodes. RIFT proposes two methods to address this issue, the positive and the negative disaggregation. Both methods flood South TIEs to advertise the impacted prefix(es).

When used for the operation of disaggregation, a positive South TIE, as usual, indicates reachability to a prefix of given length and all addresses subsumed by it. In contrast, a negative route advertisement indicates that the origin cannot route to the advertised prefix.

The positive disaggregation is originated by a router that can still reach the advertised prefix, and the operation is not transitive. In other words, the receiver does not generate its own flooding south as a consequence of receiving positive disaggregation advertisements from a higher level node. The effect of a positive disaggregation is that the traffic to the impacted prefix will follow the longest match and will be limited to the northbound routers that advertised the more specific route.

In contrast, the negative disaggregation can be transitive, and is propagated south when all the possible routes have been advertised as negative exceptions. A negative route advertisement is only actionable when the negative prefix is aggregated by a positive route advertisement for a shorter prefix. In such case, the negative advertisement "punches out a hole" in the positive route in the routing table, making the positive prefix reachable through the originator with the special consideration of the negative prefix removing certain next hop neighbors.

When the ToF is not partitioned, the collective southern flooding of the positive disaggregation by the ToF nodes that can still reach the impacted prefix is in general enough to cover all the switches at the next level south, typically the ToP nodes. If all those switches are aware of the disaggregation, they collectively create a ceiling that intercepts all the traffic north and forwards it to the ToF nodes that advertised the more specific route. In that case, the positive disaggregation alone is sufficient to solve the fallen leaf problem.

On the other hand, when the fabric is partitioned in planes, the positive disaggregation from ToF nodes in different planes do not reach the ToP switches in the affected plane and cannot solve the fallen leaves problem. In other words, a breakage in a plane can only be solved in that plane. Also, the selection of the plane for a packet typically occurs at the leaf level and the disaggregation must be transitive and reach all the leaves. In that case, the negative disaggregation is necessary. The details on the RIFT approach to deal with fallen leaves in an optimal way are specified in Section 4.2.5.2.

4.2. Specification

This section specifies the protocol in a normative fashion by either prescriptive procedures or behavior defined by Finite State Machines (FSM).

Some FSM figures are provided as [DOT] description due to limitations of ASCII art.

"On Entry" actions on FSM state are performed every time and right before the according state is entered, i.e. after any transitions from previous state.

"On Exit" actions are performed every time and immediately when a state is exited, i.e. before any transitions towards target state are performed.

Any attempt to transition from a state towards another on reception of an event where no action is specified MUST be considered an unrecoverable error.

The FSMs and procedures are normative in the sense that an implementation MUST implement them either literally or an implementation MUST exhibit externally observable behavior that is identical to the execution of the specified FSMs.

Where a FSM representation is inconvenient, i.e. the amount of procedures and kept state exceeds the amount of transitions, we defer to a more procedural description on data structures.

4.2.1. Transport

All packet formats are defined in Thrift [thrift] models in Appendix B.

The serialized model is carried in an envelope within a UDP frame that provides security and allows validation/modification of several important fields without de-serialization for performance and security reasons.

4.2.2. Link (Neighbor) Discovery (LIE Exchange)

RIFT LIE exchange auto-discovers neighbors, negotiates ZTP parameters and discovers miscablings. It uses a three-way handshake mechanism which is a cleaned up version of [RFC5303]. Observe that for easier comprehension the terminology of one/two and three-way states does NOT align with OSPF or ISIS FSMs albeit they use roughly same mechanisms. The formation progresses under normal conditions from one-way to two-way and then three-way state at which point it is ready to exchange TIEs per Section 4.2.3.

LIE exchange happens over well-known administratively locally scoped and configured or otherwise well-known IPv4 multicast address [RFC2365] and/or link-local multicast scope [RFC4291] for IPv6 [RFC8200] using a configured or otherwise a well-known destination UDP port defined in Appendix C.1. LIEs SHOULD be sent with an IPv4 Time to Live (TTL) / IPv6 Hop Limit (HL) of 1 to prevent RIFT information reaching beyond a single L3 next-hop in the topology. LIEs SHOULD be sent with network control precedence.

Originating port of the LIE has no further significance other than identifying the origination point. LIEs are exchanged over all links running RIFT.

An implementation MAY listen and send LIEs on IPv4 and/or IPv6 multicast addresses. A node MUST NOT originate LIEs on an address family if it does not process received LIEs on that family. LIEs on same link are considered part of the same negotiation independent of the address family they arrive on. Observe further that the LIE source address may not identify the peer uniquely in unnumbered or link-local address cases so the response transmission MUST occur over the same interface the LIEs have been received on. A node MAY use any of the adjacency's source addresses it saw in LIEs on the specific interface during adjacency formation to send TIEs. That implies that an implementation MUST be ready to accept TIEs on all addresses it used as source of LIE frames.

A three-way adjacency over any address family implies support for IPv4 forwarding if the 'v4_forwarding_capable' flag is set to true and a node can use [RFC5549] type of forwarding in such a situation. It is expected that the whole fabric supports the same type of forwarding of address families on all the links. Operation of a fabric where only some of the links are supporting forwarding on an address family and others do not is outside the scope of this specification.

The protocol does NOT support selective disabling of address families, disabling v4 forwarding capability or any local address changes in three-way state, i.e. if a link has entered three-way IPv4 and/or IPv6 with a neighbor on an adjacency and it wants to stop supporting one of the families or change any of its local addresses or stop v4 forwarding, it has to tear down and rebuild the adjacency. It also has to remove any information it stored about the adjacency such as LIE source addresses seen.

Unless ZTP as described in Section 4.2.7 is used, each node is provisioned with the level at which it is operating. It MAY be also provisioned with its PoD. If any of those values is undefined, then accordingly a default level and/or an "undefined" PoD are assumed. This means that leaves do not need to be configured at all if initial configuration values are all left at "undefined" value. Nodes above ToP MUST remain at "any" PoD value which has the same value as "undefined" PoD. This information is propagated in the LIEs exchanged.

Further definitions of leaf flags are found in Section 4.2.7 given they have implications in terms of level and adjacency forming here.

A node tries to form a three-way adjacency if and only if

1. the node is in the same PoD or either the node or the neighbor advertises "undefined/any" PoD membership (PoD# = 0) AND

2. the neighboring node is running the same MAJOR schema version AND

3. the neighbor is not member of some PoD while the node has a northbound adjacency already joining another PoD AND

4. the neighboring node uses a valid System ID AND

5. the neighboring node uses a different System ID than the node itself

6. the advertised MTUs match on both sides AND

7. both nodes advertise defined level values AND

8. [

   i) the node is at level 0 and has no three way adjacencies already to nodes at Highest Adjacency Three-Way level (HAT as defined later in Section 4.2.7.1) with level different than the adjacent node OR

   ii) the node is not at level 0 and the neighboring node is at level 0 OR

   iii) both nodes are at level 0 AND both indicate support for Section 4.3.8 OR

   iv) neither node is at level 0 and the neighboring node is at most one level away

   ].

The rules checking PoD numbering MAY be optionally disregarded by a node if PoD detection is undesirable or has to be ignored. This will not affect the correctness of the protocol except preventing detection of certain miscabling cases.

A node configured with "undefined" PoD membership MUST, after building first northbound three way adjacencies to a node being in a defined PoD, advertise that PoD as part of its LIEs. In case that adjacency is lost, from all available northbound three way adjacencies the node with the highest System ID and defined PoD is chosen. That way the northmost defined PoD value (normally the ToP nodes) can diffuse southbound towards the leaves "forcing" the PoD value on any node with "undefined" PoD.

LIEs arriving with IPv4 Time to Live (TTL) / IPv6 Hop Limit (HL) larger than 1 MUST be ignored.

A node SHOULD NOT send out LIEs without defined level in the header but in certain scenarios it may be beneficial for trouble-shooting purposes.

4.2.2.1. LIE FSM

This section specifies the precise, normative LIE FSM and can be omitted unless the reader is pursuing an implementation of the protocol.

Initial state is 'OneWay'.

Event 'MultipleNeighbors' occurs normally when more than two nodes see each other on the same link or a remote node is quickly reconfigured or rebooted without regressing to 'OneWay' first. Each occurrence of the event SHOULD generate a clear, according notification to help operational deployments.

The machine sends LIEs on several transitions to accelerate adjacency bring-up without waiting for the timer tic.

```
               Enter
                 |
                 V
           +-----------+
           | OneWay    |<----+
           |           |     | HALChanged [StoreHAL]
           | Entry:    |     | HALSChanged [StoreHALS]
           | [CleanUp] |     | HATChanged [StoreHAT]
           |           |     | HoldTimerExpired [-]
           |           |     | InstanceNameMismatch [-]
           |           |     | LevelChanged [UpdateLevel, PUSH SendLie]
           |           |     | LieReceived [ProcessLIE]
           |           |     | MTUMismatch [-]
           |           |     | NeighborAddressAdded [-]
           |           |     | NeighborChangedAddress [-]
           |           |     | NeighborChangedLevel [-]
           |           |     | NeighborChangedMinorFields [-]
           |           |     | NeighborDroppedReflection [-]
           |           |     | PODMismatch [-]
           |           |     | SendLIE [SendLIE]
           |           |     | TimerTick [PUSH SendLIE]
           |           |     | UnacceptableHeader
           |           |     | UpdateZTPOffer [SendOfferToZTPFSM]
           |           |-----+
           |           |
           |           |<--------------------- (ThreeWay)
           |           |--------------------->
           |           | ValidReflection [-]
           |           |
           |           |---------------------> (Multiple
           |           | MultipleNeighbors      Neighbors
           +-----------+  [StartMulNeighTimer]  Wait)
               ^    |
               |    |
               |    | NewNeighbor [PUSH SendLIE]
               |    V
             (TwoWay)
```

LIE FSM

```
              (OneWay)
               |    ^
               |    | HoldTimeExpired [-]
               |    | InstanceNameMismatch [-]
               |    | LevelChanged [StoreLevel]
               |    | MTUMismatch [-]
               |    | NeighborChangedAddress [-]
               |    | NeighborChangedLevel [-]
               |    | PODMismatch [-]
               |    | UnacceptableHeader [-]
               V    |
           +-----------+
           | TwoWay    |<----+
           |           |     | HALChanged [StoreHAL]
           |           |     | HALSChanged [StoreHALS]
           |           |     | HATChanged [StoreHAT]
           |           |     | LevelChanged [StoreLevel]
           |           |     | LIERcvd [ProcessLIE]
           |           |     | SendLIE [SendLIE]
           |           |     | TimerTick [PUSH SendLIE,
           |           |     |            IF HoldTimer expired
           |           |     |            PUSH HoldTimerExpired]
           |           |     | UpdateZTPOffer [SendOfferToZTPFSM]
           |           |-----+
           |           |
           |           |<----------------------
           |           |----------------------> (Multiple
           |           | NewNeighbor             Neighbors
           |           |   [StartMulNeighTimer]  Wait)
           |           | MultipleNeighbors
           +-----------+  [StartMulNeighTimer]
               ^    |
               |    | ValidReflection [-]
               |    V
             (ThreeWay)

                            LIE FSM (continued)
```

```
                     (TwoWay)    (OneWay)
                      ^    |        ^
                      |    |        | HoldTimerExpired [-]
                      |    |        | InstanceNameMismatch [-]
                      |    |        | LevelChanged [UpdateLevel]
                      |    |        | MTUMismatch [-]
                      |    |        | NeighborChangedAddress [-]
                      |    |        | NeighborChangedLevel [-]
  NeighborDropped-    |    |        | PODMismatch [-]
       Reflection [-] |    |        | UnacceptableHeader [-]
                      |    V        |
                  +-----------+     |
                  | ThreeWay  |-----+
                  |           |
                  |           |<----+
                  |           |     | HALChanged [StoreHAL]
                  |           |     | HALSChanged [StoreHALS]
                  |           |     | HATChanged [StoreHAT]
                  |           |     | LieReceived [ProcessLIE]
                  |           |     | SendLIE [SendLIE]
                  |           |     | TimerTick [PUSH SendLie,
                  |           |     |            IF HoldTimer expired
                  |           |     |            PUSH HoldTimerExpired]
                  |           |     | UpdateZTPOffer [SendOfferToZTPFSM]
                  |           |     | ValidReflection [-]
                  |           |-----+
                  |           |----------------------> (Multiple
                  |           | MultipleNeighbors       Neighbors
                  +-----------+   [StartMulNeighTimer]  Wait)
```

LIE FSM (continued)

```
          (TwoWay) (ThreeWay)
               |      |
               V      V
           +------------+
           | Multiple   |<----+
           | Neighbors  |     | HALChanged [StoreHAL]
           | Wait       |     | HALSChanged [StoreHALS]
           |            |     | HATChanged [StoreHAT]
           |            |     | MultipleNeighbors
           |            |     |   [StartMultipleNeighborsTimer]
           |            |     | TimerTick [IF MulNeighTimer expired
           |            |     |              PUSH MultipleNeighborsDone]
           |            |     | UpdateZTPOffer [SendOfferToZTP]
           |            |-----+
           |            |
           |            |<---------------------------
           |            |---------------------------> (OneWay)
           |            | LevelChanged [StoreLevel]
           +------------+ MultipleNeighborsDone [-]
```

LIE FSM (continued)

Events

- TimerTick: one second timer tic
- LevelChanged: node’s level has been changed by ZTP or configuration
- HALChanged: best HAL computed by ZTP has changed
- HATChanged: HAT computed by ZTP has changed
- HALSChanged: set of HAL offering systems computed by ZTP has changed
- LieRcvd: received LIE
- NewNeighbor: new neighbor parsed
- ValidReflection: received own reflection from neighbor
- NeighborDroppedReflection: lost previous own reflection from neighbor
- NeighborChangedLevel: neighbor changed advertised level
- NeighborChangedAddress: neighbor changed IP address

- o UnacceptableHeader: unacceptable header seen
- o MTUMismatch: MTU mismatched
- o InstanceNameMismatch: Instance mismatched
- o PODMismatch: Unacceptable PoD seen
- o HoldtimeExpired: adjacency hold down expired
- o MultipleNeighbors: more than one neighbor seen on interface
- o MultipleNeighborsDone: cooldown for multiple neighbors expired
- o SendLie: send a LIE out
- o UpdateZTPOffer: update this node's ZTP offer

Actions

on MultipleNeighbors in OneWay finishes in MultipleNeighborsWait: start multiple neighbors timer as 4 * DEFAULT_LIE_HOLDTIME

on NeighborDroppedReflection in ThreeWay finishes in TwoWay: no action

on NeighborDroppedReflection in OneWay finishes in OneWay: no action

on PODMismatch in TwoWay finishes in OneWay: no action

on NewNeighbor in TwoWay finishes in MultipleNeighborsWait: PUSH SendLie event

on LieRcvd in OneWay finishes in OneWay: PROCESS_LIE

on UnacceptableHeader in ThreeWay finishes in OneWay: no action

on UpdateZTPOffer in TwoWay finishes in TwoWay: send offer to ZTP FSM

on NeighborChangedAddress in ThreeWay finishes in OneWay: no action

on HALChanged in MultipleNeighborsWait finishes in MultipleNeighborsWait: store new HAL

on NeighborChangedAddress in TwoWay finishes in OneWay: no action on MultipleNeighbors in TwoWay finishes in MultipleNeighborsWait: start multiple neighbors timer as 4 * DEFAULT_LIE_HOLDTIME

on LevelChanged in ThreeWay finishes in OneWay: update level with event value

on LieRcvd in ThreeWay finishes in ThreeWay: PROCESS_LIE

on ValidReflection in OneWay finishes in ThreeWay: no action

on NeighborChangedLevel in TwoWay finishes in OneWay: no action

on MultipleNeighbors in ThreeWay finishes in MultipleNeighborsWait: start multiple neighbors timer as 4 * DEFAULT_LIE_HOLDTIME

on InstanceNameMismatch in OneWay finishes in OneWay: no action

on NewNeighbor in OneWay finishes in TwoWay: PUSH SendLie event

on UpdateZTPOffer in OneWay finishes in OneWay: send offer to ZTP FSM

on UpdateZTPOffer in ThreeWay finishes in ThreeWay: send offer to ZTP FSM

on MTUMismatch in ThreeWay finishes in OneWay: no action

on TimerTick in OneWay finishes in OneWay: PUSH SendLie event

on SendLie in TwoWay finishes in TwoWay: SEND_LIE

on ValidReflection in ThreeWay finishes in ThreeWay: no action

on InstanceNameMismatch in TwoWay finishes in OneWay: no action

on HoldtimeExpired in OneWay finishes in OneWay: no action

on TimerTick in ThreeWay finishes in ThreeWay: PUSH SendLie event, if holdtime expired PUSH HoldtimeExpired event

on HALChanged in TwoWay finishes in TwoWay: store new HAL

on HoldtimeExpired in ThreeWay finishes in OneWay: no action

on HALSChanged in TwoWay finishes in TwoWay: store HALS

on HALSChanged in ThreeWay finishes in ThreeWay: store HALS on ValidReflection in TwoWay finishes in ThreeWay: no action

on MultipleNeighborsDone in MultipleNeighborsWait finishes in OneWay: no action

on NeighborAddressAdded in OneWay finishes in OneWay: no action

on TimerTick in MultipleNeighborsWait finishes in MultipleNeighborsWait: decrement MultipleNeighbors timer, if expired PUSH MultipleNeighborsDone

on MTUMismatch in OneWay finishes in OneWay: no action

on MultipleNeighbors in MultipleNeighborsWait finishes in MultipleNeighborsWait: start multiple neighbors timer as 4 * DEFAULT_LIE_HOLDTIME

on LieRcvd in TwoWay finishes in TwoWay: PROCESS_LIE

on HATChanged in MultipleNeighborsWait finishes in MultipleNeighborsWait: store HAT

on HoldtimeExpired in TwoWay finishes in OneWay: no action

on NeighborChangedLevel in ThreeWay finishes in OneWay: no action

on LevelChanged in OneWay finishes in OneWay: update level with event value, PUSH SendLie event

on SendLie in OneWay finishes in OneWay: SEND_LIE

on HATChanged in OneWay finishes in OneWay: store HAT

on LevelChanged in TwoWay finishes in TwoWay: update level with event value

on HATChanged in TwoWay finishes in TwoWay: store HAT

on PODMismatch in ThreeWay finishes in OneWay: no action

on LevelChanged in MultipleNeighborsWait finishes in OneWay: update level with event value

on UnacceptableHeader in TwoWay finishes in OneWay: no action

on NeighborChangedLevel in OneWay finishes in OneWay: no action

on InstanceNameMismatch in ThreeWay finishes in OneWay: no action on HATChanged in ThreeWay finishes in ThreeWay: store HAT

on HALChanged in OneWay finishes in OneWay: store new HAL

on UnacceptableHeader in OneWay finishes in OneWay: no action

on HALChanged in ThreeWay finishes in ThreeWay: store new HAL

on UpdateZTPOffer in MultipleNeighborsWait finishes in MultipleNeighborsWait: send offer to ZTP FSM

on NeighborChangedMinorFields in OneWay finishes in OneWay: no action

on NeighborChangedAddress in OneWay finishes in OneWay: no action

on MTUMismatch in TwoWay finishes in OneWay: no action

on PODMismatch in OneWay finishes in OneWay: no action

on SendLie in ThreeWay finishes in ThreeWay: SEND_LIE

on TimerTick in TwoWay finishes in TwoWay: PUSH SendLie event, if holdtime expired PUSH HoldtimeExpired event

on HALSChanged in OneWay finishes in OneWay: store HALS

on HALSChanged in MultipleNeighborsWait finishes in MultipleNeighborsWait: store HALS

on Entry into OneWay: CLEANUP

Following words are used for well known procedures:

1. PUSH Event: pushes an event to be executed by the FSM upon exit of this action

2. CLEANUP: neighbor MUST be reset to unknown

3. SEND_LIE: create a new LIE packet

    1. reflecting the neighbor if known and valid and

    2. setting the necessary 'not_a_ztp_offer' variable if level was derived from last known neighbor on this interface and

    3. setting 'you_are_flood_repeater' to computed value

4. PROCESS_LIE:

    1. if lie has wrong major version OR our own system ID or invalid system ID then CLEANUP else

    2. if lie has non matching MTUs then CLEANUP, PUSH UpdateZTPOffer, PUSH MTUMismatch else

    3. if PoD rules do not allow adjacency forming then CLEANUP, PUSH PODMismatch, PUSH MTUMismatch else

    4. if lie has undefined level OR my level is undefined OR this node is leaf and remote level lower than HAT OR (lie's level is not leaf AND its difference is more than one from my level) then CLEANUP, PUSH UpdateZTPOffer, PUSH UnacceptableHeader else

    5. PUSH UpdateZTPOffer, construct temporary new neighbor structure with values from lie, if no current neighbor exists then set neighbor to new neighbor, PUSH NewNeighbor event, CHECK_THREE_WAY else

        1. if current neighbor system ID differs from lie's system ID then PUSH MultipleNeighbors else

        2. if current neighbor stored level differs from lie's level then PUSH NeighborChangedLevel else

        3. if current neighbor stored IPv4/v6 address differs from lie's address then PUSH NeighborChangedAddress else

        4. if any of neighbor's flood address port, name, local linkid changed then PUSH NeighborChangedMinorFields and

        5. CHECK_THREE_WAY

5. CHECK_THREE_WAY: if current state is one-way do nothing else

    1. if lie packet does not contain neighbor then if current state is three-way then PUSH NeighborDroppedReflection else

    2. if packet reflects this system's ID and local port and state is three-way then PUSH event ValidReflection else PUSH event MultipleNeighbors

4.2.3. Topology Exchange (TIE Exchange)

4.2.3.1. Topology Information Elements

Topology and reachability information in RIFT is conveyed by the means of TIEs which have good amount of commonalities with LSAs in OSPF.

The TIE exchange mechanism uses the port indicated by each node in the LIE exchange and the interface on which the adjacency has been formed as destination. It SHOULD use TTL of 1 as well and set internetwork control precedence on according packets.

TIEs contain sequence numbers, lifetimes and a type. Each type has ample identifying number space and information is spread across possibly many TIEs of a certain type by the means of a hash function that a node or deployment can individually determine. One extreme design choice is a prefix per TIE which leads to more BGP-like behavior where small increments are only advertised on route changes vs. deploying with dense prefix packing into few TIEs leading to more traditional IGP trade-off with fewer TIEs. An implementation may even rehash prefix to TIE mapping at any time at the cost of significant amount of re-advertisements of TIEs.

More information about the TIE structure can be found in the schema in Appendix B.

4.2.3.2. South- and Northbound Representation

A central concept of RIFT is that each node represents itself differently depending on the direction in which it is advertising information. More precisely, a spine node represents two different databases over its adjacencies depending whether it advertises TIEs to the north or to the south/sideways. We call those differing TIE databases either south- or northbound (South TIEs and North TIEs) depending on the direction of distribution.

The North TIEs hold all of the node's adjacencies and local prefixes while the South TIEs hold only all of the node's adjacencies, the default prefix with necessary disaggregated prefixes and local prefixes. We will explain this in detail further in Section 4.2.5.

The TIE types are mostly symmetric in both directions and Table 2 provides a quick reference to main TIE types including direction and their function.

TIE-Type	Content
Node North TIE	node properties and adjacencies
Node South TIE	same content as node North TIE
Prefix North TIE	contains nodes' directly reachable prefixes
Prefix South TIE	contains originated defaults and directly reachable prefixes
Positive Disaggregation South TIE	contains disaggregated prefixes
Negative Disaggregation South TIE	contains special, negatively disaggregated prefixes to support multi-plane designs
External Prefix North TIE	contains external prefixes
Key-Value North TIE	contains nodes northbound KVs
Key-Value South TIE	contains nodes southbound KVs

Table 2: TIE Types

As an example illustrating a databases holding both representations, consider the topology in Figure 2 with the optional link between spine 111 and spine 112 (so that the flooding on an East-West link can be shown). This example assumes unnumbered interfaces. First, here are the TIEs generated by some nodes. For simplicity, the key value elements which may be included in their South TIEs or North TIEs are not shown.

```
ToF 21 South TIEs:
Node South TIE:
  NodeElement(level=2, neighbors((Spine 111, level 1, cost 1),
  (Spine 112, level 1, cost 1), (Spine 121, level 1, cost 1),
  (Spine 122, level 1, cost 1)))
Prefix South TIE:
  SouthPrefixesElement(prefixes(0/0, cost 1), (::/0, cost 1))
```

```
Spine 111 South TIEs:
Node South TIE:
  NodeElement(level=1, neighbors((ToF 21, level 2, cost 1,
              links(...)),
   (ToF 22, level 2, cost 1, links(...)),
   (Spine 112, level 1, cost 1, links(...)),
   (Leaf111, level 0, cost 1, links(...)),
   (Leaf112, level 0, cost 1, links(...))))
Prefix South TIE:
  SouthPrefixesElement(prefixes(0/0, cost 1), (::/0, cost 1))

Spine 111 North TIEs:
Node North TIE:
  NodeElement(level=1,
  neighbors((ToF 21, level 2, cost 1, links(...)),
   (ToF 22, level 2, cost 1, links(...)),
   (Spine 112, level 1, cost 1, links(...)),
   (Leaf111, level 0, cost 1, links(...)),
   (Leaf112, level 0, cost 1, links(...))))
Prefix North TIE:
  NorthPrefixesElement(prefixes(Spine 111.loopback)

Spine 121 South TIEs:
Node South TIE:
  NodeElement(level=1, neighbors((ToF 21,level 2,cost 1),
   (ToF 22, level 2, cost 1), (Leaf121, level 0, cost 1),
   (Leaf122, level 0, cost 1)))
Prefix South TIE:
  SouthPrefixesElement(prefixes(0/0, cost 1), (::/0, cost 1))

Spine 121 North TIEs:
Node North TIE:
  NodeElement(level=1,
  neighbors((ToF 21, level 2, cost 1, links(...)),
   (ToF 22, level 2, cost 1, links(...)),
   (Leaf121, level 0, cost 1, links(...)),
   (Leaf122, level 0, cost 1, links(...))))
Prefix North TIE:
  NorthPrefixesElement(prefixes(Spine 121.loopback)

Leaf112 North TIEs:
Node North TIE:
  NodeElement(level=0,
  neighbors((Spine 111, level 1, cost 1, links(...)),
   (Spine 112, level 1, cost 1, links(...))))
Prefix North TIE:
  NorthPrefixesElement(prefixes(Leaf112.loopback, Prefix112,
  Prefix_MH))
```

Figure 14: Example TIES Generated in a 2 Level Spine-and-Leaf Topology

It may be here not necessarily obvious why the node South TIEs contain all the adjacencies of the according node. This will be necessary for algorithms given in Section 4.2.3.9 and Section 4.3.6.

4.2.3.3. Flooding

The mechanism used to distribute TIEs is the well-known (albeit modified in several respects to take advantage of fat tree topology) flooding mechanism used by today's link-state protocols. Although flooding is initially more demanding to implement it avoids many problems with update style used in diffused computation such as distance vector protocols. Since flooding tends to present an unscalable burden in large, densely meshed topologies (fat trees being unfortunately such a topology) we provide as solution close to optimal global flood reduction and load balancing optimization in Section 4.2.3.9.

As described before, TIEs themselves are transported over UDP with the ports indicated in the LIE exchanges and using the destination address on which the LIE adjacency has been formed. For unnumbered IPv4 interfaces same considerations apply as in equivalent OSPF case.

4.2.3.3.1. Normative Flooding Procedures

On reception of a TIE with an undefined level value in the packet header the node SHOULD issue a warning and indiscriminately discard the packet.

This section specifies the precise, normative flooding mechanism and can be omitted unless the reader is pursuing an implementation of the protocol.

Flooding Procedures are described in terms of a flooding state of an adjacency and resulting operations on it driven by packet arrivals. The FSM itself has basically just a single state and is not well suited to represent the behavior. An implementation MUST behave on the wire in the same way as the provided normative procedures of this paragraph.

RIFT does not specify any kind of flood rate limiting since such specifications always assume particular points in available technology speeds and feeds and those points are shifting at faster and faster rate (speed of light holding for the moment). The encoded packets provide hints to react accordingly to losses or overruns.

Flooding of all according topology exchange elements SHOULD be performed at highest feasible rate whereas the rate of transmission MUST be throttled by reacting to adequate features of the system such as e.g. queue lengths or congestion indications in the protocol packets.

A node SHOULD NOT send out any topology information elements if the adjacency is not in a "three-way" state. No further tightening of this rule is possible due to possible link buffering and re-ordering of LIEs and TIEs/TIDEs/TIREs.

A node MUST drop any received TIEs/TIDEs/TIREs unless it is in three-way state.

TIDEs and TIREs MUST NOT be re-flooded the way TIEs of other nodes MUST be always generated by the node itself and cross only to the neighboring node.

4.2.3.3.1.1. FloodState Structure per Adjacency

The structure contains conceptually the following elements. The word collection or queue indicates a set of elements that can be iterated:

TIES_TX: Collection containing all the TIEs to transmit on the adjacency.

TIES_ACK: Collection containing all the TIEs that have to be acknowledged on the adjacency.

TIES_REQ: Collection containing all the TIE headers that have to be requested on the adjacency.

TIES_RTX: Collection containing all TIEs that need retransmission with the according time to retransmit.

Following words are used for well known procedures operating on this structure:

TIE Describes either a full RIFT TIE or accordingly just the 'TIEHeader' or 'TIEID'. The according meaning is unambiguously contained in the context of the algorithm.

is_flood_reduced(TIE): returns whether a TIE can be flood reduced or not.

is_tide_entry_filtered(TIE): returns whether a header should be propagated in TIDE according to flooding scopes.

is_request_filtered(TIE): returns whether a TIE request should be propagated to neighbor or not according to flooding scopes.

is_flood_filtered(TIE): returns whether a TIE requested be flooded to neighbor or not according to flooding scopes.

try_to_transmit_tie(TIE):

A. if not is_flood_filtered(TIE) then

   1. remove TIE from TIES_RTX if present

   2. if TIE" with same key on TIES_ACK then

      a. if TIE" same or newer than TIE do nothing else

      b. remove TIE" from TIES_ACK and add TIE to TIES_TX

   3. else insert TIE into TIES_TX

ack_tie(TIE): remove TIE from all collections and then insert TIE into TIES_ACK.

tie_been_acked(TIE): remove TIE from all collections.

remove_from_all_queues(TIE): same as 'tie_been_acked'.

request_tie(TIE): if not is_request_filtered(TIE) then remove_from_all_queues(TIE) and add to TIES_REQ.

move_to_rtx_list(TIE): remove TIE from TIES_TX and then add to TIES_RTX using TIE retransmission interval.

clear_requests(TIEs): remove all TIEs from TIES_REQ.

bump_own_tie(TIE): for self-originated TIE originate an empty or re-generate with version number higher then the one in TIE.

The collection SHOULD be served with following priorities if the system cannot process all the collections in real time:

Elements on TIES_ACK should be processed with highest priority

TIES_TX

TIES_REQ and TIES_RTX

4.2.3.3.1.2. TIDEs

`TIEID` and `TIEHeader` space forms a strict total order (modulo incomparable sequence numbers in the very unlikely event that can occur if a TIE is "stuck" in a part of a network while the originator reboots and reissues TIEs many times to the point its sequence# rolls over and forms incomparable distance to the "stuck" copy) which implies that a comparison relation is possible between two elements. With that it is implicitly possible to compare TIEs, TIEHeaders and TIEIDs to each other whereas the shortest viable key is always implied.

When generating and sending TIDEs an implementation SHOULD ensure that enough bandwidth is left to send elements of Floodstate structure.

4.2.3.3.1.2.1. TIDE Generation

As given by timer constant, periodically generate TIDEs by:

NEXT_TIDE_ID: ID of next TIE to be sent in TIDE.

TIDE_START: Begin of TIDE packet range.

a. NEXT_TIDE_ID = MIN_TIEID

b. while NEXT_TIDE_ID not equal to MAX_TIEID do

    1. TIDE_START = NEXT_TIDE_ID

    2. HEADERS = At most TIRDEs_PER_PKT headers in TIEDB starting at NEXT_TIDE_ID or higher that SHOULD be filtered by is_tide_entry_filtered and MUST either have a lifetime left > 0 or have no content

    3. if HEADERS is empty then START = MIN_TIEID else START = first element in HEADERS

    4. if HEADERS' size less than TIRDEs_PER_PKT then END = MAX_TIEID else END = last element in HEADERS

    5. send sorted HEADERS as TIDE setting START and END as its range

    6. NEXT_TIDE_ID = END

The constant 'TIRDEs_PER_PKT' SHOULD be generated and used by the implementation to limit the amount of TIE headers per TIDE so the sent TIDE PDU does not exceed interface MTU.

TIDE PDUs SHOULD be spaced on sending to prevent packet drops.

4.2.3.3.1.2.2. TIDE Processing

On reception of TIDEs the following processing is performed:

TXKEYS: Collection of TIE Headers to be send after processing of the packet

REQKEYS: Collection of TIEIDs to be requested after processing of the packet

CLEARKEYS: Collection of TIEIDs to be removed from flood state queues

LASTPROCESSED: Last processed TIEID in TIDE

DBTIE: TIE in the LSDB if found

a. LASTPROCESSED = TIDE.start_range

b. for every HEADER in TIDE do

    1. DBTIE = find HEADER in current LSDB

    2. if HEADER < LASTPROCESSED then report error and reset adjacency and return

    3. put all TIEs in LSDB where (TIE.HEADER > LASTPROCESSED and TIE.HEADER < HEADER) into TXKEYS

    4. LASTPROCESSED = HEADER

    5. if DBTIE not found then

        I) if originator is this node then bump_own_tie

        II) else put HEADER into REQKEYS

    6. if DBTIE.HEADER < HEADER then

        I) if originator is this node then bump_own_tie else

i. if this is a North TIE header from a northbound neighbor then override DBTIE in LSDB with HEADER

ii. else put HEADER into REQKEYS

7. if DBTIE.HEADER > HEADER then put DBTIE.HEADER into TXKEYS

8. if DBTIE.HEADER = HEADER then

I) if DBTIE has content already then put DBTIE.HEADER into CLEARKEYS

II) else put HEADER into REQKEYS

c. put all TIEs in LSDB where (TIE.HEADER > LASTPROCESSED and TIE.HEADER <= TIDE.end_range) into TXKEYS

d. for all TIEs in TXKEYS try_to_transmit_tie(TIE)

e. for all TIEs in REQKEYS request_tie(TIE)

f. for all TIEs in CLEARKEYS remove_from_all_queues(TIE)

4.2.3.3.1.3. TIREs

4.2.3.3.1.3.1. TIRE Generation

Elements from both TIES_REQ and TIES_ACK MUST be collected and sent out as fast as feasible as TIREs. When sending TIREs with elements from TIES_REQ the 'lifetime' field MUST be set to 0 to force reflooding from the neighbor even if the TIEs seem to be same.

4.2.3.3.1.3.2. TIRE Processing

On reception of TIREs the following processing is performed:

TXKEYS: Collection of TIE Headers to be send after processing of the packet

REQKEYS: Collection of TIEIDs to be requested after processing of the packet

ACKKEYS: Collection of TIEIDs that have been acked

DBTIE: TIE in the LSDB if found

a. for every HEADER in TIRE do

1. DBTIE = find HEADER in current LSDB

2. if DBTIE not found then do nothing

3. if DBTIE.HEADER < HEADER then put HEADER into REQKEYS

4. if DBTIE.HEADER > HEADER then put DBTIE.HEADER into TXKEYS

5. if DBTIE.HEADER = HEADER then put DBTIE.HEADER into ACKKEYS

b. for all TIEs in TXKEYS try_to_transmit_tie(TIE)

c. for all TIEs in REQKEYS request_tie(TIE)

d. for all TIEs in ACKKEYS tie_been_acked(TIE)

4.2.3.3.1.4. TIEs Processing on Flood State Adjacency

On reception of TIEs the following processing is performed:

ACKTIE: TIE to acknowledge

TXTIE: TIE to transmit

DBTIE: TIE in the LSDB if found

a. DBTIE = find TIE in current LSDB

b. if DBTIE not found then

1. if originator is this node then bump_own_tie with a short remaining lifetime

2. else insert TIE into LSDB and ACKTIE = TIE

else

1. if DBTIE.HEADER = TIE.HEADER then

i. if DBTIE has content already then ACKTIE = TIE

ii. else process like the "DBTIE.HEADER < TIE.HEADER" case

2. if DBTIE.HEADER < TIE.HEADER then

i. if originator is this node then bump_own_tie

ii. else insert TIE into LSDB and ACKTIE = TIE

3. if DBTIE.HEADER > TIE.HEADER then

   i. if DBTIE has content already then TXTIE = DBTIE

   ii. else ACKTIE = DBTIE

c. if TXTIE is set then try_to_transmit_tie(TXTIE)

d. if ACKTIE is set then ack_tie(TIE)

4.2.3.3.1.5. TIEs Processing When LSDB Received Newer Version on Other Adjacencies

The Link State Database can be considered to be a switchboard that does not need any flooding procedures but can be given new versions of TIEs by a peer. Consecutively, a peer receives from the LSDB newer versions of TIEs received by other peers and processes them (without any filtering) just like receiving TIEs from its remote peer. This publisher model can be implemented in many ways.

4.2.3.3.1.6. Sending TIEs

On a periodic basis all TIEs with lifetime left > 0 MUST be sent out on the adjacency, removed from TIES_TX list and requeued onto TIES_RTX list.

4.2.3.4. TIE Flooding Scopes

In a somewhat analogous fashion to link-local, area and domain flooding scopes, RIFT defines several complex "flooding scopes" depending on the direction and type of TIE propagated.

Every North TIE is flooded northbound, providing a node at a given level with the complete topology of the Clos or Fat Tree network that is reachable southwards of it, including all specific prefixes. This means that a packet received from a node at the same or lower level whose destination is covered by one of those specific prefixes will be routed directly towards the node advertising that prefix rather than sending the packet to a node at a higher level.

A node's Node South TIEs, consisting of all node's adjacencies and prefix South TIEs limited to those related to default IP prefix and disaggregated prefixes, are flooded southbound in order to allow the nodes one level down to see connectivity of the higher level as well as reachability to the rest of the fabric. In order to allow an E-W disconnected node in a given level to receive the South TIEs of other nodes at its level, every *NODE* South TIE is "reflected" northbound to level from which it was received. It should be noted that East-

West links are included in South TIE flooding (except at ToF level); those TIEs need to be flooded to satisfy algorithms in Section 4.2.4. In that way nodes at same level can learn about each other without a lower level, e.g. in case of leaf level. The precise, normative flooding scopes are given in Table 3. Those rules govern as well what SHOULD be included in TIDEs on the adjacency. Again, East-West flooding scopes are identical to South flooding scopes except in case of ToF East-West links (rings) which are basically performing northbound flooding.

Node South TIE "south reflection" allows to support positive disaggregation on failures describes in Section 4.2.5 and flooding reduction in Section 4.2.3.9.

Type / Direction	South	North	East-West
node South TIE	flood if level of originator is equal to this node	flood if level of originator is higher than this node	flood only if this node is not ToF
non-node South TIE	flood self-originated only	flood only if neighbor is originator of TIE	flood only if self-originated and this node is not ToF
all North TIEs	never flood	flood always	flood only if this node is ToF
TIDE	include at least all non-self originated North TIE headers and self-originated South TIE headers and node South TIEs of nodes at same level	include at least all node South TIEs and all South TIEs originated by peer and all North TIEs	if this node is ToF then include all North TIEs, otherwise only self-originated TIEs
TIRE as Request	request all North TIEs and all peer's self-originated TIEs and all node South TIEs	request all South TIEs	if this node is ToF then apply North scope rules, otherwise South scope rules
TIRE as Ack	Ack all received TIEs	Ack all received TIEs	Ack all received TIEs

Table 3: Normative Flooding Scopes

If the TIDE includes additional TIE headers beside the ones specified, the receiving neighbor must apply according filter to the received TIDE strictly and MUST NOT request the extra TIE headers that were not allowed by the flooding scope rules in its direction.

As an example to illustrate these rules, consider using the topology in Figure 2, with the optional link between spine 111 and spine 112, and the associated TIEs given in Figure 14. The flooding from particular nodes of the TIEs is given in Table 4.

Router floods to	Neighbor	TIEs
Leaf111	Spine 112	Leaf111 North TIEs, Spine 111 node South TIE
Leaf111	Spine 111	Leaf111 North TIEs, Spine 112 node South TIE
Spine 111	Leaf111	Spine 111 South TIEs
Spine 111	Leaf112	Spine 111 South TIEs
Spine 111	Spine 112	Spine 111 South TIEs
Spine 111	ToF 21	Spine 111 North TIEs, Leaf111 North TIEs, Leaf112 North TIEs, ToF 22 node South TIE
Spine 111	ToF 22	Spine 111 North TIEs, Leaf111 North TIEs, Leaf112 North TIEs, ToF 21 node South TIE
...	...	...
ToF 21	Spine 111	ToF 21 South TIEs
ToF 21	Spine 112	ToF 21 South TIEs
ToF 21	Spine 121	ToF 21 South TIEs
ToF 21	Spine 122	ToF 21 South TIEs
...	...	...

Table 4: Flooding some TIEs from example topology

4.2.3.5. 'Flood Only Node TIEs' Bit

RIFT includes an optional ECN (Explicit Congestion Notification) mechanism to prevent "flooding inrush" on restart or bring-up with many southbound neighbors. A node MAY set on its LIEs the according bit to indicate to the neighbor that it should temporarily flood node TIEs only to it. It SHOULD only set it in the southbound direction. The receiving node SHOULD accommodate the request to lessen the flooding load on the affected node if south of the sender and SHOULD ignore the bit if northbound.

Obviously this mechanism is most useful in southbound direction. The distribution of node TIEs guarantees correct behavior of algorithms like disaggregation or default route origination. Furthermore though, the use of this bit presents an inherent trade-off between processing load and convergence speed since suppressing flooding of northbound prefixes from neighbors will lead to blackholes.

4.2.3.6. Initial and Periodic Database Synchronization

The initial exchange of RIFT is modeled after ISIS with TIDE being equivalent to CSNP and TIRE playing the role of PSNP. The content of TIDEs and TIREs is governed by Table 3.

4.2.3.7. Purging and Roll-Overs

When a node exits the network, if "unpurged", residual stale TIEs may exist in the network until their lifetimes expire (which in case of RIFT is by default a rather long period to prevent ongoing re-origination of TIEs in very large topologies). RIFT does however not have a "purging mechanism" in the traditional sense based on sending specialized "purge" packets. In other routing protocols such mechanism has proven to be complex and fragile based on many years of experience. RIFT simply issues a new, empty version of the TIE with a short lifetime and relies on each node to age out and delete such TIE copy independently. Abundant amounts of memory are available today even on low-end platforms and hence keeping those relatively short-lived extra copies for a while is acceptable. The information will age out and in the meantime all computations will deliver correct results if a node leaves the network due to the new information distributed by its adjacent nodes breaking bi-directional connectivity checks in different computations.

Once a RIFT node issues a TIE with an ID, it SHOULD preserve the ID as long as feasible (also when the protocol restarts), even if the TIE looses all content. The re-advertisement of empty TIE fulfills the purpose of purging any information advertised in previous versions. The originator is free to not re-originate the according empty TIE again or originate an empty TIE with relatively short lifetime to prevent large number of long-lived empty stubs polluting the network. Each node MUST timeout and clean up the according empty TIEs independently.

Upon restart a node MUST, as any link-state implementation, be prepared to receive TIEs with its own system ID and supersede them with equivalent, newly generated, empty TIEs with a higher sequence number. As above, the lifetime can be relatively short since it only needs to exceed the necessary propagation and processing delay by all the nodes that are within the TIE's flooding scope.

TIE sequence numbers are rolled over using the method described in Appendix A. First sequence number of any spontaneously originated TIE (i.e. not originated to override a detected older copy in the network) MUST be a reasonably unpredictable random number in the interval [0, 2^30-1] which will prevent otherwise identical TIE headers to remain "stuck" in the network with content different from TIE originated after reboot. In traditional link-state protocols this is delegated to a 16-bit checksum on packet content. RIFT avoids this design due to the CPU burden presented by computation of such checksums and additional complications tied to the fact that the checksum must be "patched" into the packet after the computation, a difficult proposition in binary hand-crafted formats already and highly incompatible with model-based, serialized formats. The sequence number space is hence consciously chosen to be 64-bits wide to make the occurrence of a TIE with same sequence number but different content as much or even more unlikely than the checksum method. To emulate the "checksum behavior" an implementation could e.g. choose to compute 64-bit checksum over the packet content and use that as first sequence number after reboot.

4.2.3.8. Southbound Default Route Origination

Under certain conditions nodes issue a default route in their South Prefix TIEs with costs as computed in Section 4.3.6.1.

A node X that

1. is NOT overloaded AND

2. has southbound or East-West adjacencies

originates in its south prefix TIE such a default route IIF

1. all other nodes at X's' level are overloaded OR

2. all other nodes at X's' level have NO northbound adjacencies OR

3. X has computed reachability to a default route during N-SPF.

The term "all other nodes at X's' level" describes obviously just the nodes at the same level in the PoD with a viable lower level (otherwise the node South TIEs cannot be reflected and the nodes in e.g. PoD 1 and PoD 2 are "invisible" to each other).

A node originating a southbound default route MUST install a default discard route if it did not compute a default route during N-SPF.

4.2.3.9. Northbound TIE Flooding Reduction

Section 1.4 of the Optimized Link State Routing Protocol [RFC3626] (OLSR) introduces the concept of a "multipoint relay" (MPR) that minimize the overhead of flooding messages in the network by reducing redundant retransmissions in the same region.

A similar technique is applied to RIFT to control northbound flooding. Important observations first:

1. a node MUST flood self-originated North TIEs to all the reachable nodes at the level above which we call the node's "parents";

2. it is typically not necessary that all parents reflood the North TIEs to achieve a complete flooding of all the reachable nodes two levels above which we choose to call the node's "grandparents";

3. to control the volume of its flooding two hops North and yet keep it robust enough, it is advantageous for a node to select a subset of its parents as "Flood Repeaters" (FRs), which combined together deliver two or more copies of its flooding to all of its parents, i.e. the originating node's grandparents;

4. nodes at the same level do NOT have to agree on a specific algorithm to select the FRs, but overall load balancing should be achieved so that different nodes at the same level should tend to select different parents as FRs;

5. there are usually many solutions to the problem of finding a set of FRs for a given node; the problem of finding the minimal set is (similar to) a NP-Complete problem and a globally optimal set may not be the minimal one if load-balancing with other nodes is an important consideration;

6. it is expected that there will be often sets of equivalent nodes at a level L, defined as having a common set of parents at L+1. Applying this observation at both L and L+1, an algorithm may attempt to split the larger problem in a sum of smaller separate problems;

7. it is another expectation that there will be from time to time a broken link between a parent and a grandparent, and in that case the parent is probably a poor FR due to its lower reliability. An algorithm may attempt to eliminate parents with broken

northbound adjacencies first in order to reduce the number of FRs. Albeit it could be argued that relying on higher fanout FRs will slow flooding due to higher replication load reliability of FR's links seems to be a more pressing concern.

In a fully connected Clos Network, this means that a node selects one arbitrary parent as FR and then a second one for redundancy. The computation can be kept relatively simple and completely distributed without any need for synchronization amongst nodes. In a "PoD" structure, where the Level L+2 is partitioned in silos of equivalent grandparents that are only reachable from respective parents, this means treating each silo as a fully connected Clos Network and solve the problem within the silo.

In terms of signaling, a node has enough information to select its set of FRs; this information is derived from the node's parents' Node South TIEs, which indicate the parent's reachable northbound adjacencies to its own parents, i.e. the node's grandparents. A node may send a LIE to a northbound neighbor with the optional boolean field 'you_are_flood_repeater' set to false, to indicate that the northbound neighbor is not a flood repeater for the node that sent the LIE. In that case the northbound neighbor SHOULD NOT reflood northbound TIEs received from the node that sent the LIE. If the 'you_are_flood_repeater' is absent or if 'you_are_flood_repeater' is set to true, then the northbound neighbor is a flood repeater for the node that sent the LIE and MUST reflood northbound TIEs received from that node.

This specification proposes a simple default algorithm that SHOULD be implemented and used by default on every RIFT node.

- let |NA(Node) be the set of Northbound adjacencies of node Node and CN(Node) be the cardinality of |NA(Node);
- let |SA(Node) be the set of Southbound adjacencies of node Node and CS(Node) be the cardinality of |SA(Node);
- let |P(Node) be the set of node Node's parents;
- let |G(Node) be the set of node Node's grandparents. Observe that |G(Node) = |P(|P(Node));
- let N be the child node at level L computing a set of FR;
- let P be a node at level L+1 and a parent node of N, i.e. bi-directionally reachable over adjacency A(N, P);

- o let G be a grandparent node of N, reachable transitively via a parent P over adjacencies ADJ(N, P) and ADJ(P, G). Observe that N does not have enough information to check bidirectional reachability of ADJ(P, G);

- o let R be a redundancy constant integer; a value of 2 or higher for R is RECOMMENDED;

- o let S be a similarity constant integer; a value in range 0 .. 2 for S is RECOMMENDED, the value of 1 SHOULD be used. Two cardinalities are considered as equivalent if their absolute difference is less than or equal to S, i.e. |a-b|<=S.

- o let RND be a 64-bit random number generated by the system once on startup.

The algorithm consists of the following steps:

1. Derive a 64-bits number by XOR’ing ’N’s system ID with RND.

2. Derive a 16-bits pseudo-random unsigned integer PR(N) from the resulting 64-bits number by splitting it in 16-bits-long words W1, W2, W3, W4 (where W1 are the least significant 16 bits of the 64-bits number, and W4 are the most significant 16 bits) and then XOR’ing the circularly shifted resulting words together:

    A. (W1<<1) xor (W2<<2) xor (W3<<3) xor (W4<<4);

       where << is the circular shift operator.

3. Sort the parents by decreasing number of northbound adjacencies (using decreasing system id of the parent as tie-breaker): sort |P(N) by decreasing CN(P), for all P in |P(N), as ordered array |A(N)

4. Partition |A(N) in subarrays |A_k(N) of parents with equivalent cardinality of northbound adjacencies (in other words with equivalent number of grandparents they can reach):

    A. set k=0; // k is the ID of the subarrray

    B. set i=0;

    C. while i < CN(N) do

       i) set j=i;

       ii) while i < CN(N) and CN(|A(N)[j]) - CN(|A(N)[i]) <= S

a. place |A(N)[i] in |A_k(N) // abstract action, maybe noop

b. set i=i+1;

iii) /* At this point j is the index in |A(N) of the first member of |A_k(N) and (i-j) is C_k(N) defined as the cardinality of |A_k(N) */

set k=k+1;

/* At this point k is the total number of subarrays, initialized for the shuffling operation below */

5. shuffle individually each subarrays |A_k(N) of cardinality C_k(N) within |A(N) using the Durstenfeld variation of Fisher-Yates algorithm that depends on N's System ID:

A. while k > 0 do

i) for i from C_k(N)-1 to 1 decrementing by 1 do

a. set j to PR(N) modulo i;

b. exchange |A_k[j] and |A_k[i];

ii) set k=k-1;

6. For each grandparent G, initialize a counter c(G) with the number of its south-bound adjacencies to elected flood repeaters (which is initially zero):

A. for each G in |G(N) set c(G) = 0;

7. Finally keep as FRs only parents that are needed to maintain the number of adjacencies between the FRs and any grandparent G equal or above the redundancy constant R:

A. for each P in reshuffled |A(N);

i) if there exists an adjacency ADJ(P, G) in |NA(P) such that c(G) < R then

a. place P in FR set;

b. for all adjacencies ADJ(P, G') in |NA(P) increment c(G')

B. If any c(G) is still < R, it was not possible to elect a set of FRs that covers all grandparents with redundancy R

Additional rules for flooding reduction:

1. The algorithm MUST be re-evaluated by a node on every change of local adjacencies or reception of a parent South TIE with changed adjacencies. A node MAY apply a hysteresis to prevent excessive amount of computation during periods of network instability just like in case of reachability computation.

2. Upon a change of the flood repeater set, a node SHOULD send out LIEs that grant flood repeater status to newly promoted nodes before it sends LIEs that revoke the status to the nodes that have been newly demoted. This is done to prevent transient behavior where the full coverage of grandparents is not guaranteed. Such a condition is sometimes unavoidable in case of lost LIEs but it will correct itself though at possible transient hit in flooding propagation speeds.

3. A node MUST always flood its self-originated TIEs.

4. A node receiving a TIE originated by a node for which it is not a flood repeater SHOULD NOT reflood such TIEs to its neighbors except for rules in Paragraph 6.

5. The indication of flood reduction capability MUST be carried in the node TIEs and MAY be used to optimize the algorithm to account for nodes that will flood regardless.

6. A node generates TIDEs as usual but when receiving TIREs or TIDEs resulting in requests for a TIE of which the newest received copy came on an adjacency where the node was not flood repeater it SHOULD ignore such requests on first and only first request. Normally, the nodes that received the TIEs as flooding repeaters should satisfy the requesting node and with that no further TIREs for such TIEs will be generated. Otherwise, the next set of TIDEs and TIREs MUST lead to flooding independent of the flood repeater status. This solves a very difficult incast problem on nodes restarting with a very wide fanout, especially northbound. To retrieve the full database they often end up processing many in-rushing copies whereas this approach load-balances the incoming database between adjacent nodes and flood repeaters should guarantee that two copies are sent by different nodes to ensure against any losses.

4.2.3.10. Special Considerations

First, due to the distributed, asynchronous nature of ZTP, it can create temporary convergence anomalies where nodes at higher levels of the fabric temporarily see themselves lower than they belong. Since flooding can begin before ZTP is "finished" and in fact must do so given there is no global termination criteria, information may end up in wrong layers. A special clause when changing level takes care of that.

More difficult is a condition where a node (e.g. a leaf) floods a TIE north towards its grandparent, then its parent reboots, in fact partitioning the grandparent from leaf directly and then the leaf itself reboots. That can leave the grandparent holding the "primary copy" of the leaf's TIE. Normally this condition is resolved easily by the leaf re-originating its TIE with a higher sequence number than it sees in northbound TIEs, here however, when the parent comes back it won't be able to obtain leaf's North TIE from the grandparent easily and with that the leaf may not issue the TIE with a higher sequence number that can reach the grandparent for a long time. Flooding procedures are extended to deal with the problem by the means of special clauses that override the database of a lower level with headers of newer TIEs seen in TIDEs coming from the north.

4.2.4. Reachability Computation

A node has three possible sources of relevant information for reachability computation. A node knows the full topology south of it from the received North Node TIEs or alternately north of it from the South Node TIEs. A node has the set of prefixes with their associated distances and bandwidths from corresponding prefix TIEs.

To compute prefix reachability, a node runs conceptually a northbound and a southbound SPF. We call that N-SPF and S-SPF denoting the direction in which the computation front is progressing.

Since neither computation can "loop", it is possible to compute non-equal-cost or even k-shortest paths [EPPSTEIN] and "saturate" the fabric to the extent desired but we use simple, familiar SPF algorithms and concepts here as example due to their prevalence in today's routing.

4.2.4.1. Northbound SPF

N-SPF MUST use exclusively northbound and East-West adjacencies in the computing node's node North TIEs (since if the node is a leaf it may not have generated a node South TIE) when starting SPF. Observe that N-SPF is really just a one hop variety since Node South TIEs are not re-flooded southbound beyond a single level (or East-West) and with that the computation cannot progress beyond adjacent nodes.

Once progressing, we are using the next higher level's node South TIEs to find according adjacencies to verify backlink connectivity. Just as in case of IS-IS or OSPF, two unidirectional links MUST be associated together to confirm bidirectional connectivity. Particular care MUST be paid that the Node TIEs do not only contain the correct system IDs but matching levels as well.

Default route found when crossing an E-W link SHOULD be used IIF

1. the node itself does NOT have any northbound adjacencies AND

2. the adjacent node has one or more northbound adjacencies

This rule forms a "one-hop default route split-horizon" and prevents looping over default routes while allowing for "one-hop protection" of nodes that lost all northbound adjacencies except at Top-of-Fabric where the links are used exclusively to flood topology information in multi-plane designs.

Other south prefixes found when crossing E-W link MAY be used IIF

1. no north neighbors are advertising same or supersuming non-default prefix AND

2. the node does not originate a non-default supersuming prefix itself.

i.e. the E-W link can be used as a gateway of last resort for a specific prefix only. Using south prefixes across E-W link can be beneficial e.g. on automatic de-aggregation in pathological fabric partitioning scenarios.

A detailed example can be found in Section 5.4.

4.2.4.2. Southbound SPF

S-SPF MUST use exclusively the southbound adjacencies in the node South TIEs, i.e. progresses towards nodes at lower levels. Observe that E-W adjacencies are NEVER used in the computation. This enforces the requirement that a packet traversing in a southbound direction must never change its direction.

S-SPF MUST use northbound adjacencies in node North TIEs to verify backlink connectivity by checking for presence of the link beside correct SystemID and level.

4.2.4.3. East-West Forwarding Within a non-ToF Level

Using south prefixes over horizontal links MAY occur if the N-SPF includes East-West adjacencies in computation. It can protect against pathological fabric partitioning cases that leave only paths to destinations that would necessitate multiple changes of forwarding direction between north and south.

4.2.4.4. East-West Links Within ToF Level

E-W ToF links behave in terms of flooding scopes defined in Section 4.2.3.4 like northbound links and MUST be used exclusively for control plane information flooding. Even though a ToF node could be tempted to use those links during southbound SPF and carry traffic over them this MUST NOT be attempted since it may lead in, e.g. anycast cases to routing loops. An implementation MAY try to resolve the looping problem by following on the ring strictly tie-broken shortest-paths only but the details are outside this specification. And even then, the problem of proper capacity provisioning of such links when they become traffic-bearing in case of failures is vexing.

4.2.5. Automatic Disaggregation on Link & Node Failures

4.2.5.1. Positive, Non-transitive Disaggregation

Under normal circumstances, node's South TIEs contain just the adjacencies and a default route. However, if a node detects that its default IP prefix covers one or more prefixes that are reachable through it but not through one or more other nodes at the same level, then it MUST explicitly advertise those prefixes in an South TIE. Otherwise, some percentage of the northbound traffic for those prefixes would be sent to nodes without according reachability, causing it to be black-holed. Even when not black-holing, the resulting forwarding could 'backhaul' packets through the higher level spines, clearly an undesirable condition affecting the blocking probabilities of the fabric.

We refer to the process of advertising additional prefixes southbound as 'positive de-aggregation' or 'positive dis-aggregation'. Such dis-aggregation is non-transitive, i.e. its' effects are always contained to a single level of the fabric only. Naturally, multiple node or link failures can lead to several independent instances of positive dis-aggregation necessary to prevent looping or bow-tying the fabric.

A node determines the set of prefixes needing de-aggregation using the following steps:

1. A DAG computation in the southern direction is performed first, i.e. the North TIEs are used to find all of prefixes it can reach and the set of next-hops in the lower level for each of them. Such a computation can be easily performed on a fat tree by e.g. setting all link costs in the southern direction to 1 and all northern directions to infinity. We term set of those prefixes |R, and for each prefix, r, in |R, we define its set of next-hops to be |H(r).

2. The node uses reflected South TIEs to find all nodes at the same level in the same PoD and the set of southbound adjacencies for each. The set of nodes at the same level is termed |N and for each node, n, in |N, we define its set of southbound adjacencies to be |A(n).

3. For a given r, if the intersection of |H(r) and |A(n), for any n, is null then that prefix r must be explicitly advertised by the node in an South TIE.

4. Identical set of de-aggregated prefixes is flooded on each of the node's southbound adjacencies. In accordance with the normal flooding rules for an South TIE, a node at the lower level that receives this South TIE SHOULD NOT propagate it south-bound or reflect the disaggregated prefixes back over its adjacencies to nodes at the level from which it was received.

To summarize the above in simplest terms: if a node detects that its default route encompasses prefixes for which one of the other nodes in its level has no possible next-hops in the level below, it has to disaggregate it to prevent black-holing or suboptimal routing through such nodes. Hence a node X needs to determine if it can reach a different set of south neighbors than other nodes at the same level, which are connected to it via at least one common south neighbor. If it can, then prefix disaggregation may be required. If it can't, then no prefix disaggregation is needed. An example of disaggregation is provided in Section 5.3.

A possible algorithm is described last:

1. Create partial_neighbors = (empty), a set of neighbors with partial connectivity to the node X's level from X's perspective. Each entry in the set is a south neighbor of X and a list of nodes of X.level that can't reach that neighbor.

2. A node X determines its set of southbound neighbors X.south_neighbors.

3. For each South TIE originated from a node Y that X has which is at X.level, if Y.south_neighbors is not the same as X.south_neighbors but the nodes share at least one southern neighbor, for each neighbor N in X.south_neighbors but not in Y.south_neighbors, add (N, (Y)) to partial_neighbors if N isn't there or add Y to the list for N.

4. If partial_neighbors is empty, then node X does not disaggregate any prefixes. If node X is advertising disaggregated prefixes in its South TIE, X SHOULD remove them and re-advertise its according South TIEs.

A node X computes reachability to all nodes below it based upon the received North TIEs first. This results in a set of routes, each categorized by (prefix, path_distance, next-hop set). Alternately, for clarity in the following procedure, these can be organized by next-hop set as ((next-hops), {(prefix, path_distance)}). If partial_neighbors isn't empty, then the following procedure describes how to identify prefixes to disaggregate.

```
disaggregated_prefixes = { empty }
nodes_same_level = { empty }
for each South TIE
  if (South TIE.level == X.level and
      X shares at least one S-neighbor with X)
    add South TIE.originator to nodes_same_level
    end if
  end for

for each next-hop-set NHS
  isolated_nodes = nodes_same_level
  for each NH in NHS
    if NH in partial_neighbors
      isolated_nodes =
        intersection(isolated_nodes,
                     partial_neighbors[NH].nodes)
      end if
    end for

  if isolated_nodes is not empty
    for each prefix using NHS
      add (prefix, distance) to disaggregated_prefixes
      end for
    end if
  end for

copy disaggregated_prefixes to X's South TIE
if X's South TIE is different
  schedule South TIE for flooding
  end if
```

Figure 15: Computation of Disaggregated Prefixes

Each disaggregated prefix is sent with the according path_distance. This allows a node to send the same South TIE to each south neighbor. The south neighbor which is connected to that prefix will thus have a shorter path.

Finally, to summarize the less obvious points partially omitted in the algorithms to keep them more tractable:

1. all neighbor relationships MUST perform backlink checks.

2. overload bits as introduced in Section 4.3.1 have to be respected during the computation.

3. all the lower level nodes are flooded the same disaggregated prefixes since we don't want to build an South TIE per node and complicate things unnecessarily. The lower level node that can compute a southbound route to the prefix will prefer it to the disaggregated route anyway based on route preference rules.

4. positively disaggregated prefixes do NOT have to propagate to lower levels. With that the disturbance in terms of new flooding is contained to a single level experiencing failures.

5. disaggregated Prefix South TIEs are not "reflected" by the lower level, i.e. nodes within same level do NOT need to be aware which node computed the need for disaggregation.

6. The fabric is still supporting maximum load balancing properties while not trying to send traffic northbound unless necessary.

In case positive disaggregation is triggered and due to the very stable but un-synchronized nature of the algorithm the nodes may issue the necessary disaggregated prefixes at different points in time. This can lead for a short time to an "incast" behavior where the first advertising router based on the nature of longest prefix match will attract all the traffic. An implementation MAY hence choose different strategies to address this behavior if needed.

To close this section it is worth to observe that in a single plane ToF this disaggregation prevents blackholing up to (K_LEAF * P) link failures in terms of Section 4.1.2 or in other terms, it takes at minimum that many link failures to partition the ToF into multiple planes.

4.2.5.2. Negative, Transitive Disaggregation for Fallen Leaves

As explained in Section 4.1.3 failures in multi-plane Top-of-Fabric or more than (K_LEAF * P) links failing in single plane design can generate fallen leaves. Such scenario cannot be addressed by positive disaggregation only and needs a further mechanism.

4.2.5.2.1. Cabling of Multiple Top-of-Fabric Planes

Let us return in this section to designs with multiple planes as shown in Figure 3. Figure 16 highlights how the ToF is cabled in case of two planes by the means of dual-rings to distribute all the North TIEs within both planes. For people familiar with traditional link-state routing protocols ToF level can be considered equivalent to area 0 in OSPF or level-2 in ISIS which need to be "connected" as well for the protocol to operate correctly.

```
        .     ++==========++          ++==========++
        .     II          II          II          II
        .+----++--+  +----++--+  +----++--+  +----++--+
        .|ToF   A1|  |ToF   B1|  |ToF   B2|  |ToF   A2|
        .++-+-++--+  ++-+-++--+  ++-+-++--+  ++-+-++--+
        . | | II      | | II      | | II      | | II
        . | | ++==========++      | | ++==========++
        . | |         | |         | |         | |
        .
        . ~~~ Highlighted ToF of the previous multi-plane figure ~~
```

Figure 16: Topologically Connected Planes

As described in Section 4.1.3 failures in multi-plane fabrics can lead to blackholes which normal positive disaggregation cannot fix. The mechanism of negative, transitive disaggregation incorporated in RIFT provides the according solution.

4.2.5.2.2. Transitive Advertisement of Negative Disaggregates

A ToF node that discovers that it cannot reach a fallen leaf disaggregates all the prefixes of such leaves. It uses for that purpose negative prefix South TIEs that are, as usual, flooded southwards with the scope defined in Section 4.2.3.4.

Transitively, a node explicitly loses connectivity to a prefix when none of its children advertises it and when the prefix is negatively disaggregated by all of its parents. When that happens, the node originates the negative prefix further down south. Since the mechanism applies recursively south the negative prefix may propagate transitively all the way down to the leaf. This is necessary since leaves connected to multiple planes by means of disjoint paths may have to choose the correct plane already at the very bottom of the fabric to make sure that they don't send traffic towards another leaf using a plane where it is "fallen" at which in point a blackhole is unavoidable.

When the connectivity is restored, a node that disaggregated a prefix withdraws the negative disaggregation by the usual mechanism of re-advertising TIEs omitting the negative prefix.

4.2.5.2.3. Computation of Negative Disaggregates

The document omitted so far the description of the computation necessary to generate the correct set of negative prefixes. Negative prefixes can in fact be advertised due to two different triggers. We describe them consecutively.

The first origination reason is a computation that uses all the node North TIEs to build the set of all reachable nodes by reachability computation over the complete graph and including ToF links. The computation uses the node itself as root. This is compared with the result of the normal southbound SPF as described in Section 4.2.4.2. The difference are the fallen leaves and all their attached prefixes are advertised as negative prefixes southbound if the node does not see the prefix being reachable within southbound SPF.

The second mechanism hinges on the understanding how the negative prefixes are used within the computation as described in Figure 17. When attaching the negative prefixes at certain point in time the negative prefix may find itself with all the viable nodes from the shorter match nexthop being pruned. In other words, all its northbound neighbors provided a negative prefix advertisement. This is the trigger to advertise this negative prefix transitively south and normally caused by the node being in a plane where the prefix belongs to a fabric leaf that has "fallen" in this plane. Obviously, when one of the northbound switches withdraws its negative advertisement, the node has to withdraw its transitively provided negative prefix as well.

4.2.6. Attaching Prefixes

After SPF is run, it is necessary to attach the resulting reachability information in form of prefixes. For S-SPF, prefixes from an North TIE are attached to the originating node with that node's next-hop set and a distance equal to the prefix's cost plus the node's minimized path distance. The RIFT route database, a set of (prefix, prefix-type, attributes, path_distance, next-hop set), accumulates these results.

In case of N-SPF prefixes from each South TIE need to also be added to the RIFT route database. The N-SPF is really just a stub so the computing node needs simply to determine, for each prefix in an South TIE that originated from adjacent node, what next-hops to use to reach that node. Since there may be parallel links, the next-hops to use can be a set; presence of the computing node in the associated Node South TIE is sufficient to verify that at least one link has bidirectional connectivity. The set of minimum cost next-hops from the computing node X to the originating adjacent node is determined.

Each prefix has its cost adjusted before being added into the RIFT route database. The cost of the prefix is set to the cost received plus the cost of the minimum distance next-hop to that neighbor while taking into account its attributes such as mobility per Section 4.3.3. Then each prefix can be added into the RIFT route database with the next-hop set; ties are broken based upon type first and then distance and further on 'PrefixAttributes' and only the best combination is used for forwarding. RIFT route preferences are normalized by the according Thrift [thrift] model type.

An example implementation for node X follows:

```
  for each South TIE
     if South TIE.level > X.level
        next_hop_set = set of minimum cost links to the
            South TIE.originator
        next_hop_cost = minimum cost link to
            South TIE.originator
        end if
     for each prefix P in the South TIE
        P.cost = P.cost + next_hop_cost
        if P not in route_database:
          add (P, P.cost, P.type,
               P.attributes, next_hop_set) to route_database
          end if
        if (P in route_database):
          if route_database[P].cost > P.cost or
                route_database[P].type > P.type:
            update route_database[P] with (P, P.type, P.cost,
                                           P.attributes,
                                           next_hop_set)
          else if route_database[P].cost == P.cost and
                route_database[P].type == P.type:
            update route_database[P] with (P, P.type,
                                           P.cost, P.attributes,
               merge(next_hop_set, route_database[P].next_hop_set))
          else
            // Not preferred route so ignore
            end if
          end if
        end for
     end for
```

Figure 17: Adding Routes from South TIE Positive and Negative Prefixes

After the positive prefixes are attached and tie-broken, negative prefixes are attached and used in case of northbound computation, ideally from the shortest length to the longest. The nexthop adjacencies for a negative prefix are inherited from the longest positive prefix that aggregates it, and subsequently adjacencies to nodes that advertised negative for this prefix are removed.

The rule of inheritance MUST be maintained when the nexthop list for a prefix is modified, as the modification may affect the entries for matching negative prefixes of immediate longer prefix length. For instance, if a nexthop is added, then by inheritance it must be added to all the negative routes of immediate longer prefixes length unless it is pruned due to a negative advertisement for the same next hop. Similarly, if a nexthop is deleted for a given prefix, then it is deleted for all the immediately aggregated negative routes. This will recurse in the case of nested negative prefix aggregations.

The rule of inheritance must also be maintained when a new prefix of intermediate length is inserted, or when the immediately aggregating prefix is deleted from the routing table, making an even shorter aggregating prefix the one from which the negative routes now inherit their adjacencies. As the aggregating prefix changes, all the negative routes must be recomputed, and then again the process may recurse in case of nested negative prefix aggregations.

Although these operations can be computationally expensive, the overall load on devices in the network is low because these computations are not run very often, as positive route advertisements are always preferred over negative ones. This prevents recursion in most cases because positive reachability information never inherits next hops.

To make the negative disaggregation less abstract and provide an example let us consider a ToP node T1 with 4 ToF parents S1..S4 as represented in Figure 18:

```
+----+    +----+    +----+    +----+          N
| S1 |    | S1 |    | S1 |    | S1 |          ^
+----+    +----+    +----+    +----+       W< + >E
 |         |         |         |              v
 |+--------+         |         |              S
 ||+-----------------+         |
 |||+----------------+---------+
 ||||
+----+
| T1 |
+----+
```

Figure 18: A ToP Node with 4 Parents

If all ToF nodes can reach all the prefixes in the network; with RIFT, they will normally advertise a default route south. An abstract Routing Information Base (RIB), more commonly known as a routing table, stores all types of maintained routes including the negative ones and "tie-breaks" for the best one, whereas an abstract Forwarding table (FIB) retains only the ultimately computed "positive" routing instructions. In T1, those tables would look as illustrated in Figure 19:

```
                      +---------+
                      | Default |
                      +---------+
                           |
                           |     +--------+
                           +---> | Via S1 |
                           |     +--------+
                           |
                           |     +--------+
                           +---> | Via S2 |
                           |     +--------+
                           |
                           |     +--------+
                           +---> | Via S3 |
                           |     +---------+
                           |
                           |     +--------+
                           +---> | Via S4 |
                                 +--------+
```

Figure 19: Abstract RIB

In case T1 receives a negative advertisement for prefix 2001:db8::/32 from S1 a negative route is stored in the RIB (indicated by a ~ sign), while the more specific routes to the complementing ToF nodes are installed in FIB. RIB and FIB in T1 now look as illustrated in Figure 20 and Figure 21, respectively:

```
           +---------+                 +-----------------+
           | Default | <-------------- | ~2001:db8::/32  |
           +---------+                 +-----------------+
                |                                |
                |     +--------+                 |     +--------+
                +---> | Via S1 |                 +---> | Via S1 |
                |     +--------+                       +--------+
                |
                |     +--------+
                +---> | Via S2 |
                |     +--------+
                |
                |     +--------+
                +---> | Via S3 |
                |     +---------+
                |
                |     +--------+
                +---> | Via S4 |
                      +--------+
```

Figure 20: Abstract RIB after Negative 2001:db8::/32 from S1

The negative 2001:db8::/32 prefix entry inherits from ::/0, so the positive more specific routes are the complements to S1 in the set of next-hops for the default route. That entry is composed of S2, S3, and S4, or, in other words, it uses all entries the the default route with a "hole punched" for S1 into them. These are the next hops that are still available to reach 2001:db8::/32, now that S1 advertised that it will not forward 2001:db8::/32 anymore. Ultimately, those resulting next-hops are installed in FIB for the more specific route to 2001:db8::/32 as illustrated below:

```
           +---------+                  +---------------+
           | Default |                  | 2001:db8::/32 |
           +---------+                  +---------------+
                |                               |
                |     +--------+                |
                +---> | Via S1 |                |
                |     +--------+                |
                |                               |
                |     +--------+                |     +--------+
                +---> | Via S2 |                +---> | Via S2 |
                |     +--------+                |     +--------+
                |                               |
                |     +--------+                |     +--------+
                +---> | Via S3 |                +---> | Via S3 |
                |     +--------+                |     +--------+
                |                               |
                |     +--------+                |     +--------+
                +---> | Via S4 |                +---> | Via S4 |
                      +--------+                      +--------+
```

Figure 21: Abstract FIB after Negative 2001:db8::/32 from S1

To illustrate matters further let us consider T1 receiving a negative advertisement for prefix 2001:db8:1::/48 from S2, which is stored in RIB again. After the update, the RIB in T1 is illustrated in Figure 22:

```
 +---------+          +----------------+          +------------------+
 | Default | <------  | ~2001:db8::/32 | <------  | ~2001:db8:1::/48 |
 +---------+          +----------------+          +------------------+
      |                     |                              |
      |     +--------+      |     +--------+               |
      +---> | Via S1 |      +---> | Via S1 |               |
      |     +--------+            +--------+               |
      |                                                    |
      |     +--------+                                     |     +--------+
      +---> | Via S2 |                                     +---> | Via S2 |
      |     +--------+                                           +--------+
      |
      |     +--------+
      +---> | Via S3 |
      |     +---------+
      |
      |     +--------+
      +---> | Via S4 |
            +--------+
```

Figure 22: Abstract RIB after Negative 2001:db8:1::/48 from S2

Negative 2001:db8:1::/48 inherits from 2001:db8::/32 now, so the positive more specific routes are the complements to S2 in the set of next hops for 2001:db8::/32, which are S3 and S4, or, in other words, all entries of the parent with the negative holes "punched in" again. After the update, the FIB in T1 shows as illustrated in Figure 23:

```
 +---------+         +---------------+         +-----------------+
 | Default |         | 2001:db8::/32 |         | 2001:db8:1::/48 |
 +---------+         +---------------+         +-----------------+
      |                     |                             |
      |     +--------+      |                             |
      +---> | Via S1 |      |                             |
      |     +--------+      |                             |
      |                     |                             |
      |     +--------+      |     +--------+              |
      +---> | Via S2 |      +---> | Via S2 |              |
      |     +--------+      |     +--------+              |
      |                     |                             |
      |     +--------+      |     +--------+              |     +--------+
      +---> | Via S3 |      +---> | Via S3 |              +---> | Via S3 |
      |     +--------+      |     +--------+              |     +--------+
      |                     |                             |
      |     +--------+      |     +--------+              |     +--------+
      +---> | Via S4 |      +---> | Via S4 |              +---> | Via S4 |
            +--------+            +--------+                    +--------+
```

Figure 23: Abstract FIB after Negative 2001:db8:1::/48 from S2

Further, let us say that S3 stops advertising its service as default gateway. The entry is removed from RIB as usual. In order to update the FIB, it is necessary to eliminate the FIB entry for the default route, as well as all the FIB entries that were created for negative routes pointing to the RIB entry being removed (::/0). This is done recursively for 2001:db8::/32 and then for, 2001:db8:1::/48. The related FIB entries via S3 are removed, as illustrated in Figure 24.

```
 +---------+         +---------------+         +-----------------+
 | Default |         | 2001:db8::/32 |         | 2001:db8:1::/48 |
 +---------+         +---------------+         +-----------------+
      |                     |                           |
      |     +--------+      |                           |
      +---> | Via S1 |      |                           |
      |     +--------+      |                           |
      |                     |                           |
      |     +--------+      |     +--------+            |
      +---> | Via S2 |      +---> | Via S2 |            |
      |     +--------+      |     +--------+            |
      |                     |                           |
      |                     |                           |
      |                     |                           |
      |                     |                           |
      |                     |                           |
      |     +--------+      |     +--------+            |     +--------+
      +---> | Via S4 |      +---> | Via S4 |            +---> | Via S4 |
            +--------+            +--------+                  +--------+
```

Figure 24: Abstract FIB after Loss of S3

Say that at that time, S4 would also disaggregate prefix 2001:db8:1::/48. This would mean that the FIB entry for 2001:db8:1::/48 becomes a discard route, and that would be the signal for T1 to disaggregate prefix 2001:db8:1::/48 negatively in a transitive fashion with its own children.

Finally, let us look at the case where S3 becomes available again as a default gateway, and a negative advertisement is received from S4 about prefix 2001:db8:2::/48 as opposed to 2001:db8:1::/48. Again, a negative route is stored in the RIB, and the more specific route to the complementing ToF nodes are installed in FIB. Since 2001:db8:2::/48 inherits from 2001:db8::/32, the positive FIB routes are chosen by removing S4 from S2, S3, S4. The abstract FIB in T1 now shows as illustrated in Figure 25:

```
                                                +------------------+
                                                | 2001:db8:2::/48  |
                                                +------------------+
                                                        |
  +---------+       +----------------+   +------------------+
  | Default |       | 2001:db8::/32  |   | 2001:db8:1::/48  |
  +---------+       +----------------+   +------------------+
       |                    |                    |      |
       |     +--------+     |                    |      |     +--------+
       +---> | Via S1 |     |                    |      +---> | Via S2 |
       |     +--------+     |                    |      |     +--------+
       |                    |                    |      |
       |     +--------+     |     +--------+     |      |     +--------+
       +---> | Via S2 |     +---> | Via S2 |     |      +---> | Via S3 |
       |     +--------+     |     +--------+     |            +--------+
       |                    |                    |
       |     +--------+     |     +--------+     |     +--------+
       +---> | Via S3 |     +---> | Via S3 |     +---> | Via S3 |
       |     +--------+     |     +--------+     |     +--------+
       |                    |                    |
       |     +--------+     |     +--------+     |     +--------+
       +---> | Via S4 |     +---> | Via S4 |     +---> | Via S4 |
             +--------+            +--------+          +--------+
```

Figure 25: Abstract FIB after Negative 2001:db8:2::/48 from S4

4.2.7. Optional Zero Touch Provisioning (ZTP)

Each RIFT node can operate in zero touch provisioning (ZTP) mode, i.e. it has no configuration (unless it is a ToF or it is configured to operate in the overall topology as leaf and/or support leaf-2-leaf procedures) and it will fully configure itself after being attached to the topology. Configured nodes and nodes operating in ZTP can be mixed and will form a valid topology if achievable.

The derivation of the level of each node happens based on offers received from its neighbors whereas each node (with possibly exceptions of configured leaves) tries to attach at the highest possible point in the fabric. This guarantees that even if the diffusion front reaches a node from "below" faster than from "above", it will greedily abandon already negotiated level derived from nodes topologically below it and properly peers with nodes above.

The fabric is very consciously numbered from the top to allow for PoDs of different heights and minimize number of provisioning necessary, in this case just a TOP_OF_FABRIC flag on every node at the top of the fabric.

This section describes the necessary concepts and procedures for ZTP operation.

4.2.7.1. Terminology

The interdependencies between the different flags and the configured level can be somewhat vexing at first and it may take multiple reads of the glossary to comprehend them.

Automatic Level Derivation: Procedures which allow nodes without level configured to derive it automatically. Only applied if CONFIGURED_LEVEL is undefined.

UNDEFINED_LEVEL: A "null" value that indicates that the level has not been determined and has not been configured. Schemas normally indicate that by a missing optional value without an available defined default.

LEAF_ONLY: An optional configuration flag that can be configured on a node to make sure it never leaves the "bottom of the hierarchy". TOP_OF_FABRIC flag and CONFIGURED_LEVEL cannot be defined at the same time as this flag. It implies CONFIGURED_LEVEL value of 0.

TOP_OF_FABRIC flag: Configuration flag that MUST be provided to all Top-of-Fabric nodes. LEAF_FLAG and CONFIGURED_LEVEL cannot be defined at the same time as this flag. It implies a CONFIGURED_LEVEL value. In fact, it is basically a shortcut for configuring same level at all Top-of-Fabric nodes which is unavoidable since an initial 'seed' is needed for other ZTP nodes to derive their level in the topology. The flag plays an important role in fabrics with multiple planes to enable successful negative disaggregation (Section 4.2.5.2).

CONFIGURED_LEVEL: A level value provided manually. When this is defined (i.e. it is not an UNDEFINED_LEVEL) the node is not participating in ZTP. TOP_OF_FABRIC flag is ignored when this value is defined. LEAF_ONLY can be set only if this value is undefined or set to 0.

DERIVED_LEVEL: Level value computed via automatic level derivation when CONFIGURED_LEVEL is equal to UNDEFINED_LEVEL.

LEAF_2_LEAF: An optional flag that can be configured on a node to make sure it supports procedures defined in Section 4.3.8. In a strict sense it is a capability that implies LEAF_ONLY and the according restrictions. TOP_OF_FABRIC flag is ignored when set at the same time as this flag.

LEVEL_VALUE: In ZTP case the original definition of "level" in Section 3.1 is both extended and relaxed. First, level is defined now as LEVEL_VALUE and is the first defined value of CONFIGURED_LEVEL followed by DERIVED_LEVEL. Second, it is possible for nodes to be more than one level apart to form adjacencies if any of the nodes is at least LEAF_ONLY.

Valid Offered Level (VOL): A neighbor's level received on a valid LIE (i.e. passing all checks for adjacency formation while disregarding all clauses involving level values) persisting for the duration of the holdtime interval on the LIE. Observe that offers from nodes offering level value of 0 do not constitute VOLs (since no valid DERIVED_LEVEL can be obtained from those and consequently `not_a_ztp_offer` MUST be ignored). Offers from LIEs with `not_a_ztp_offer` being true are not VOLs either. If a node maintains parallel adjacencies to the neighbor, VOL on each adjacency is considered as equivalent, i.e. the newest VOL from any such adjacency updates the VOL received from the same node.

Highest Available Level (HAL): Highest defined level value seen from all VOLs received.

Highest Available Level Systems (HALS): Set of nodes offering HAL VOLs.

Highest Adjacency Three Way (HAT): Highest neighbor level of all the formed three way adjacencies for the node.

4.2.7.2. Automatic SystemID Selection

RIFT nodes require a 64 bit SystemID which SHOULD be derived as EUI-64 MA-L derive according to [EUI64]. The organizationally governed portion of this ID (24 bits) can be used to generate multiple IDs if required to indicate more than one RIFT instance."

As matter of operational concern, the router MUST ensure that such identifier is not changing very frequently (or at least not without sending all its TIEs with fairly short lifetimes) since otherwise the network may be left with large amounts of stale TIEs in other nodes (though this is not necessarily a serious problem if the procedures described in Section 7 are implemented).

4.2.7.3. Generic Fabric Example

ZTP forces us to think about miscabled or unusually cabled fabric and how such a topology can be forced into a "lattice" structure which a fabric represents (with further restrictions). Let us consider a necessary and sufficient physical cabling in Figure 26. We assume all nodes being in the same PoD.

```
       .        +---+
       .        | A |                      s   = TOP_OF_FABRIC
       .        | s |                      l   = LEAF_ONLY
       .        ++-++                      l2l = LEAF_2_LEAF
       .         | |
       .      +--+ +--+
       .      |       |
       .   +--++     ++--+
       .   | E |     | F |
       .   |   +-+   |   +-----------+
       .   ++--+ |   ++-++           |
       .    |    |    | |            |
       .    | +-------+ |            |
       .    | |  |      |            |
       .    | |  +----+ |            |
       .    | |       | |            |
       .   ++-++     ++-++           |
       .   | I +-----+ J |           |
       .   |   |     |   +-+         |
       .   ++-++     +--++ |         |
       .    | |         |  |         |
       .    +---------+ |  +------+  |
       .      |       | |         |  |
       .      +-----------------+ |  |
       .              | |       | |  |
       .             ++-++     ++-++ |
       .             | X +-----+ Y +-+
       .             |l2l|     | l |
       .             +---+     +---+
```

Figure 26: Generic ZTP Cabling Considerations

First, we must anchor the "top" of the cabling and that's what the TOP_OF_FABRIC flag at node A is for. Then things look smooth until we have to decide whether node Y is at the same level as I, J (and as consequence, X is south of it) or at the same level as X. This is unresolvable here until we "nail down the bottom" of the topology. To achieve that we choose to use in this example the leaf flags in X and Y. In case where Y would not have a leaf flag it will try to elect highest level offered and end up being in same level as I and J.

4.2.7.4. Level Determination Procedure

A node starting up with UNDEFINED_VALUE (i.e. without a CONFIGURED_LEVEL or any leaf or TOP_OF_FABRIC flag) MUST follow those additional procedures:

1. It advertises its LEVEL_VALUE on all LIEs (observe that this can be UNDEFINED_LEVEL which in terms of the schema is simply an omitted optional value).

2. It computes HAL as numerically highest available level in all VOLs.

3. It chooses then MAX(HAL-1,0) as its DERIVED_LEVEL. The node then starts to advertise this derived level.

4. A node that lost all adjacencies with HAL value MUST hold down computation of new DERIVED_LEVEL for a short period of time unless it has no VOLs from southbound adjacencies. After the holddown expired, it MUST discard all received offers, recompute DERIVED_LEVEL and announce it to all neighbors.

5. A node MUST reset any adjacency that has changed the level it is offering and is in three-way state.

6. A node that changed its defined level value MUST readvertise its own TIEs (since the new 'PacketHeader' will contain a different level than before). Sequence number of each TIE MUST be increased.

7. After a level has been derived the node MUST set the 'not_a_ztp_offer' on LIEs towards all systems offering a VOL for HAL.

8. A node that changed its level SHOULD flush from its link state database TIEs of all other nodes, otherwise stale information may persist on "direction reversal", i.e. nodes that seemed south are now north or east-west. This will not prevent the correct operation of the protocol but could be slightly confusing operationally.

A node starting with LEVEL_VALUE being 0 (i.e. it assumes a leaf function by being configured with the appropriate flags or has a CONFIGURED_LEVEL of 0) MUST follow those additional procedures:

1. It computes HAT per procedures above but does NOT use it to compute DERIVED_LEVEL. HAT is used to limit adjacency formation per Section 4.2.2.

It MAY also follow modified procedures:

1. It may pick a different strategy to choose VOL, e.g. use the VOL value with highest number of VOLs. Such strategies are only possible since the node always remains "at the bottom of the fabric" while another layer could "invert" the fabric by picking its preferred VOL in a different fashion than always trying to achieve the highest viable level.

4.2.7.5. ZTP FSM

This section specifies the precise, normative ZTP FSM and can be omitted unless the reader is pursuing an implementation of the protocol.

Initial state is ComputeBestOffer.

```
   Enter
     |
     v
+------------------+
| ComputeBestOffer |
|                  |<----+
| Entry:           |     | BetterHAL [LEVEL_COMPUTE]
| [LEVEL_COMPUTE]  |     | BetterHAT [LEVEL_COMPUTE]
|                  |     | ChangeLocalConfiguredLevel [StoreConfigLevel,
|                  |     |                             LEVEL_COMPUTE]
|                  |     | ChangeLocalHierarchyIndications
|                  |     |   [StoreLeafFlags,
|                  |     |    LEVEL_COMPUTE]
|                  |     | LostHAT [LEVEL_COMPUTE]
|                  |     | NeighborOffer [IF NoLevelOffered
|                  |     |                  THEN REMOVE_OFFER
|                  |     |                  ELSE IF OfferedLevel > Leaf
|                  |     |                    THEN UPDATE_OFFER
|                  |     |                    ELSE REMOVE_OFFER
|                  |     | ShortTic [RemoveExpiredOffers]
|                  |-----+
|                  |
|                  |<---------------------
|                  |---------------------> (UpdatingClients)
|                  | ComputationDone [-]
+------------------+
    ^   |
    |   | LostHAL [IF AnySouthBoundAdjacenciesPresent
    |   |            THEN UpdateHoldDownTimerToNormalValue
    |   |            ELSE FireHoldDownTimerImmediately]
    |   V
(HoldingDown)
```

ZTP FSM FSM

```
 (ComputeBestOffer)
     |   ^
     |   | ChangeLocalConfiguredLevel [StoreConfiguredLevel]
     |   | ChangeLocalHierarchyIndications [StoreLeafFlags]
     |   | HoldDownExpired [PURGE_OFFERS]
     V   |
 +------------------+
 |  HoldingDown     |
 |                  |<----+
 |                  |     | BetterHAL [-]
 |                  |     | BetterHAT [-]
 |                  |     | ComputationDone [-]
 |                  |     | LostHAL [-]
 |                  |     | LostHat [-]
 |                  |     | NeighborOffer [IF NoLevelOffered
 |                  |     |                  THEN REMOVE_OFFER
 |                  |     |                  ELSE IF OfferedLevel > Leaf
 |                  |     |                    THEN UPDATE_OFFER
 |                  |     |                    ELSE REMOVE_OFFER
 |                  |     | ShortTic [RemoveExpiredOffers,
 |                  |     |           IF HoldDownTimer expired
 |                  |     |             THEN PUSH HoldDownExpired]
 |                  |-----+
 +------------------+
     ^
     |
   (UpdatingClients)
```

ZTP FSM FSM (continued)

```
 (ComputeBestOffer)
     |    ^
     |    | BetterHAL [-]
     |    | BetterHAT [-]
     |    | LostHAT [-]
     |    | ChangeLocalHierarchyIndications [StoreLeafFlags]
     |    | ChangeLocalConfiguredLevel [StoreConfigLevel]
     V    |
+-------------------+
| UpdatingClients   |
|                   |<-----+
| Entry:            |      |
| [UpdateAllLIE-    |      | NeighborOffer [IF NoLevelOffered
|  FSMsWith-        |      |                  THEN REMOVE_OFFER
|  Computation-     |      |                  ELSE IF OfferedLevel > Leaf
|  Results]         |      |                    THEN UPDATE_OFFER
|                   |      |                    ELSE REMOVE_OFFER
|                   |      | ShortTic [RemoveExpiredOffers]
|                   |------+
+-------------------+
     |
     | LostHAL [IF AnySouthBoundAdjacenciesPresent
     |             THEN UpdateHoldDownTimerToNormalValue
     |             ELSE FireHoldDownTimerImmediately]
     V
 (HoldingDown)
```

ZTP FSM FSM (continued)

Events

- ChangeLocalHierarchyIndications: node locally configured with new leaf flags

- ChangeLocalConfiguredLevel: node locally configured with a defined level

- NeighborOffer: a new neighbor offer with optional level and neighbor state

- BetterHAL: better HAL computed internally

- BetterHAT: better HAT computed internally

- LostHAL: lost last HAL in computation

- LostHAT: lost HAT in computation

o ComputationDone: computation performed

o HoldDownExpired: holddown expired

o ShortTic: one second timer tick, to be ignored if transition does not exist

Actions

on ShortTic in HoldingDown finishes in HoldingDown: remove expired offers and if holddown timer expired PUSH_EVENT HoldDownExpired

on ShortTic in ComputeBestOffer finishes in ComputeBestOffer: remove expired offers

on HoldDownExpired in HoldingDown finishes in ComputeBestOffer: PURGE_OFFERS

on ChangeLocalConfiguredLevel in HoldingDown finishes in ComputeBestOffer: store configured level

on ShortTic in UpdatingClients finishes in UpdatingClients: remove expired offers

on BetterHAT in ComputeBestOffer finishes in ComputeBestOffer: LEVEL_COMPUTE

on BetterHAL in HoldingDown finishes in HoldingDown: no action

on ChangeLocalHierarchyIndications in HoldingDown finishes in ComputeBestOffer: store leaf flags

on BetterHAT in UpdatingClients finishes in ComputeBestOffer: no action

on BetterHAL in UpdatingClients finishes in ComputeBestOffer: no action

on ChangeLocalHierarchyIndications in UpdatingClients finishes in ComputeBestOffer: store leaf flags

on LostHAL in HoldingDown finishes in HoldingDown:

on LostHAT in ComputeBestOffer finishes in ComputeBestOffer: LEVEL_COMPUTE

on LostHAT in HoldingDown finishes in HoldingDown: no action on BetterHAT in HoldingDown finishes in HoldingDown: no action

on NeighborOffer in UpdatingClients finishes in UpdatingClients:

if no level offered then REMOVE_OFFER

else

if offered level > leaf then UPDATE_OFFER

else REMOVE_OFFER

on LostHAL in ComputeBestOffer finishes in HoldingDown: if any southbound adjacencies present then update holddown timer to normal duration else fire holddown timer immediately

on LostHAL in UpdatingClients finishes in HoldingDown: if any southbound adjacencies present then update holddown timer to normal duration else fire holddown timer immediately

on ComputationDone in ComputeBestOffer finishes in UpdatingClients: no action

on LostHAT in UpdatingClients finishes in ComputeBestOffer: no action

on ComputationDone in HoldingDown finishes in HoldingDown:

on ChangeLocalConfiguredLevel in ComputeBestOffer finishes in ComputeBestOffer: store configured level and LEVEL_COMPUTE

on ChangeLocalConfiguredLevel in UpdatingClients finishes in ComputeBestOffer: store configured level

on NeighborOffer in ComputeBestOffer finishes in ComputeBestOffer:

if no level offered then REMOVE_OFFER

else

if offered level > leaf then UPDATE_OFFER

else REMOVE_OFFER

on NeighborOffer in HoldingDown finishes in HoldingDown:

if no level offered then REMOVE_OFFER

else

if offered level > leaf then UPDATE_OFFER

else REMOVE_OFFER

on ChangeLocalHierarchyIndications in ComputeBestOffer finishes in ComputeBestOffer: store leaf flags and LEVEL_COMPUTE

on BetterHAL in ComputeBestOffer finishes in ComputeBestOffer: LEVEL_COMPUTE

on Entry into UpdatingClients: update all LIE FSMs with computation results

on Entry into ComputeBestOffer: LEVEL_COMPUTE

Following words are used for well known procedures:

1. PUSH Event: pushes an event to be executed by the FSM upon exit of this action

2. COMPARE_OFFERS: checks whether based on current offers and held last results the events BetterHAL/LostHAL/BetterHAT/LostHAT are necessary and returns them

3. UPDATE_OFFER: store current offer with adjacency holdtime as lifetime and COMPARE_OFFERS, then PUSH according events

4. LEVEL_COMPUTE: compute best offered or configured level and HAL/HAT, if anything changed PUSH ComputationDone

5. REMOVE_OFFER: remove the according offer and COMPARE_OFFERS, PUSH according events

6. PURGE_OFFERS: REMOVE_OFFER for all held offers, COMPARE OFFERS, PUSH according events

4.2.7.6. Resulting Topologies

The procedures defined in Section 4.2.7.4 will lead to the RIFT topology and levels depicted in Figure 27.

```
.        +---+
.        | As|
.        | 24|
.        ++-++
.         | |
.      +--+ +--+
.      |       |
.   +--++     ++--+
.   | E |     | F |
.   | 23+-+   | 23+-----------+
.   ++--+ |   ++-++           |
.    |    |    | |            |
.    | +-------+ |            |
.    | |  |      |            |
.    | |  +----+ |            |
.    | |       | |            |
.   ++-++     ++-++           |
.   | I +-----+ J |           |
.   | 22|     | 22|           |
.   ++--+     +--++           |
.    |           |            |
.    +---------+ |            |
.              | |            |
.             ++-++     +---+ |
.             | X |     | Y +-+
.             | 0 |     | 0 |
.             +---+     +---+
```

Figure 27: Generic ZTP Topology Autoconfigured

In case we imagine the LEAF_ONLY restriction on Y is removed the outcome would be very different however and result in Figure 28. This demonstrates basically that auto configuration makes miscabling detection hard and with that can lead to undesirable effects in cases where leaves are not "nailed" by the accordingly configured flags and arbitrarily cabled.

A node MAY analyze the outstanding level offers on its interfaces and generate warnings when its internal ruleset flags a possible miscabling. As an example, when a node's sees ZTP level offers that differ by more than one level from its chosen level (with proper accounting for leaf's being at level 0) this can indicate miscabling.

```
.        +---+
.        | As|
.        | 24|
.        ++-++
.         | |
.      +--+ +--+
.      |       |
.   +--++     ++--+
.   | E |     | F |
.   | 23+-+   | 23+-------+
.   ++--+ |   ++-++       |
.    |    |    | |        |
.    | +-------+ |        |
.    | |  |      |        |
.    | |  +----+ |        |
.    | |       | |        |
.   ++-++     ++-++     +-+-+
.   | I +-----+ J +-----+ Y |
.   | 22|     | 22|     | 22|
.   ++-++     +--++     ++-++
.    | |         |       | |
.    | +-----------------+ |
.    |           |         |
.    +---------+ |         |
.              | |         |
.             ++-++        |
.             | X +--------+
.             | 0 |
.             +---+
```

Figure 28: Generic ZTP Topology Autoconfigured

4.2.8. Stability Considerations

The autoconfiguration mechanism computes a global maximum of levels by diffusion. The achieved equilibrium can be disturbed massively by all nodes with highest level either leaving or entering the domain (with some finer distinctions not explained further). It is therefore recommended that each node is multi-homed towards nodes with respective HAL offerings. Fortunately, this is the natural state of things for the topology variants considered in RIFT.

4.3. Further Mechanisms

4.3.1. Overload Bit

The overload bit MUST be respected by all necessary SPF computations. A node with the overload bit set SHOULD advertise all locally hosted prefixes both northbound and southbound, all other southbound prefixes SHOULD NOT be advertised.

Leaf nodes SHOULD set the overload bit on all originated Node TIEs. If spine nodes were to forward traffic not intended for the local node, the leaf node would not be able to prevent routing/forwarding loops as it does not have the necessary topology information to do so.

4.3.2. Optimized Route Computation on Leaves

Leaf nodes only have visibility to directly connected nodes and therefore are not required to run "full" SPF computations. Instead, prefixes from neighboring nodes can be gathered to run a "partial" SPF computation in order to build the routing table.

Leaf nodes SHOULD only hold their own N-TIEs, and in cases of L2L implementations, the N-TIEs of their East/West neighbors. Leaf nodes MUST hold all S-TIEs from their neighbors.

Normally, a full network graph is created based on local N-TIEs and remote S-TIEs that it receives from neighbors, at which time, necessary SPF computations are performed. Instead, leaf nodes can simply compute the minimum cost and next-hop set of each leaf neighbor by examining its local adjacencies. Associated N-TIEs are used to determine bi-directionality and derive the next-hop set. Cost is then derived from the minimum cost of the local adjacency to the neighbor and the prefix cost.

Leaf nodes would then attach necessary prefixes as described in Section 4.2.6.

4.3.3. Mobility

The RIFT control plane MUST maintain the real time status of every prefix, to which port it is attached, and to which leaf node that port belongs. This is still true in cases of IP mobility where the point of attachment may change several times a second.

There are two classic approaches to explicitly maintain this information:

timestamp: With this method, the infrastructure SHOULD record the precise time at which the movement is observed. One key advantage of this technique is that it has no dependency on the mobile device. One drawback is that the infrastructure MUST be precisely synchronized in order to be able to compare timestamps as the points of attachment change. This could be accomplished by utilizing Precision Time Protocol (PTP) IEEE Std. 1588 [IEEEstd1588] or 802.1AS [IEEEstd8021AS] which is designed for bridged LANs. Both the precision of the synchronization protocol and the resolution of the timestamp must beat the highest possible roaming time on the fabric. Another drawback is that the presence of a mobile device may only be observed asynchronously, such as when it starts using an IP protocol like ARP [RFC0826], IPv6 Neighbor Discovery [RFC4861], IPv6 Stateless Address Configuration [RFC4862], DHCP [RFC2131], or DHCPv6 [RFC8415].

sequence counter: With this method, a mobile device notifies its point of attachment on arrival with a sequence counter that is incremented upon each movement. On the positive side, this method does not have a dependency on a precise sense of time, since the sequence of movements is kept in order by the mobile device. The disadvantage of this approach is the lack of support for protocols that may be used by the mobile device to register its presence to the leaf node with the capability to provide a sequence counter. Well-known issues with sequence counters such as wrapping and comparison rules MUST be addressed properly. Sequence numbers MUST be compared by a single homogenous source to make operation feasible. Sequence number comparison from multiple heterogeneous sources would be extremely difficult to implement.

RIFT supports a hybrid approach by using an optional 'PrefixSequenceType' attribute (that we also call a 'monotonic clock') that consists of a timestamp and optional sequence number field. When this attribute is present (observe that per data schema the attribute itself is optional but in case it is included the 'timestamp' field is required):

o The leaf node MAY advertise a timestamp of the latest sighting of a prefix, e.g., by snooping IP protocols or the node using the time at which it advertised the prefix. RIFT transports the timestamp within the desired prefix North TIEs as 802.1AS timestamp.

o RIFT MAY interoperate with "Registration Extensions for 6LoWPAN Neighbor Discovery" [RFC8505], which provides a method for registering a prefix with a sequence number called a Transaction ID (TID). In such cases, RIFT SHOULD transport the derived TID without modification.

- RIFT also defines an abstract negative clock (ASNC) (also called an 'undefined' clock). ASNC MUST be considered older than any other defined clock. By default, when a node receives a prefix North TIE that does not contain a 'PrefixSequenceType' attribute, it MUST interpret the absence as ASNC.

- Any prefix present on the fabric in multiple nodes that has the `same` clock is considered as anycast.

- RIFT specification assumes that all nodes are being synchronized to at least 200 milliseconds of precision. This is achievable through the use of NTP [RFC5905]. An implementation MAY provide a way to reconfigure a domain to a different value, we call this variable MAXIMUM_CLOCK_DELTA.

4.3.3.1. Clock Comparison

All monotonic clock values MUST be compared to each other using the following rules:

1. ASNC is older than any other value except ASNC AND

2. Clock with timestamp differing by more than MAXIMUM_CLOCK_DELTA are comparable by using the timestamps only AND

3. Clocks with timestamps differing by less than MAXIMUM_CLOCK_DELTA are comparable by using their TIDs only AND

4. An undefined TID is always older than any other TID AND

5. TIDs are compared using rules of [RFC8505].

4.3.3.2. Interaction between Time Stamps and Sequence Counters

For attachment changes that occur less frequently (e.g. once per second), the timestamp that the RIFT infrastructure captures should be enough to determine the most current discovery. If the point of attachment changes faster than the maximum drift of the time stamping mechanism (i.e. MAXIMUM_CLOCK_DELTA), then a sequence number SHOULD be used to enable necessary precision to determine currency.

The sequence counter in [RFC8505] is encoded as one octet and wraps around using Appendix A.

Within the resolution of MAXIMUM_CLOCK_DELTA, sequence counter values captured during 2 sequential iterations of the same timestamp SHOULD be comparable. This means that with default values, a node may move up to 127 times in a 200 millisecond period and the clocks will remain comparable. This allows the RIFT infrastructure to explicitly assert the most up-to-date advertisement.

4.3.3.3. Anycast vs. Unicast

A unicast prefix can be attached to at most one leaf, whereas an anycast prefix may be reachable via more than one leaf.

If a monotonic clock attribute is provided on the prefix, then the prefix with the 'newest' clock value is strictly preferred. An anycast prefix does not carry a clock or all clock attributes MUST be the same under the rules of Section 4.3.3.1.

Observe that it is important that in mobility events the leaf is re-flooding as quickly as possible the absence of the prefix that moved away.

Observe further that without support for [RFC8505] movements on the fabric within intervals smaller than 100msec will be seen as anycast.

4.3.3.4. Overlays and Signaling

RIFT is agnostic to any overlay technologies and their associated control and transports that run on top of it (e.g. VXLAN). It is expected that leaf nodes and possibly Top-of-Fabric nodes can perform necessary data plane encapsulation.

In the context of mobility, overlays provide another possible solution to avoid injecting mobile prefixes into the fabric as well as improving scalability of the deployment. It makes sense to consider overlays for mobility solutions in IP fabrics. As an example, a mobility protocol such as LISP [RFC6830] may inform the ingress leaf of the location of the egress leaf in real time.

Another possibility is to consider that mobility as an underlay service and support it in RIFT to an extent. The load on the fabric augments with the amount of mobility obviously since a move forces flooding and computation on all nodes in the scope of the move so tunneling from leaf to the Top-of-Fabric may be desired to speed up convergence times.

4.3.4. Key/Value Store

4.3.4.1. Southbound

RIFT supports the southbound distribution of key-value pairs that can be used to distribute information to facilitate higher levels of functionality (e.g. distribution of configuration information). KV

South TIEs may arrive from multiple nodes and therefore MUST execute the following tie-breaking rules for each key:

1. Only KV TIEs received from nodes to which a bi-directional adjacency exists MUST be considered.

2. For each valid KV South TIEs that contains the same key, the value within the South TIE with the highest level will be preferred. If the levels are identical, the highest originating system ID will be preferred. In the case of overlapping keys in the winning South TIE, the behavior is undefined.

Consider that if a node goes down, nodes south of it will lose associated adjacencies causing them to disregard corresponding KVs. New KV South TIEs are advertised to prevent stale information being used by nodes that are farther south. KV advertisements southbound are not a result of independent computation by every node over the same set of South TIEs, but a diffused computation.

4.3.4.2. Northbound

Certain use cases necessitate distribution of essential KV information that is generated by the leaves in the northbound direction. Such information is flooded in KV North TIEs. Since the originator of the KV North TIEs is preserved during flooding, overlapping keys MAY be used. However, to avoid further protocol complexity, the same tie-breaking rules as used in southbound distribution SHOULD be used.

4.3.5. Interactions with BFD

RIFT MAY incorporate BFD [RFC5881] to react quickly to link failures. In such case following procedures are introduced:

After RIFT three-way hello adjacency convergence a BFD session MAY be formed automatically between the RIFT endpoints without further configuration using the exchanged discriminators. The capability of the remote side to support BFD is carried in the LIEs.

In case established BFD session goes Down after it was Up, RIFT adjacency SHOULD be re-initialized and subsequently started from Init after it sees a consecutive BFD Up.

In case of parallel links between nodes each link MAY run its own independent BFD session or they MAY share a session.

If link identifiers or BFD capabilities change, both the LIE and any BFD sessions SHOULD be brought down and back up again. In case only the advertised capabilities change, the node MAY choose to persist the BFD session.

Multiple RIFT instances MAY choose to share a single BFD session, in such cases the behavior for which discriminators are used is undefined. However, RIFT MAY advertise the same link ID for the same interface in multiple instances to "share" discriminators.

BFD TTL follows [RFC5082].

4.3.6. Fabric Bandwidth Balancing

A well understood problem in fabrics is that in case of link failures, it would be ideal to rebalance how much traffic is sent to switches in the next level based on available ingress and egress bandwidth.

RIFT supports a very light weight mechanism that can deal with the problem in an approximate way based on the fact that RIFT is loop-free.

4.3.6.1. Northbound Direction

Every RIFT node SHOULD compute the amount of northbound bandwidth available through neighbors at higher level and modify distance received on default route from this neighbor. Default routes with differing distances SHOULD be used to support weighted ECMP forwarding. We call such a distance Bandwidth Adjusted Distance or BAD. This is best illustrated by a simple example.

```
.   100  x             100 100 MBits
.    |   x              |   |
.  +-+---+-+          +-+---+-+
.  |       |          |       |
.  |Spin111|          |Spin112|
.  +-+---+++          ++----+++
.    |x  ||           ||    ||
.    ||  |+---------------+ ||
.    ||  +---------------+| ||
.    ||               || || ||
.    ||               || || ||
.   -----All Links 10 MBit-------
.    ||               || || ||
.    ||               || || ||
.    ||  +------------+| || ||
.    ||  |+------------+ || ||
.    |x  ||              || ||
.  +-+---+++          +--++-+++
.  |       |          |       |
.  |Leaf111|          |Leaf112|
.  +-------+          +-------+
```

Figure 29: Balancing Bandwidth

Figure 29 depicts an example topology where links between leaf and spine nodes are 10 MBit/s and links from spine nodes northbound are 100 MBit/s. Consider a parallel link failure between Leaf 111 and Spine 111 and as a result, Leaf 111 wants to forward more traffic toward Spine 112. Additionally, we consider an uplink failure on Spine 111.

The local modification of the received default route distance from upper level is achieved by running a relatively simple algorithm where the bandwidth is weighted exponentially, while the distance on the default route represents a multiplier for the bandwidth weight for easy operational adjustments.

On a node, L, use Node TIEs to compute from each non-overloaded northbound neighbor N to compute 3 values:

L_N_u: as sum of the bandwidth available to N

N_u: as sum of the uplink bandwidth available on N

T_N_u: as sum of L_N_u * OVERSUBSCRIPTION_CONSTANT + N_u

For all T_N_u determine the according M_N_u as log_2(next_power_2(T_N_u)) and determine MAX_M_N_u as maximum value of all such M_N_u values.

For each advertised default route from a node N modify the advertised distance D to BAD = D * (1 + MAX_M_N_u - M_N_u) and use BAD instead of distance D to weight balance default forwarding towards N.

For the example above, a simple table of values will help in understanding of the concept. We assume that all default route distances are advertised with D=1 and that OVERSUBSCRIPTION_CONSTANT = 1.

Node	N	T_N_u	M_N_u	BAD
Leaf111	Spine 111	110	7	2
Leaf111	Spine 112	220	8	1
Leaf112	Spine 111	120	7	2
Leaf112	Spine 112	220	8	1

Table 5: BAD Computation

If a calculation produces a result exceeding the range of the type, e.g. bandwidth, the result is set to the highest possible value for that type.

BAD SHOULD be only computed for default routes. A node MAY compute and use BAD for any disaggregated prefixes or other RIFT routes. A node MAY use a different algorithm to weight northbound traffic based on bandwidth. If a different algorithm is used, its successful behavior MUST NOT depend on uniformity of algorithm or synchronization of BAD computations across the fabric. E.g. it is conceivable that leaves could use real time link loads gathered by analytics to change the amount of traffic assigned to each default route next hop.

Furthermore, a change in available bandwidth will only affect, at most, two levels down in the fabric, i.e. the blast radius of bandwidth adjustments is constrained no matter the fabric’s height.

4.3.6.2. Southbound Direction

Due to its loop free nature, during South SPF, a node MAY account for maximum available bandwidth on nodes in lower levels and modify the amount of traffic offered to the next level's southbound nodes. It is worth considering that such computations may be more effective if standardized, but do not have to be. As long as a packet continues to flow southbound, it will take some viable, loop-free path to reach its destination.

4.3.7. Label Binding

A node MAY advertise in its LIEs, a locally significant, downstream assigned, interface specific label. One use of such a label is a hop-by-hop encapsulation allowing forwarding planes to be easily distinguished among multiple RIFT instances.

4.3.8. Leaf to Leaf Procedures

RIFT implementations SHOULD support special East-West adjacencies between leaf nodes. Leaf nodes supporting these procedures MUST:

advertise the LEAF_2_LEAF flag in its node capabilities AND

set the overload bit on all leaf's node TIEs AND

flood only a node's own north and south TIEs over E-W leaf adjacencies AND

always use E-W leaf adjacency in all SPF computations AND

install a discard route for any advertised aggregate routes in a leaf?s TIE AND

never form southbound adjacencies.

This will allow the E-W leaf nodes to exchange traffic strictly for the prefixes advertised in each other's north prefix TIEs (since the southbound computation will find the reverse direction in the other node's TIE and install its north prefixes).

4.3.9. Address Family and Multi Topology Considerations

Multi-Topology (MT)[RFC5120] and Multi-Instance (MI)[RFC8202] concepts are used today in link-state routing protocols to support several domains on the same physical topology. RIFT supports this capability by carrying transport ports in the LIE protocol exchanges.

Multiplexing of LIEs can be achieved by either choosing varying multicast addresses or ports on the same address.

BFD interactions in Section 4.3.5 are implementation dependent when multiple RIFT instances run on the same link.

4.3.10. Reachability of Internal Nodes in the Fabric

RIFT does not require that nodes have reachable addresses in the fabric, though it is clearly desirable for operational purposes. Under normal operating conditions this can be easily achieved by injecting the node's loopback address into North and South Prefix TIEs or other implementation specific mechanisms.

Special considerations arise when a node loses all northbound adjacencies, but is not at the top of the fabric. These are outside the scope of this document and could be discussed in a separate document.

4.3.11. One-Hop Healing of Levels with East-West Links

Based on the rules defined in Section 4.2.4, Section 4.2.3.8 and given presence of E-W links, RIFT can provide a one-hop protection for nodes that lost all their northbound links. This can also be applied to multi-plane designs where complex link set failures occur at the Top-of-Fabric when links are exclusively used for flooding topology information. Section 5.4 outlines this behavior.

4.4. Security

4.4.1. Security Model

An inherent property of any security and ZTP architecture is the resulting trade-off in regard to integrity verification of the information distributed through the fabric vs. provisioning and auto-configuration requirements. At a minimum the security of an established adjacency should be ensured. The stricter the security model the more provisioning must take over the role of ZTP.

RIFT supports the following security models to allow for flexible control by the operator.

- o The most security conscious operators may choose to have control over which ports interconnect between a given pair of nodes, we call this the "Port-Association Model" (PAM). This is achievable by configuring each pair of directly connected ports with a designated shared key or public/private key pair.

- o In physically secure data center locations, operators may choose to control connectivity between entire nodes, we call this the "Node-Association Model" (NAM). A benefit of this model is that it allows for simplified port sparing.

- o In the most relaxed environments, an operator may only choose to control which nodes join a particular fabric. We call this the "Fabric-Association Model" (FAM). This is achievable by using a single shared secret across the entire fabric. Such flexibility makes sense when we consider servers as leaf devices, which are replaced more often than network nodes. In addition, this model allows for simplified node sparing.

- o These models may be mixed throughout the fabric depending upon security requirements at various levels of the fabric and willingness to accept increased provisioning complexity.

In order to support the cases mentioned above, RIFT implementations supports, through operator control, mechanisms that allow for:

a. specification of the appropriate level in the fabric,

b. discovery and reporting of missing connections,

c. discovery and reporting of unexpected connections while preventing them from forming insecure adjacencies.

Operators may only choose to configure the level of each node, but not explicitly configure which connections are allowed. In this case, RIFT will only allow adjacencies to establish between nodes that are in adjacent levels. Operators with the lowest security requirements may not use any configuration to specify which connections are allowed. Nodes in such fabrics could rely fully on ZTP and only established adjacencies between nodes in adjacent levels. Figure 30 illustrates inherent tradeoffs between the different security models.

Some level of link quality verification may be required prior to an adjacency being used for forwarding. For example, an implementation may require that a BFD session comes up before advertising the adjacency.

For the cases outlined above, RIFT has two approaches to enforce that a local port is connected to the correct port on the correct remote node. One approach is to piggy-back on RIFT's authentication mechanism. Assuming the provisioning model (e.g. the YANG model) is flexible enough, operators can choose to provision a unique authentication key for:

a. each pair of ports in "port-association model" or

b. each pair of switches in "node-association model" or

c. each pair of levels or

d. the entire fabric in "fabric-association model".

The other approach is to rely on the system-id, port-id and level fields in the LIE message to validate an adjacency against the expected cabling topology, and optionally introduce some new rules in the FSM to allow the adjacency to come up if the expectations are met.

```
                 ^                 /\                   |
                /|\               /  \                  |
                 |               /    \                 |
                 |              / PAM  \                |
             Increasing        /        \          Increasing
             Integrity        +----------+         Flexibility
                 &           /    NAM     \             &
            Increasing      +--------------+          Less
            Provisioning   /      FAM       \     Configuration
                 |        +------------------+          |
                 |       / Level Provisioning \         |
                 |      +----------------------+       \|/
                 |     /    Zero Configuration  \       v
                      +--------------------------+
```

Figure 30: Security Model

4.4.2. Security Mechanisms

RIFT Security goals are to ensure:

1. authentication

2. message integrity

3. the prevention of replay attacks

4. low processing overhead

5. efficient messaging

Message confidentiality is a non-goal.

The model in the previous section allows a range of security key types that are analogous to the various security association models. PAM and NAM allow security associations at the port or node level using symmetric or asymmetric keys that are pre-installed. FAM argues for security associations to be applied only at a group level or to be refined once the topology has been established. RIFT does not specify how security keys are installed or updated, though it does specify how the key can be used to achieve security goals.

The protocol has provisions for "weak" nonces to prevent replay attacks and includes authentication mechanisms comparable to [RFC5709] and [RFC7987].

4.4.3. Security Envelope

RIFT MUST be carried in a mandatory secure envelope illustrated in Figure 31. Any value in the packet following a security fingerprint MUST be used only after the appropriate fingerprint has been validated.

Local configuration MAY allow for the envelope's integrity checks to be skipped.

```
       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

      UDP Header:
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
      |           Source Port         |       RIFT destination port   |
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
      |           UDP Length          |        UDP Checksum           |
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

      Outer Security Envelope Header:
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
      |           RIFT MAGIC          |         Packet Number         |
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
      |    Reserved   |  RIFT Major   | Outer Key ID  | Fingerprint   |
      |               |    Version    |               |    Length     |
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
      |                                                               |
      ~       Security Fingerprint covers all following content       ~
      |                                                               |
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
      | Weak Nonce Local              | Weak Nonce Remote             |
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
      |            Remaining TIE Lifetime (all 1s in case of LIE)     |
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

      TIE Origin Security Envelope Header:
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
      |              TIE Origin Key ID                |  Fingerprint  |
      |                                               |    Length     |
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
      |                                                               |
      ~       Security Fingerprint covers all following content       ~
      |                                                               |
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

      Serialized RIFT Model Object
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
      |                                                               |
      ~                Serialized RIFT Model Object                   ~
      |                                                               |
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 31: Security Envelope

RIFT MAGIC: 16 bits. Constant value of 0xA1F7 that allows to classify RIFT packets independent of used UDP port.

Packet Number: 16 bits. An optional, per packet type monotonically growing number rolling over using sequence number arithmetic defined in Appendix A. A node SHOULD correctly set the number on subsequent packets or otherwise MUST set the value to 'undefined_packet_number' as provided in the schema. This number can be used to detect losses and misordering in flooding for either operational purposes or in implementation to adjust flooding behavior to current link or buffer quality. This number MUST NOT be used to discard or validate the correctness of packets.

RIFT Major Version: 8 bits. It allows to check whether protocol versions are compatible, i.e. if the serialized object can be decoded at all. An implementation MUST drop packets with unexpected values and MAY report a problem.

Outer Key ID: 8 bits to allow key rollovers. This implies key type and algorithm. Value 0 means that no valid fingerprint was computed. This key ID scope is local to the nodes on both ends of the adjacency.

TIE Origin Key ID: 24 bits. This implies key type and used algorithm. Value 0 means that no valid fingerprint was computed. This key ID scope is global to the RIFT instance since it implies the originator of the TIE so the contained object does not have to be de-serialized to obtain it.

Length of Fingerprint: 8 bits. Length in 32-bit multiples of the following fingerprint (not including lifetime or weak nonces). It allows the structure to be navigated when an unknown key type is present. To clarify, a common corner case when this value is set to 0 is when it signifies an empty (0 bytes long) security fingerprint.

Security Fingerprint: 32 bits * Length of Fingerprint. This is a signature that is computed over all data following after it. If the significant bits of fingerprint are fewer than the 32 bits padded length than the significant bits MUST be left aligned and remaining bits on the right padded with 0s. When using PKI the Security fingerprint originating node uses its private key to create the signature. The original packet can then be verified provided the public key is shared and current.

Remaining TIE Lifetime: 32 bits. In case of anything but TIEs this field MUST be set to all ones and Origin Security Envelope Header

MUST NOT be present in the packet. For TIEs this field represents the remaining lifetime of the TIE and Origin Security Envelope Header MUST be present in the packet. The value in the serialized model object MUST be ignored.

Weak Nonce Local: 16 bits. Local Weak Nonce of the adjacency as advertised in LIEs.

Weak Nonce Remote: 16 bits. Remote Weak Nonce of the adjacency as received in LIEs.

TIE Origin Security Envelope Header: It MUST be present if and only if the Remaining TIE Lifetime field is NOT all ones. It carries through the originators key ID and according fingerprint of the object to protect TIE from modification during flooding. This ensures origin validation and integrity (but does not provide validation of a chain of trust).

Observe that due to the schema migration rules per Appendix B the contained model can be always decoded if the major version matches and the envelope integrity has been validated. Consequently, description of the TIE is available to flood it properly including unknown TIE types.

4.4.4. Weak Nonces

The protocol uses two 16 bit nonces to salt generated signatures. We use the term "nonce" a bit loosely since RIFT nonces are not being changed in every packet as common in cryptography. For efficiency purposes they are changed at a high enough frequency to dwarf practical replay attack attempts. Therefore, we call them "weak" nonces.

Any implementation including RIFT security MUST generate and wrap around local nonces properly. When a nonce increment leads to 'undefined_nonce' value, the value MUST be incremented again immediately. All implementation MUST reflect the neighbor's nonces. An implementation SHOULD increment a chosen nonce on every LIE FSM transition that ends up in a different state from the previous and MUST increment its nonce at least every 5 minutes (such considerations allow for efficient implementations without opening a significant security risk). When flooding TIEs, the implementation MUST use recent (i.e. within allowed difference) nonces reflected in the LIE exchange. The schema specifies the maximum allowable nonce value difference on a packet compared to reflected nonces in the LIEs. Any packet received with nonces deviating more than the allowed delta MUST be discarded without further computation of signatures to prevent computation load attacks.

In cases where a secure implementation does not receive signatures or receives undefined nonces from a neighbor (indicating that it does not support or verify signatures), it is a matter of local policy as to how those packets are treated. A secure implementation MAY refuse forming an adjacency with an implementation that is not advertising signatures or valid nonces, or it MAY continue signing local packets while accepting a neighbor's packets without further security validation.

As a necessary exception, an implementation MUST advertise the remote nonce value as 'undefined_nonce' when the FSM is not in two-way or three-way state and accept an 'undefined_nonce' for its local nonce value on packets in any other state than three-way.

As optional optimization, an implementation MAY send one LIE with previously negotiated neighbor's nonce to try to speed up a neighbor's transition from three-way to one-way and MUST revert to sending 'undefined_nonce' after that.

4.4.5. Lifetime

Protecting flooding lifetime may lead to an excessive number of security fingerprint computations and to avoid this the application generating the fingerprints for advertised TIEs, MAY round the value down to the next 'rounddown_lifetime_interval'. Such an optimization in the presence of security hashes over advancing weak nonces, may not be feasible.

4.4.6. Key Management

As outlined in Section Section 7, either a private shared key or a public/private key pair is used to authenticate the adjacency. Both the key distribution and key synchronization methods are out of scope for this document. Both nodes in the adjacency MUST share the same keys, key type, and algorithm for a given key ID. Mismatched keys will not inter-operate as their security envelopes will be unverifiable.

Key roll-over while the adjacency is active MAY be supported. The specific mechanism is well documented in [RFC6518].

4.4.7. Security Association Changes

There in no mechanism to convert a security envelope for the same key ID from one algorithm to another once the envelope is operational. The recommended procedure to change to a new algorithm is to take the adjacency down, make the necessary changes, and bring the adjacency back up. Obviously, an implementation MAY choose to stop verifying security envelope for the duration of algorithm change to keep the adjacency up but since this introduces a security vulnerability window, such roll-over SHOULD NOT be recommended.

5. Examples

5.1. Normal Operation

```
               ^ N      +--------+          +--------+
Level 2        |        |ToF   21|          |ToF   22|
           E <-*-> W    ++-+--+-++          ++-+--+-++
               |         | |  | |            | |  | |
             S v      P111/2  |P121/2        | |  | |
                         ^ ^  ^ ^            | |  | |
                         | |  | |            | |  | |
          +--------------+ |  +-----------+  | |  | +---------------+
          |                |    |         |  | |  |                 |
         South +-----------------------------+ |  |                 ^
          |    |           |    |         |    |  |             All TIEs
          0/0  0/0        0/0   +-----------------------------+     |
          v    v           v              |    |  |           |     |
          |    |           +-+    +<-0/0----------+           |     |
          |    |             |    |       |    |              |     |
        +-+----++          +-+----++     ++----+-+           ++-----++
Level 1 |       |          |       |     |       |           |       |
        |Spin111|          |Spin112|     |Spin121|           |Spin122|
        +-+---+-+          ++----+-+     +-+---+-+           ++---+--+
          |   |             |   South      |   |              |   |
          |   +---0/0--->-----+ 0/0        |   +----------------+ |
         0/0                |  |  |         |                  | | |
          |   +---<-0/0-----+ |  v         |   +--------------+ | |
          v   |               |  |         |   |                | |
        +-+---+-+          +--+--+-+     +-+---+-+          +---+-+-+
Level 0 |       |          |       |     |       |          |       |
        |Leaf111|          |Leaf112|     |Leaf121|          |Leaf122|
        +-+-----+          +-+---+-+     +--+--+-+          +-+-----+
          +                  +    \        /   +              +
          Prefix111   Prefix112    \      /   Prefix121    Prefix122
                                  multi-homed
                                    Prefix
        +---------- PoD 1 ---------+     +---------- PoD 2 ---------+
```

Figure 32: Normal Case Topology

This section describes RIFT deployment in example topology given in Figure 32 without any node or link failures. We disregard flooding reduction for simplicity's sake and compress the node names in some cases to fit them into the picture better.

First, the following bi-directional adjacencies will be established:

1. ToF 21 (PoD 0) to Spine 111, Spine 112, Spine 121, and Spine 122

2. ToF 22 (PoD 0) to Spine 111, Spine 112, Spine 121, and Spine 122

3. Spine 111 to Leaf 111, Leaf 112

4. Spine 112 to Leaf 111, Leaf 112

5. Spine 121 to Leaf 121, Leaf 122

6. Spine 122 to Leaf 121, Leaf 122

Leaf 111 and Leaf 112 originate N-TIEs for Prefix 111 and Prefix 112 (respectively) to both Spine 111 and Spine 112 (Leaf 112 also originates an N-TIE for the multi-homed prefix). Spine 111 and Spine 112 will then originate their own N-TIEs, as well as flood the N-TIEs received from Leaf 111 and Leaf 112 to both ToF 21 and ToF 22.

Similarly, Leaf 121 and Leaf 122 originate North TIEs for Prefix 121 and Prefix 122 (respectively) to Spine 121 and Spine 122 (Leaf 121 also originates an North TIE for the multi-homed prefix). Spine 121 and Spine 122 will then originate their own North TIEs, as well as flood the North TIEs received from Leaf 121 and Leaf 122 to both ToF 21 and ToF 22.

Spines hold only North TIEs of level 0 for their PoD, while leaves only hold their own North TIEs while at this point, both ToF 21 and ToF 22 (as well as any northbound connected controllers) would have the complete network topology.

ToF 21 and ToF 22 would then originate and flood South TIEs containing any established adjacencies and a default IP route to all spines. Spine 111, Spine 112, Spine 121, and Spine 122 will reflect all Node South TIEs received from ToF 21 to ToF 22, and all Node South TIEs from ToF 22 to ToF 21. South TIEs will not be re-propagated southbound.

South TIEs containing a default IP route are then originated by both Spine 111 and Spine 112 toward Leaf 111 and Leaf 112. Similarly, South TIEs containing a default IP route are originated by Spine 121 and Spine 122 toward Leaf 121 and Leaf 122.

At this point IP connectivity across maximum number of viable paths has been established for all leaves, with routing information constrained to only the minimum amount that allows for normal operation and redundancy.

5.2. Leaf Link Failure

```
                    .  |   |              |   |
                    .+-+---+-+          +-+---+-+
                    .|       |          |       |
                    .|Spin111|          |Spin112|
                    .+-+---+-+          ++----+-+
                    .  |   |             |    |
                    .  |   +---------------+  X
                    .  |                 | |  X Failure
                    .  |   +-------------+ |  X
                    .  |   |               |  |
                    .+-+---+-+          +--+--+-+
                    .|       |          |       |
                    .|Leaf111|          |Leaf112|
                    .+-------+          +-------+
                    .      +                  +
                    .    Prefix111      Prefix112
```

Figure 33: Single Leaf Link Failure

In the event of a link failure between Spine 112 and Leaf 112, both nodes will originate new Node TIEs that contain their connected adjacencies, except for the one that just failed. Leaf 112 will send a Node North TIE to Spine 111. Spine 112 will send a Node North TIE to ToF 21 and ToF 22 as well as a new Node South TIE to Leaf 111 that will be reflected to Spine 111. Necessary SPF recomputation will occur, resulting in Spine 112 no longer being in the forwarding path for Prefix 112.

Spine 111 will also disaggregate Prefix 112 by sending new Prefix South TIE to Leaf 111 and Leaf 112. Though we cover disaggregation in more detail in the following section, it is worth mentioning ini this example as it further illustrates RIFT’s blackhole mitigation mechanism. Consider that Leaf 111 has yet to receive the more specific (disaggregated) route from Spine 111. In such a scenario, traffic from Leaf 111 toward Prefix 112 may still use Spine 112’s default route, causing it to traverse ToF 21 and ToF 22 back down via Spine 111. While this behavior is suboptimal, it is transient in nature and preferred to black-holing traffic.

5.3. Partitioned Fabric

```
                         +--------+          +--------+
 Level 2                 |ToF   21|          |ToF   22|
                         ++-+--+-++          ++-+--+-++
                          | |  | |            | |  | |
                          | |  | |            | |  | 0/0
                          | |  | |            | |  | |
                          | |  | |            | |  | |
           +--------------+ |  +--- XXXXXX +  | |  | +---------------+
           |                |    |         |  | |  |                 |
           |    +-----------------------------+ |  |                 |
           0/0  |           |    |         |    |  |                 |
           |    0/0       0/0    +- XXXXXXXXXXXXXXXXXXXXXXXXX -+     |
           |  1.1/16        |              |    |  |           |     |
           |    |           +-+    +-0/0-----------+           |     |
           |    |             |   1.1./16  |    |              |     |
         +-+----++          +-+-----+     ++-----0/0          ++----0/0
 Level 1 |       |          |       |     |    1.1/16         |   1.1/16
         |Spin111|          |Spin112|     |Spin121|           |Spin122|
         +-+---+-+          ++----+-+     +-+---+-+           ++---+--+
           |   |             |    |         |   |              |   |
           |   +---------------+  |         |   +----------------+ |
           |                 | |  |         |                  | | |
           |   +-------------+ |  |         |   +--------------+ | |
           |   |               |  |         |   |                | |
         +-+---+-+          +--+--+-+     +-+---+-+          +---+-+-+
 Level 3 |       |          |       |     |       |          |       |
         |Leaf111|          |Leaf112|     |Leaf121|          |Leaf122|
         +-+-----+          ++------+     +-----+-+          +-+-----+
           +                 +                  +              +
           Prefix111    Prefix112             Prefix121     Prefix122
                                                1.1/16
```

Figure 34: Fabric Partition

Figure 34 shows one of more catastrophic scenarios where ToF 21 is completely severed from access to Prefix 121 due to a double link failure. If only default routes existed, this would result in 50% of traffic from Leaf 111 and Leaf 112 toward Prefix 121 being black-holed.

The mechanism to resolve this scenario hinges on ToF 21's South TIEs being reflected from Spine 111 and Spine 112 to ToF 22. Once ToF 22 sees that Prefix 121 cannot be reached from ToF 21, it will begin to disaggregate Prefix 121 by advertising a more specific route (1.1/16) along with the default IP prefix route to all spines (ToF 21 still only sends a default route). The result is Spine 111 and Spine112 using the more specific route to Prefix 121 via ToF 22. All other prefixes continue to use the default IP prefix route toward both ToF 21 and ToF 22.

The more specific route for Prefix 121 being advertised by ToF 22 does not need to be propagated further south to the leaves, as they do not benefit from this information. Spine 111 and Spine 112 are only required to reflect the new South Node TIEs received from ToF 22 to ToF 21. In short, only the relevant nodes received the relevant updates, thereby restricting the failure to only the partitioned level rather than burdening the whole fabric with the flooding and recomputation of the new topology information.

To finish our example, the following table shows sets computed by ToF 22 using notation introduced in Section 4.2.5:

|R = Prefix 111, Prefix 112, Prefix 121, Prefix 122

|H (for r=Prefix 111) = Spine 111, Spine 112

|H (for r=Prefix 112) = Spine 111, Spine 112

|H (for r=Prefix 121) = Spine 121, Spine 122

|H (for r=Prefix 122) = Spine 121, Spine 122

|A (for ToF 21) = Spine 111, Spine 112

With that and |H (for r=Prefix 121) and |H (for r=Prefix 122) being disjoint from |A (for ToF 21), ToF 22 will originate an South TIE with Prefix 121 and Prefix 122, which will be flooded to all spines.

5.4. Northbound Partitioned Router and Optional East-West Links

```
.   +                  +                  +
.   X N1               | N2               | N3
.   X                  |                  |
.+--+----+          +--+----+          +--+-----+
.|       |0/0>  <0/0|       |0/0>  <0/0|        |
.|  A01  +----------+  A02  +----------+  A03   | Level 1
.++-+-+--+          ++--+--++          +---+-+-++
. | | |              |  |  |               | | |
. | | +----------------------------------+ | | |
. | |                |  |  |             | | | |
. | +-------------+  |  |  |  +--------------+ |
. |               |  |  |  |  |          | |   |
. | +----------------+  |  +-----------------+ |
. | |             |     |     |          | | | |
. | | +------------------------------------+ | |
. | | |           |     |     |          |   | |
.++-+-+--+        | +---+---+ |        +-+---+-++
.|       |        +-+       +-+        |        |
.|  L01  |          |  L02  |          |  L03   | Level 0
.+-------+          +-------+          +--------+
```

Figure 35: North Partitioned Router

Figure 35 shows a part of a fabric where level 1 is horizontally connected and A01 lost its only northbound adjacency. Based on N-SPF rules in Section 4.2.4.1 A01 will compute northbound reachability by using the link A01 to A02. A02 however, will NOT use this link during N-SPF. The result is A01 utilizing the horizontal link for default route advertisement and unidirectional routing.

Furthermore, if A02 also loses its only northbound adjacency (N2), the situation evolves. A01 will no longer have northbound reachability while it sees A03's northbound adjacencies in South Node TIEs reflected by nodes south of it. As a result, A01 will no longer advertise its default route in accordance with Section 4.2.3.8.

6. Implementation and Operation: Further Details

6.1. Considerations for Leaf-Only Implementation

RIFT can and is intended to be stretched to the lowest level in the IP fabric to integrate ToRs or even servers. Since those entities would run as leaves only, it is worth to observe that a leaf only version is significantly simpler to implement and requires much less resources:

1. Leaf nodes only need to maintain a multipath default route under normal circumstances. However, in cases of catastrophic partitioning, leaf nodes SHOULD be capable of accommodating all the leaf routes in its own PoD to prevent black-holing.

2. Leaf nodes hold only their own North TIEs and South TIEs of Level 1 nodes they are connected to.

3. Leaf nodes do not have to support any type of de-aggregation computation or propagation.

4. Leaf nodes are not required to support overload bit.

5. Leaf nodes do not need to originate S-TIEs unless optional leaf-2-leaf features are desired.

6.2. Considerations for Spine Implementation

Spine nodes will never act as Top of Fabric, and are therefore not required to run a full RIFT implementation. Specifically, spines do not need to perform negative disaggregation computation other than respecting northbound disaggregation advertised from the north.

6.3. Adaptations to Other Proposed Data Center Topologies

```
.  +-----+        +-----+
.  |     |        |     |
.+-+ S0  |        | S1  |
.| ++---++        ++---++
.|  |   |          |   |
.|  | +------------+   |
.|  | | +------------+ |
.|  | |              | |
.| ++-+--+        +--+-++
.| |     |        |     |
.| | A0  |        | A1  |
.| +-+--++        ++---++
.|   |  |          |   |
.|   |  +------------+ |
.|   | +-----------+ | |
.|   | |             | |
.| +-+-+-+        +--+-++
.+-+     |        |     |
.  | L0  |        | L1  |
.  +-----+        +-----+
```

Figure 36: Level Shortcut

RIFT is not strictly limited to Clos topologies. The protocol only requires a sense of "compass rose directionality" either achieved through configuration or derivation of levels. So, conceptually, leaf-2-leaf links and even shortcuts between levels could be included. Figure 36 depicts an example of a shortcut between levels. In this example, sub-optimal routing will occur when traffic is sent from L0 to L1 via S0's default route and back down through A0 or A1. In order to ensure that only default routes from A0 or A1 are used, all leaves would be required to install each others routes.

While various technical and operational challenges may require the use of such modifications, discussion of those topics are outside the scope of this document.

6.4. Originating Non-Default Route Southbound

An implementation MAY choose to originate more specific prefixes (P') southbound instead of only the default route (as described in Section 4.2.3.8). In such a scenario, all addresses carried within the RIFT domain MUST be contained within P'.

7. Security Considerations

7.1. General

One can consider attack vectors where a router may reboot many times while changing its system ID and pollute the network with many stale TIEs or TIEs are sent with very long lifetimes and not cleaned up when the routes vanish. Those attack vectors are not unique to RIFT. Given large memory footprints available today those attacks should be relatively benign. Otherwise a node SHOULD implement a strategy of discarding contents of all TIEs that were not present in the SPF tree over a certain, configurable period of time. Since the protocol, like all modern link-state protocols, is self-stabilizing and will advertise the presence of such TIEs to its neighbors, they can be re-requested again if a computation finds that it sees an adjacency formed towards the system ID of the discarded TIEs.

7.2. ZTP

Section 4.2.7 presents many attack vectors in untrusted environments, starting with nodes that oscillate their level offers to the possibility of nodes offering a three-way adjacency with the highest possible level value and a very long holdtime trying to put itself "on top of the lattice" thereby allowing it to gain access to the whole southbound topology. Session authentication mechanisms are necessary in environments where this is possible and RIFT provides the security envelope to ensure this if so desired.

7.3. Lifetime

Traditional IGP protocols are vulnerable to lifetime modification and replay attacks that can be somewhat mitigated by using techniques like [RFC7987]. RIFT removes this attack vector by protecting the lifetime behind a signature computed over it and additional nonce combination which makes even the replay attack window very small and for practical purposes irrelevant since lifetime cannot be artificially shortened by the attacker.

7.4. Packet Number

Optional packet number is carried in the security envelope without any encryption protection and is hence vulnerable to replay and modification attacks. Contrary to nonces this number must change on every packet and would present a very high cryptographic load if signed. The attack vector packet number present is relatively benign. Changing the packet number by a man-in-the-middle attack will only affect operational validation tools and possibly some performance optimizations on flooding. It is expected that an implementation detecting too many "fake losses" or "misorderings" due to the attack on the packet number would simply suppress its further processing.

7.5. Outer Fingerprint Attacks

A node can try to inject LIE packets observing a conversation on the wire by using the outer key ID albeit it cannot generate valid hashes in case it changes the integrity of the message so the only possible attack is DoS due to excessive LIE validation.

A node can try to replay previous LIEs with changed state that it recorded but the attack is hard to replicate since the nonce combination must match the ongoing exchange and is then limited to a single flap only since both nodes will advance their nonces in case the adjacency state changed. Even in the most unlikely case the attack length is limited due to both sides periodically increasing their nonces.

7.6. TIE Origin Fingerprint DoS Attacks

A compromised node can attempt to generate "fake TIEs" using other nodes' TIE origin key identifiers. Albeit the ultimate validation of the origin fingerprint will fail in such scenarios and not progress further than immediately peering nodes, the resulting denial of service attack seems unavoidable since the TIE origin key id is only protected by the, here assumed to be compromised, node.

7.7. Host Implementations

It can be reasonably expected that with the proliferation of RotH servers, rather than dedicated networking devices, will represent a significant amount of RIFT devices. Given their normally far wider software envelope and access granted to them, such servers are also far more likely to be compromised and present an attack vector on the protocol. Hijacking of prefixes to attract traffic is a trust problem and cannot be easily addressed within the protocol if the trust model is breached, i.e. the server presents valid credentials to form an adjacency and issue TIEs. In an even more devious way, the servers can present DoS (or even DDos) vectors of issuing too many LIE packets, flood large amounts of North TIEs and attempt similar resource overrun attacks. A prudent implementation forming adjacencies to leaves should implement according thresholds mechanisms and raise warnings when e.g. a leaf is advertising an excess number of TIEs or prefixes. Additionally, such implementation could refuse any topology information except the node's own TIEs and authenticated, reflected South Node TIEs at own level.

To isolate possible attack vectors on the leaf to the largest possible extent a dedicated leaf-only implementation could run without any configuration by hard-coding a well-known adjacency key (which can be always rolled-over by the means of e.g. well-known key-value distributed from top of the fabric), leaf level value and always setting overload bit. All other values can be derived by automatic means as described earlier in the protocol specification.

8. IANA Considerations

This specification requests multicast address assignments and standard port numbers. Additionally registries for the schema are requested and suggested values provided that reflect the numbers allocated in the given schema.

8.1. Requested Multicast and Port Numbers

This document requests allocation in the 'IPv4 Multicast Address Space' registry the suggested value of 224.0.0.120 as 'ALL_V4_RIFT_ROUTERS' and in the 'IPv6 Multicast Address Space' registry the suggested value of FF02::A1F7 as 'ALL_V6_RIFT_ROUTERS'.

This document requests allocation in the 'Service Name and Transport Protocol Port Number Registry' the allocation of a suggested value of 914 on udp for 'RIFT_LIES_PORT' and suggested value of 915 for 'RIFT_TIES_PORT'.

8.2. Requested Registries with Suggested Values

This section requests registries that help govern the schema via usual IANA registry procedures. A top level 'RIFT' registry should hold the according registries requested in following sections with their pre-defined values. IANA is requested to store the schema version introducing the allocated value as well as, optionally, its description when present. This will allow to assign different values to an entry depending on schema version. Alternately, IANA is requested to consider a root RIFT/3 registry to store RIFT schema major version 3 values and may be requested in the future to create a RIFT/4 registry under that. In any case, IANA is requested to store the schema version in the entries since that will allow to distinguish between minor versions in the same major schema version. All values not suggested as to be considered `Unassigned`. The range of every registry is a 16-bit integer. Allocation of new values is always performed via `Expert Review` action.

8.2.1. Registry RIFT_v4/common/AddressFamilyType

Address family type.

8.2.1.1. Requested Entries

Name	Value	Schema Version	Description
Illegal	0	4.1	
AddressFamilyMinValue	1	4.1	
IPv4	2	4.1	
IPv6	3	4.1	
AddressFamilyMaxValue	4	4.1	

8.2.2. Registry RIFT_v4/common/HierarchyIndications

Flags indicating node configuration in case of ZTP.

8.2.2.1. Requested Entries

Name	Value	Schema Version	Description
leaf_only	0	4.1	
leaf_only_and_leaf_2_leaf_procedures	1	4.1	
top_of_fabric	2	4.1	

8.2.3. Registry RIFT_v4/common/IEEE802_1ASTimeStampType

Timestamp per IEEE 802.1AS, all values MUST be interpreted in implementation as unsigned.

8.2.3.1. Requested Entries

Name	Value	Schema Version	Description
AS_sec	1	4.1	
AS_nsec	2	4.1	

8.2.4. Registry RIFT_v4/common/IPAddressType

IP address type.

8.2.4.1. Requested Entries

Name	Value	Schema Version	Description
ipv4address	1	4.1	Content is IPv4
ipv6address	2	4.1	Content is IPv6

8.2.5. Registry RIFT_v4/common/IPPrefixType

Prefix advertisement.

@note: for interface addresses the protocol can propagate the address part beyond the subnet mask and on reachability computation that has to be normalized. The non-significant bits can be used for operational purposes.

8.2.5.1. Requested Entries

Name	Value	Schema Version	Description
ipv4prefix	1	4.1	
ipv6prefix	2	4.1	

8.2.6. Registry RIFT_v4/common/IPv4PrefixType

IPv4 prefix type.

8.2.6.1. Requested Entries

Name	Value	Schema Version	Description
address	1	4.1	
prefixlen	2	4.1	

8.2.7. Registry RIFT_v4/common/IPv6PrefixType

IPv6 prefix type.

8.2.7.1. Requested Entries

Name	Value	Schema Version	Description
address	1	4.1	
prefixlen	2	4.1	

8.2.8. Registry RIFT_v4/common/PrefixSequenceType

Sequence of a prefix in case of move.

8.2.8.1. Requested Entries

Name	Value	Schema Version	Description
timestamp	1	4.1	
transactionid	2	4.1	Transaction ID set by client in e.g. in 6LoWPAN.

8.2.9. Registry RIFT_v4/common/RouteType

RIFT route types.

@note: route types which MUST be ordered on their preference PGP prefixes are most preferred attracting traffic north (towards spine) and then south normal prefixes are attracting traffic south (towards leafs), i.e. prefix in NORTH PREFIX TIE is preferred over SOUTH PREFIX TIE.

@note: The only purpose of those values is to introduce an ordering whereas an implementation can choose internally any other values as long the ordering is preserved

8.2.9.1. Requested Entries

Name	Value	Schema Version	Description
Illegal	0	4.1	
RouteTypeMinValue	1	4.1	
Discard	2	4.1	
LocalPrefix	3	4.1	
SouthPGPPrefix	4	4.1	
NorthPGPPrefix	5	4.1	
NorthPrefix	6	4.1	
NorthExternalPrefix	7	4.1	
SouthPrefix	8	4.1	
SouthExternalPrefix	9	4.1	
NegativeSouthPrefix	10	4.1	
RouteTypeMaxValue	11	4.1	

8.2.10. Registry RIFT_v4/common/TIETypeType

Type of TIE.

This enum indicates what TIE type the TIE is carrying. In case the value is not known to the receiver, the TIE MUST be re-flooded. This allows for future extensions of the protocol within the same major schema with types opaque to some nodes UNLESS the flooding scope is not the same as prefix TIE, then a major version revision MUST be performed.

8.2.10.1. Requested Entries

Name	Value	Schema Version	Description
Illegal	0	4.1	
TIETypeMinValue	1	4.1	
NodeTIEType	2	4.1	
PrefixTIEType	3	4.1	
PositiveDisaggregationPrefixTIEType	4	4.1	
NegativeDisaggregationPrefixTIEType	5	4.1	
PGPrefixTIEType	6	4.1	
KeyValueTIEType	7	4.1	
ExternalPrefixTIEType	8	4.1	
PositiveExternalDisaggregationPrefixTIEType	9	4.1	
TIETypeMaxValue	10	4.1	

8.2.11. Registry RIFT_v4/common/TieDirectionType

Direction of TIEs.

8.2.11.1. Requested Entries

Name	Value	Schema Version	Description
Illegal	0	4.1	
South	1	4.1	
North	2	4.1	
DirectionMaxValue	3	4.1	

8.2.12. Registry RIFT_v4/encoding/Community

Prefix community.

8.2.12.1. Requested Entries

Name	Value	Schema Version	Description
top	1	4.1	Higher order bits
bottom	2	4.1	Lower order bits

8.2.13. Registry RIFT_v4/encoding/KeyValueTIEElement

Generic key value pairs.

8.2.13.1. Requested Entries

Name	Value	Schema Version	Description
keyvalues	1	4.1	

8.2.14. Registry RIFT_v4/encoding/LIEPacket

RIFT LIE Packet.

@note: this node's level is already included on the packet header

8.2.14.1. Requested Entries

Name	Value	Schema Version	Description
name	1	4.1	Node or adjacency name.
local_id	2	4.1	Local link ID.
flood_port	3	4.1	UDP port to which we can receive flooded TIEs.
link_mtu_size	4	4.1	Layer 3 MTU, used to discover to mismatch.
link_bandwidth	5	4.1	Local link bandwidth on the interface.
neighbor	6	4.1	Reflects the neighbor once received to provide 3-way connectivity.
pod	7	4.1	Node's PoD.
node_capabilities	10	4.1	Node capabilities shown in LIE. The capabilities MUST match the capabilities shown in the Node TIEs, otherwise the behavior is unspecified. A node detecting the mismatch SHOULD generate according error.
link_capabilities	11	4.1	Capabilities of this link.
holdtime	12	4.1	Required holdtime of the adjacency, i.e. how much time MUST expire without LIE for the adjacency to drop.
label	13	4.1	Unsolicited, downstream assigned locally

			significant label value for the adjacency.
not_a_ztp_offer	21	4.1	Indicates that the level on the LIE MUST NOT be used to derive a ZTP level by the receiving node.
you_are_flood_repeater	22	4.1	Indicates to northbound neighbor that it should be reflooding this node's N-TIEs to achieve flood reduction and balancing for northbound flooding. To be ignored if received from a northbound adjacency.
you_are_sending_too_quickly	23	4.1	Can be optionally set to indicate to neighbor that packet losses are seen on reception based on packet numbers or the rate is too high. The receiver SHOULD temporarily slow down flooding rates.
instance_name	24	4.1	Instance name in case multiple RIFT instances running on same interface.

8.2.15. Registry RIFT_v4/encoding/LinkCapabilities

Link capabilities.

8.2.15.1. Requested Entries

Name	Value	Schema Version	Description
bfd	1	4.1	Indicates that the link is supporting BFD.
v4_forwarding_capable	2	4.1	Indicates whether the interface will support v4 forwarding.

8.2.16. Registry RIFT_v4/encoding/LinkIDPair

LinkID pair describes one of parallel links between two nodes.

8.2.16.1. Requested Entries

Name	Value	Schema Version	Description
local_id	1	4.1	Node-wide unique value for the local link.
remote_id	2	4.1	Received remote link ID for this link.
platform_interface_index	10	4.1	Describes the local interface index of the link.
platform_interface_name	11	4.1	Describes the local interface name.
trusted_outer_security_key	12	4.1	Indication whether the link is secured, i.e. protected by outer key, absence of this element means no indication, undefined outer key means not secured.
bfd_up	13	4.1	Indication whether the link is protected by established BFD session.
address_families	14	4.1	Optional indication which address families are up on the interface

8.2.17. Registry RIFT_v4/encoding/Neighbor

Neighbor structure.

8.2.17.1. Requested Entries

Name	Value	Schema Version	Description
originator	1	4.1	System ID of the originator.
remote_id	2	4.1	ID of remote side of the link.

8.2.18. Registry RIFT_v4/encoding/NodeCapabilities

Capabilities the node supports.

@note: The schema may add to this field future capabilities to indicate whether it will support interpretation of future schema extensions on the same major revision. Such fields MUST be optional and have an implicit or explicit false default value. If a future capability changes route selection or generates blackholes if some nodes are not supporting it then a major version increment is unavoidable.

8.2.18.1. Requested Entries

Name	Value	Schema Version	Description
protocol_minor_version	1	4.1	Must advertise supported minor version dialect that way.
flood_reduction	2	4.1	Can this node participate in flood reduction.
hierarchy_indications	3	4.1	Does this node restrict itself to be top-of-fabric or leaf only (in ZTP) and does it support leaf-2-leaf procedures.

8.2.19. Registry RIFT_v4/encoding/NodeFlags

Indication flags of the node.

8.2.19.1. Requested Entries

Name	Value	Schema Version	Description
overload	1	4.1	Indicates that node is in overload, do not transit traffic through it.

8.2.20. Registry RIFT_v4/encoding/NodeNeighborsTIEElement

neighbor of a node

8.2.20.1. Requested Entries

Name	Value	Schema Version	Description
level	1	4.1	level of neighbor
cost	3	4.1	Cost to neighbor.
link_ids	4	4.1	can carry description of multiple parallel links in a TIE
bandwidth	5	4.1	total bandwith to neighbor, this will be normally sum of the bandwidths of all the parallel links.

8.2.21. Registry RIFT_v4/encoding/NodeTIEElement

Description of a node.

It may occur multiple times in different TIEs but if either

capabilities values do not match or

flags values do not match or

neighbors repeat with different values

the behavior is undefined and a warning SHOULD be generated. Neighbors can be distributed across multiple TIEs however if the sets are disjoint. Miscablings SHOULD be repeated in every node TIE, otherwise the behavior is undefined.

@note: Observe that absence of fields implies defined defaults.

8.2.21.1. Requested Entries

Name	Value	Schema Version	Description
level	1	4.1	Level of the node.
neighbors	2	4.1	Node's neighbors. If neighbor systemID repeats in other node TIEs of same node the behavior is undefined.
capabilities	3	4.1	Capabilities of the node.
flags	4	4.1	Flags of the node.
name	5	4.1	Optional node name for easier operations.
pod	6	4.1	PoD to which the node belongs.
startup_time	7	4.1	optional startup time of the node
miscabled_links	10	4.1	If any local links are miscabled, the indication is flooded.

8.2.22. Registry RIFT_v4/encoding/PacketContent

Content of a RIFT packet.

8.2.22.1. Requested Entries

Name	Value	Schema Version	Description
lie	1	4.1	
tide	2	4.1	
tire	3	4.1	
tie	4	4.1	

8.2.23. Registry RIFT_v4/encoding/PacketHeader

Common RIFT packet header.

8.2.23.1. Requested Entries

Name	Value	Schema Version	Description
major_version	1	4.1	Major version of protocol.
minor_version	2	4.1	Minor version of protocol.
sender	3	4.1	Node sending the packet, in case of LIE/TIRE/TIDE also the originator of it.
level	4	4.1	Level of the node sending the packet, required on everything except LIEs. Lack of presence on LIEs indicates UNDEFINED_LEVEL and is used in ZTP procedures.

8.2.24. Registry RIFT_v4/encoding/PrefixAttributes

Attributes of a prefix.

8.2.24.1. Requested Entries

Name	Value	Schema Version	Description
metric	2	4.1	Distance of the prefix.
tags	3	4.1	Generic unordered set of route tags, can be redistributed to other protocols or use within the context of real time analytics.
monotonic_clock	4	4.1	Monotonic clock for mobile addresses.
loopback	6	4.1	Indicates if the interface is a node loopback.
directly_attached	7	4.1	Indicates that the prefix is directly attached, i.e. should be routed to even if the node is in overload.
from_link	10	4.1	In case of locally originated prefixes, i.e. interface addresses this can describe which link the address belongs to.

8.2.25. Registry RIFT_v4/encoding/PrefixTIEElement

TIE carrying prefixes

8.2.25.1. Requested Entries

Name	Value	Schema Version	Description
prefixes	1	4.1	Prefixes with the associated attributes. If the same prefix repeats in multiple TIEs of same node behavior is unspecified.

8.2.26. Registry RIFT_v4/encoding/ProtocolPacket

RIFT packet structure.

8.2.26.1. Requested Entries

Name	Value	Schema Version	Description
header	1	4.1	
content	2	4.1	

8.2.27. Registry RIFT_v4/encoding/TIDEPacket

TIDE with sorted TIE headers, if headers are unsorted, behavior is undefined.

8.2.27.1. Requested Entries

Name	Value	Schema Version	Description
start_range	1	4.1	First TIE header in the tide packet.
end_range	2	4.1	Last TIE header in the tide packet.
headers	3	4.1	_Sorted_ list of headers.

8.2.28. Registry RIFT_v4/encoding/TIEElement

Single element in a TIE.

Schema enum `common.TIETypeType` in TIEID indicates which elements MUST be present in the TIEElement. In case of mismatch the unexpected elements MUST be ignored. In case of lack of expected element the TIE an error MUST be reported and the TIE MUST be ignored.

This type can be extended with new optional elements for new `common.TIETypeType` values without breaking the major but if it is necessary to understand whether all nodes support the new type a node capability must be added as well.

8.2.28.1. Requested Entries

Name	Value	Schema Version	Description
node	1	4.1	Used in case of enum common.TIETypeType.NodeTIEType.
prefixes	2	4.1	Used in case of enum common.TIETypeType.PrefixTIEType.
positive_disaggregation_prefixes	3	4.1	Positive prefixes (always southbound). It MUST NOT be advertised within a North TIE and ignored otherwise.
negative_disaggregation_prefixes	5	4.1	Transitive, negative prefixes (always southbound) which MUST be aggregated and propagated according to the specification southwards towards lower levels to heal pathological upper level partitioning, otherwise blackholes may occur in multiplane fabrics. It MUST NOT be advertised within a North TIE.
external_prefixes	6	4.1	Externally reimported prefixes.
positive_external_disaggregation_prefixes	7	4.1	Positive external disaggregated prefixes (always southbound). It MUST NOT be advertised within a North TIE and ignored otherwise.
keyvalues	9	4.1	Key-Value store elements.

8.2.29. Registry RIFT_v4/encoding/TIEHeader

Header of a TIE.

@note: TIEID space is a total order achieved by comparing the elements in sequence defined and comparing each value as an unsigned integer of according length.

@note: After sequence number the lifetime received on the envelope must be used for comparison before further fields.

@note: `origination_time` and `origination_lifetime` are normally disregarded for comparison purposes and carried purely for debugging/ security purposes if present. They may be used for comparison of last resort to differentiate otherwise equal ties

8.2.29.1. Requested Entries

Name	Value	Schema Version	Description
tieid	2	4.1	ID of the tie.
seq_nr	3	4.1	Sequence number of the tie.
origination_time	10	4.1	Absolute timestamp when the TIE was generated. This can be used on fabrics with synchronized clock to prevent lifetime modification attacks.
origination_lifetime	12	4.1	Original lifetime when the TIE was generated. This can be used on fabrics with synchronized clock to prevent lifetime modification attacks.

8.2.30. Registry RIFT_v4/encoding/TIEHeaderWithLifeTime

Header of a TIE as described in TIRE/TIDE.

8.2.30.1. Requested Entries

Name	Value	Schema Version	Description
header	1	4.1	
remaining_lifetime	2	4.1	Remaining lifetime that expires down to 0 just like in ISIS. TIEs with lifetimes differing by less than `lifetime_diff2ignore` MUST be considered EQUAL.

8.2.31. Registry RIFT_v4/encoding/TIEID

ID of a TIE.

@note: TIEID space is a total order achieved by comparing the elements in sequence defined and comparing each value as an unsigned integer of according length.

8.2.31.1. Requested Entries

Name	Value	Schema Version	Description
direction	1	4.1	direction of TIE
originator	2	4.1	indicates originator of the TIE
tietype	3	4.1	type of the tie
tie_nr	4	4.1	number of the tie

8.2.32. Registry RIFT_v4/encoding/TIEPacket

TIE packet

8.2.32.1. Requested Entries

Name	Value	Schema Version	Description
header	1	4.1	
element	2	4.1	

8.2.33. Registry RIFT_v4/encoding/TIREPacket

TIRE packet

8.2.33.1. Requested Entries

Name	Value	Schema Version	Description
headers	1	4.1	

9. Acknowledgments

A new routing protocol in its complexity is not a product of a parent but of a village as the author list shows already. However, many more people provided input, fine-combed the specification based on their experience in design, implementation or application of protocols in IP fabrics. This section will make an inadequate attempt in recording their contribution.

Many thanks to Naiming Shen for some of the early discussions around the topic of using IGPs for routing in topologies related to Clos. Russ White to be especially acknowledged for the key conversation on epistemology that allowed to tie current asynchronous distributed systems theory results to a modern protocol design presented in this scope. Adrian Farrel, Joel Halpern, Jeffrey Zhang, Krzysztof Szarkowicz, Nagendra Kumar, Melchior Aelmans, Kaushal Tank, Will Jones, Moin Ahmed, Sandy Zhang and Jordan Head (in no particular order) provided thoughtful comments that improved the readability of the document and found good amount of corners where the light failed to shine. Kris Price was first to mention single router, single arm default considerations. Jeff Tantsura helped out with some initial thoughts on BFD interactions while Jeff Haas corrected several misconceptions about BFD's finer points and helped to improve the security section around leaf considerations. Artur Makutunowicz pointed out many possible improvements and acted as sounding board in regard to modern protocol implementation techniques RIFT is exploring. Barak Gafni formalized first time clearly the problem of partitioned spine and fallen leaves on a (clean) napkin in Singapore that led to the very important part of the specification centered around multiple Top-of-Fabric planes and negative disaggregation. Igor Gashinsky and others shared many thoughts on problems encountered in design and operation of large-scale data center fabrics. Xu Benchong found a delicate error in the flooding procedures and a schema datatype size mismatch.

Last but not least, Alvaro Retana guided the undertaking by asking many necessary procedural and technical questions which did not only improve the content but did also lay out the track towards publication.

10. References

10.1. Normative References

[EUI64] IEEE, "Guidelines for Use of Extended Unique Identifier (EUI), Organizationally Unique Identifier (OUI), and Company ID (CID)", IEEE EUI, <http://standards.ieee.org/develop/regauth/tut/eui.pdf>.

[ISO10589] ISO "International Organization for Standardization", "Intermediate system to Intermediate system intra-domain routeing information exchange protocol for use in conjunction with the protocol for providing the connectionless-mode Network Service (ISO 8473), ISO/IEC 10589:2002, Second Edition.", Nov 2002.

[RFC1982] Elz, R. and R. Bush, "Serial Number Arithmetic", RFC 1982, DOI 10.17487/RFC1982, August 1996, <https://www.rfc-editor.org/info/rfc1982>.

[RFC2328] Moy, J., "OSPF Version 2", STD 54, RFC 2328, DOI 10.17487/RFC2328, April 1998, <https://www.rfc-editor.org/info/rfc2328>.

[RFC2365] Meyer, D., "Administratively Scoped IP Multicast", BCP 23, RFC 2365, DOI 10.17487/RFC2365, July 1998, <https://www.rfc-editor.org/info/rfc2365>.

[RFC4271] Rekhter, Y., Ed., Li, T., Ed., and S. Hares, Ed., "A Border Gateway Protocol 4 (BGP-4)", RFC 4271, DOI 10.17487/RFC4271, January 2006, <https://www.rfc-editor.org/info/rfc4271>.

[RFC4291] Hinden, R. and S. Deering, "IP Version 6 Addressing Architecture", RFC 4291, DOI 10.17487/RFC4291, February 2006, <https://www.rfc-editor.org/info/rfc4291>.

[RFC5082] Gill, V., Heasley, J., Meyer, D., Savola, P., Ed., and C. Pignataro, "The Generalized TTL Security Mechanism (GTSM)", RFC 5082, DOI 10.17487/RFC5082, October 2007, <https://www.rfc-editor.org/info/rfc5082>.

[RFC5120] Przygienda, T., Shen, N., and N. Sheth, "M-ISIS: Multi Topology (MT) Routing in Intermediate System to Intermediate Systems (IS-ISs)", RFC 5120, DOI 10.17487/RFC5120, February 2008, <https://www.rfc-editor.org/info/rfc5120>.

[RFC5303] Katz, D., Saluja, R., and D. Eastlake 3rd, "Three-Way Handshake for IS-IS Point-to-Point Adjacencies", RFC 5303, DOI 10.17487/RFC5303, October 2008, <https://www.rfc-editor.org/info/rfc5303>.

[RFC5549] Le Faucheur, F. and E. Rosen, "Advertising IPv4 Network Layer Reachability Information with an IPv6 Next Hop", RFC 5549, DOI 10.17487/RFC5549, May 2009, <https://www.rfc-editor.org/info/rfc5549>.

[RFC5709] Bhatia, M., Manral, V., Fanto, M., White, R., Barnes, M., Li, T., and R. Atkinson, "OSPFv2 HMAC-SHA Cryptographic Authentication", RFC 5709, DOI 10.17487/RFC5709, October 2009, <https://www.rfc-editor.org/info/rfc5709>.

[RFC5881] Katz, D. and D. Ward, "Bidirectional Forwarding Detection (BFD) for IPv4 and IPv6 (Single Hop)", RFC 5881, DOI 10.17487/RFC5881, June 2010, <https://www.rfc-editor.org/info/rfc5881>.

[RFC5905] Mills, D., Martin, J., Ed., Burbank, J., and W. Kasch, "Network Time Protocol Version 4: Protocol and Algorithms Specification", RFC 5905, DOI 10.17487/RFC5905, June 2010, <https://www.rfc-editor.org/info/rfc5905>.

[RFC6830] Farinacci, D., Fuller, V., Meyer, D., and D. Lewis, "The Locator/ID Separation Protocol (LISP)", RFC 6830, DOI 10.17487/RFC6830, January 2013, <https://www.rfc-editor.org/info/rfc6830>.

[RFC7752] Gredler, H., Ed., Medved, J., Previdi, S., Farrel, A., and S. Ray, "North-Bound Distribution of Link-State and Traffic Engineering (TE) Information Using BGP", RFC 7752, DOI 10.17487/RFC7752, March 2016, <https://www.rfc-editor.org/info/rfc7752>.

[RFC7987] Ginsberg, L., Wells, P., Decraene, B., Przygienda, T., and H. Gredler, "IS-IS Minimum Remaining Lifetime", RFC 7987, DOI 10.17487/RFC7987, October 2016, <https://www.rfc-editor.org/info/rfc7987>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC8200] Deering, S. and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification", STD 86, RFC 8200, DOI 10.17487/RFC8200, July 2017, <https://www.rfc-editor.org/info/rfc8200>.

[RFC8202] Ginsberg, L., Previdi, S., and W. Henderickx, "IS-IS Multi-Instance", RFC 8202, DOI 10.17487/RFC8202, June 2017, <https://www.rfc-editor.org/info/rfc8202>.

[RFC8505] Thubert, P., Ed., Nordmark, E., Chakrabarti, S., and C. Perkins, "Registration Extensions for IPv6 over Low-Power Wireless Personal Area Network (6LoWPAN) Neighbor Discovery", RFC 8505, DOI 10.17487/RFC8505, November 2018, <https://www.rfc-editor.org/info/rfc8505>.

[thrift] Apache Software Foundation, "Thrift Interface Description Language", <https://thrift.apache.org/docs/idl>.

10.2. Informative References

[CLOS] Yuan, X., "On Nonblocking Folded-Clos Networks in Computer Communication Environments", IEEE International Parallel & Distributed Processing Symposium, 2011.

[DIJKSTRA] Dijkstra, E., "A Note on Two Problems in Connexion with Graphs", Journal Numer. Math. , 1959.

[DOT] Ellson, J. and L. Koutsofios, "Graphviz: open source graph drawing tools", Springer-Verlag , 2001.

[DYNAMO] De Candia et al., G., "Dynamo: amazon's highly available key-value store", ACM SIGOPS symposium on Operating systems principles (SOSP '07), 2007.

[EPPSTEIN]
Eppstein, D., "Finding the k-Shortest Paths", 1997.

[FATTREE] Leiserson, C., "Fat-Trees: Universal Networks for Hardware-Efficient Supercomputing", 1985.

[IEEEstd1588]
IEEE, "IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems", IEEE Standard 1588, <https://ieeexplore.ieee.org/document/4579760/>.

[IEEEstd8021AS]
IEEE, "IEEE Standard for Local and Metropolitan Area Networks - Timing and Synchronization for Time-Sensitive Applications in Bridged Local Area Networks", IEEE Standard 802.1AS, <https://ieeexplore.ieee.org/document/5741898/>.

[ISO10589-Second-Edition]
International Organization for Standardization, "Intermediate system to Intermediate system intra-domain routeing information exchange protocol for use in conjunction with the protocol for providing the connectionless-mode Network Service (ISO 8473)", Nov 2002.

[RFC0826] Plummer, D., "An Ethernet Address Resolution Protocol: Or Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware", STD 37, RFC 826, DOI 10.17487/RFC0826, November 1982, <https://www.rfc-editor.org/info/rfc826>.

[RFC2131] Droms, R., "Dynamic Host Configuration Protocol", RFC 2131, DOI 10.17487/RFC2131, March 1997, <https://www.rfc-editor.org/info/rfc2131>.

[RFC3626] Clausen, T., Ed. and P. Jacquet, Ed., "Optimized Link State Routing Protocol (OLSR)", RFC 3626, DOI 10.17487/RFC3626, October 2003, <https://www.rfc-editor.org/info/rfc3626>.

[RFC4861] Narten, T., Nordmark, E., Simpson, W., and H. Soliman, "Neighbor Discovery for IP version 6 (IPv6)", RFC 4861, DOI 10.17487/RFC4861, September 2007, <https://www.rfc-editor.org/info/rfc4861>.

[RFC4862] Thomson, S., Narten, T., and T. Jinmei, "IPv6 Stateless Address Autoconfiguration", RFC 4862, DOI 10.17487/RFC4862, September 2007, <https://www.rfc-editor.org/info/rfc4862>.

[RFC6518] Lebovitz, G. and M. Bhatia, "Keying and Authentication for Routing Protocols (KARP) Design Guidelines", RFC 6518, DOI 10.17487/RFC6518, February 2012, <https://www.rfc-editor.org/info/rfc6518>.

[RFC7938] Lapukhov, P., Premji, A., and J. Mitchell, Ed., "Use of BGP for Routing in Large-Scale Data Centers", RFC 7938, DOI 10.17487/RFC7938, August 2016, <https://www.rfc-editor.org/info/rfc7938>.

[RFC8415] Mrugalski, T., Siodelski, M., Volz, B., Yourtchenko, A., Richardson, M., Jiang, S., Lemon, T., and T. Winters, "Dynamic Host Configuration Protocol for IPv6 (DHCPv6)", RFC 8415, DOI 10.17487/RFC8415, November 2018, <https://www.rfc-editor.org/info/rfc8415>.

[VAHDAT08]
Al-Fares, M., Loukissas, A., and A. Vahdat, "A Scalable, Commodity Data Center Network Architecture", SIGCOMM , 2008.

[Wikipedia]
Wikipedia, "https://en.wikipedia.org/wiki/Serial_number_arithmetic", 2016.

Appendix A. Sequence Number Binary Arithmetic

The only reasonably reference to a cleaner than [RFC1982] sequence number solution is given in [Wikipedia]. It basically converts the problem into two complement's arithmetic. Assuming a straight two complement's subtractions on the bit-width of the sequence number the according >: and =: relations are defined as:

```
U_1, U_2 are 12-bits aligned unsigned version number

D_f is  ( U_1 - U_2 ) interpreted as two complement signed 12-bits
D_b is  ( U_2 - U_1 ) interpreted as two complement signed 12-bits

U_1 >: U_2 IIF D_f > 0 AND D_b < 0
U_1 =: U_2 IIF D_f = 0
```

The >: relationship is anti-symmetric but not transitive. Observe that this leaves >: of the numbers having maximum two complement distance, e.g. (0 and 0x800) undefined in our 12-bits case since D_f and D_b are both -0x7ff.

A simple example of the relationship in case of 3-bit arithmetic follows as table indicating D_f/D_b values and then the relationship of U_1 to U_2:

U2 / U1	0	1	2	3	4	5	6	7
0	+/+	+/-	+/-	+/-	-/-	-/+	-/+	-/+
1	-/+	+/+	+/-	+/-	+/-	-/-	-/+	-/+
2	-/+	-/+	+/+	+/-	+/-	+/-	-/-	-/+
3	-/+	-/+	-/+	+/+	+/-	+/-	+/-	-/-
4	-/-	-/+	-/+	-/+	+/+	+/-	+/-	+/-
5	+/-	-/-	-/+	-/+	-/+	+/+	+/-	+/-
6	+/-	+/-	-/-	-/+	-/+	-/+	+/+	+/-
7	+/-	+/-	+/-	-/-	-/+	-/+	-/+	+/+

U2 / U1	0	1	2	3	4	5	6	7
0	=	>	>	>	?	<	<	<
1	<	=	>	>	>	?	<	<
2	<	<	=	>	>	>	?	<
3	<	<	<	=	>	>	>	?
4	?	<	<	<	=	>	>	>
5	>	?	<	<	<	=	>	>
6	>	>	?	<	<	<	=	>
7	>	>	>	?	<	<	<	=

Appendix B. Information Elements Schema

This section introduces the schema for information elements. The IDL is Thrift [thrift].

On schema changes that

1. change field numbers or

2. add new *required* fields or

3. remove any fields or

4. change lists into sets, unions into structures or

5. change multiplicity of fields or

6. changes name of any field or type or

7. change data types of any field or

8. adds, changes or removes a default value of any *existing* field or

9. removes or changes any defined constant or constant value or

10. changes any enumeration type except extending `common.TIETypeType` (use of enumeration types is generally discouraged)

major version of the schema MUST increase. All other changes MUST increase minor version within the same major.

The above set of rules guarantees that every decoder can process serialized content generated by a higher minor version of the schema and with that the protocol can progress without a 'fork-lift'. Additionally, based on the propagated minor version in encoded content and added optional node capabilities new TIE types or even de-facto mandatory fields can be introduced without progressing the major version albeit only nodes supporting such new extensions would decode them. Given the model is encoded at the source and never re-encoded flooding through nodes not understanding any new extensions will preserve the according fields.

Content serialized using a major version X is NOT expected to be decodable by any implementation using decoder for a model with a major version lower than X.

Observe especially that introducing an optional field does not cause a major version increase even if the fields inside the structure are optional with defaults.

All signed integer as forced by Thrift [thrift] support must be cast for internal purposes to equivalent unsigned values without discarding the signedness bit. An implementation SHOULD try to avoid using the signedness bit when generating values.

The schema is normative.

B.1. common.thrift

```
/**
    Thrift file with common definitions for RIFT
*/


/** @note MUST be interpreted in implementation as unsigned 64 bits.
 *        The implementation SHOULD NOT use the MSB.
 */
typedef i64      SystemIDType
typedef i32      IPv4Address
/** this has to be long enough to accomodate prefix */
typedef binary   IPv6Address
/** @note MUST be interpreted in implementation as unsigned */
typedef i16      UDPPortType
/** @note MUST be interpreted in implementation as unsigned */
typedef i32      TIENrType
/** @note MUST be interpreted in implementation as unsigned */
typedef i32      MTUSizeType
/** @note MUST be interpreted in implementation as unsigned
    rolling over number */
typedef i64      SeqNrType
/** @note MUST be interpreted in implementation as unsigned */
typedef i32      LifeTimeInSecType
/** @note MUST be interpreted in implementation as unsigned */
typedef i8       LevelType
/** optional, recommended monotonically increasing number
    _per packet type per adjacency_
    that can be used to detect losses/misordering/restarts.
    @note MUST be interpreted in implementation as unsigned
          rolling over number */
typedef i16      PacketNumberType
/** @note MUST be interpreted in implementation as unsigned */
typedef i32      PodType
/** @note MUST be interpreted in implementation as unsigned.
          This is carried in the
          security envelope and MUST fit into 8 bits. */
typedef i8       VersionType
/** @note MUST be interpreted in implementation as unsigned */
typedef i16      MinorVersionType
/** @note MUST be interpreted in implementation as unsigned */
typedef i32      MetricType
/** @note MUST be interpreted in implementation as unsigned
          and unstructured */
typedef i64      RouteTagType
/** @note MUST be interpreted in implementation as unstructured
```

```
          label value */
typedef i32      LabelType
/** @note MUST be interpreted in implementation as unsigned */
typedef i32      BandwithInMegaBitsType
/** @note Key Value key ID type */
typedef string   KeyIDType
/** node local, unique identification for a link (interface/tunnel
  * etc. Basically anything RIFT runs on). This is kept
  * at 32 bits so it aligns with BFD [RFC5880] discriminator size.
  */
typedef i32    LinkIDType
typedef string KeyNameType
typedef i8     PrefixLenType
/** timestamp in seconds since the epoch */
typedef i64    TimestampInSecsType
/** security nonce.
    @note MUST be interpreted in implementation as rolling
          over unsigned value */
typedef i16    NonceType
/** LIE FSM holdtime type */
typedef i16    TimeIntervalInSecType
/** Transaction ID type for prefix mobility as specified by RFC6550,
    value MUST be interpreted in implementation as unsigned  */
typedef i8     PrefixTransactionIDType
/** Timestamp per IEEE 802.1AS, all values MUST be interpreted in
    implementation as unsigned.  */
struct IEEE802_1ASTimeStampType {
    1: required     i64     AS_sec;
    2: optional     i32     AS_nsec;
}
/** generic counter type */
typedef i64 CounterType
/** Platform Interface Index type, i.e. index of interface on hardware,
    can be used e.g. with RFC5837 */
typedef i32 PlatformInterfaceIndex

/** Flags indicating node configuration in case of ZTP.
 */
enum HierarchyIndications {
    /** forces level to `leaf_level` and enables according procedures */
    leaf_only                            = 0,
    /** forces level to `leaf_level` and enables according procedures */
    leaf_only_and_leaf_2_leaf_procedures = 1,
    /** forces level to `top_of_fabric` and enables according
        procedures */
    top_of_fabric                        = 2,
}
```

```
const PacketNumberType  undefined_packet_number    = 0
/** This MUST be used when node is configured as top of fabric in ZTP.
    This is kept reasonably low to alow for fast ZTP convergence on
    failures. */
const LevelType   top_of_fabric_level              = 24
/** default bandwidth on a link */
const BandwithInMegaBitsType  default_bandwidth    = 100
/** fixed leaf level when ZTP is not used */
const LevelType   leaf_level                  = 0
const LevelType   default_level               = leaf_level
const PodType     default_pod                 = 0
const LinkIDType  undefined_linkid            = 0

/** default distance used */
const MetricType  default_distance         = 1
/** any distance larger than this will be considered infinity */
const MetricType  infinite_distance       = 0x7FFFFFFF
/** represents invalid distance */
const MetricType  invalid_distance        = 0
const bool overload_default               = false
const bool flood_reduction_default        = true
/** default LIE FSM holddown time */
const TimeIntervalInSecType   default_lie_holdtime  = 3
/** default ZTP FSM holddown time */
const TimeIntervalInSecType   default_ztp_holdtime  = 1
/** by default LIE levels are ZTP offers */
const bool default_not_a_ztp_offer        = false
/** by default everyone is repeating flooding */
const bool default_you_are_flood_repeater = true
/** 0 is illegal for SystemID */
const SystemIDType IllegalSystemID        = 0
/** empty set of nodes */
const set<SystemIDType> empty_set_of_nodeids = {}
/** default lifetime of TIE is one week */
const LifeTimeInSecType default_lifetime      = 604800
/** default lifetime when TIEs are purged is 5 minutes */
const LifeTimeInSecType purge_lifetime        = 300
/** round down interval when TIEs are sent with security hashes
    to prevent excessive computation. **/
const LifeTimeInSecType rounddown_lifetime_interval = 60
/** any `TieHeader` that has a smaller lifetime difference
    than this constant is equal (if other fields equal). This
    constant MUST be larger than `purge_lifetime` to avoid
    retransmissions */
const LifeTimeInSecType lifetime_diff2ignore  = 400

/** default UDP port to run LIEs on */
const UDPPortType     default_lie_udp_port       =  914
```

```
/** default UDP port to receive TIEs on, that can be peer specific */
const UDPPortType     default_tie_udp_flood_port =  915

/** default MTU link size to use */
const MTUSizeType     default_mtu_size           = 1400
/** default link being BFD capable */
const bool            bfd_default                = true

/** undefined nonce, equivalent to missing nonce */
const NonceType       undefined_nonce            = 0;
/** outer security key id, MUST be interpreted as in implementation
    as unsigned */
typedef i8            OuterSecurityKeyID
/** security key id, MUST be interpreted as in implementation
    as unsigned */
typedef i32           TIESecurityKeyID
/** undefined key */
const TIESecurityKeyID undefined_securitykey_id   = 0;
/** Maximum delta (negative or positive) that a mirrored nonce can
    deviate from local value to be considered valid. If nonces are
    changed every minute on both sides this opens statistically
    a `maximum_valid_nonce_delta` minutes window of identical LIEs,
    TIE, TI(x)E replays.
    The interval cannot be too small since LIE FSM may change
    states fairly quickly during ZTP without sending LIEs*/
const i16             maximum_valid_nonce_delta  = 5;

/** Direction of TIEs. */
enum TieDirectionType {
    Illegal           = 0,
    South             = 1,
    North             = 2,
    DirectionMaxValue = 3,
}

/** Address family type. */
enum AddressFamilyType {
   Illegal               = 0,
   AddressFamilyMinValue  = 1,
   IPv4     = 2,
   IPv6     = 3,
   AddressFamilyMaxValue  = 4,
}

/** IPv4 prefix type. */
struct IPv4PrefixType {
    1: required IPv4Address    address;
    2: required PrefixLenType  prefixlen;
```

```
}

/** IPv6 prefix type. */
struct IPv6PrefixType {
    1: required IPv6Address    address;
    2: required PrefixLenType  prefixlen;
}

/** IP address type. */
union IPAddressType {
    /** Content is IPv4 */
    1: optional IPv4Address   ipv4address;
    /** Content is IPv6 */
    2: optional IPv6Address   ipv6address;
}

/** Prefix advertisement.

    @note: for interface
        addresses the protocol can propagate the address part beyond
        the subnet mask and on reachability computation that has to
        be normalized. The non-significant bits can be used
        for operational purposes.
*/
union IPPrefixType {
    1: optional IPv4PrefixType   ipv4prefix;
    2: optional IPv6PrefixType   ipv6prefix;
}

/** Sequence of a prefix in case of move.
 */
struct PrefixSequenceType {
    1: required IEEE802_1ASTimeStampType  timestamp;
    /** Transaction ID set by client in e.g. in 6LoWPAN. */
    2: optional PrefixTransactionIDType   transactionid;
}

/** Type of TIE.

    This enum indicates what TIE type the TIE is carrying.
    In case the value is not known to the receiver,
    the TIE MUST be re-flooded. This allows for
    future extensions of the protocol within the same major schema
    with types opaque to some nodes UNLESS the flooding scope is not
    the same as prefix TIE, then a major version revision MUST
    be performed.
*/
enum TIETypeType {
```

```
    Illegal                                      = 0,
    TIETypeMinValue                              = 1,
    /** first legal value */
    NodeTIEType                                  = 2,
    PrefixTIEType                                = 3,
    PositiveDisaggregationPrefixTIEType          = 4,
    NegativeDisaggregationPrefixTIEType          = 5,
    PGPrefixTIEType                              = 6,
    KeyValueTIEType                              = 7,
    ExternalPrefixTIEType                        = 8,
    PositiveExternalDisaggregationPrefixTIEType = 9,
    TIETypeMaxValue                              = 10,
}

/** RIFT route types.

    @note: route types which MUST be ordered on their preference
           PGP prefixes are most preferred attracting
           traffic north (towards spine) and then south
           normal prefixes are attracting traffic south
           (towards leafs), i.e. prefix in NORTH PREFIX TIE
           is preferred over SOUTH PREFIX TIE.

    @note: The only purpose of those values is to introduce an
           ordering whereas an implementation can choose internally
           any other values as long the ordering is preserved
 */
enum RouteType {
    Illegal               =  0,
    RouteTypeMinValue     =  1,
    /** First legal value. */
    /** Discard routes are most preferred */
    Discard               =  2,

    /** Local prefixes are directly attached prefixes on the
     *  system such as e.g. interface routes.
     */
    LocalPrefix           =  3,
    /** Advertised in S-TIEs */
    SouthPGPPrefix        =  4,
    /** Advertised in N-TIEs */
    NorthPGPPrefix        =  5,
    /** Advertised in N-TIEs */
    NorthPrefix           =  6,
    /** Externally imported north */
    NorthExternalPrefix   =  7,
    /** Advertised in S-TIEs, either normal prefix or positive
        disaggregation */
```

```
    SouthPrefix           =  8,
    /** Externally imported south */
    SouthExternalPrefix   =  9,
    /** Negative, transitive prefixes are least preferred */
    NegativeSouthPrefix   = 10,
    RouteTypeMaxValue     = 11,
}
```

B.2. encoding.thrift

```
 /**
     Thrift file for packet encodings for RIFT
 */

 include "common.thrift"


 namespace rs models
 namespace py encoding

 /** Represents protocol encoding schema major version */
 const common.VersionType protocol_major_version = 4
 /** Represents protocol encoding schema minor version */
 const common.MinorVersionType protocol_minor_version =  1

 /** Common RIFT packet header. */
 struct PacketHeader {
     /** Major version of protocol. */
     1: required common.VersionType      major_version =
             protocol_major_version;
     /** Minor version of protocol. */
     2: required common.MinorVersionType minor_version =
             protocol_minor_version;
     /** Node sending the packet, in case of LIE/TIRE/TIDE
         also the originator of it. */
     3: required common.SystemIDType  sender;
     /** Level of the node sending the packet, required on everything
         except LIEs. Lack of presence on LIEs indicates UNDEFINED_LEVEL
         and is used in ZTP procedures.
      */
     4: optional common.LevelType            level;
 }

 /** Prefix community. */
 struct Community {
```

```
     /** Higher order bits */
     1: required i32          top;
     /** Lower order bits */
     2: required i32          bottom;
 }

 /** Neighbor structure.  */
 struct Neighbor {
     /** System ID of the originator. */
     1: required common.SystemIDType        originator;
     /** ID of remote side of the link. */
     2: required common.LinkIDType          remote_id;
 }

 /** Capabilities the node supports.

     @note: The schema may add to this
     field future capabilities to indicate whether it will support
     interpretation of future schema extensions on the same major
     revision. Such fields MUST be optional and have an implicit or
     explicit false default value. If a future capability changes route
     selection or generates blackholes if some nodes are not supporting
     it then a major version increment is unavoidable.
 */
 struct NodeCapabilities {
     /** Must advertise supported minor version dialect that way. */
     1: required common.MinorVersionType        protocol_minor_version =
             protocol_minor_version;
     /** Can this node participate in flood reduction. */
     2: optional bool                           flood_reduction =
             common.flood_reduction_default;
     /** Does this node restrict itself to be top-of-fabric or
         leaf only (in ZTP) and does it support leaf-2-leaf
         procedures. */
     3: optional common.HierarchyIndications    hierarchy_indications;
 }

 /** Link capabilities. */
 struct LinkCapabilities {
     /** Indicates that the link is supporting BFD. */
     1: optional bool                           bfd =
             common.bfd_default;
     /** Indicates whether the interface will support v4 forwarding.

         @note: This MUST be set to true when LIEs from a v4 address are
                sent and MAY be set to true in LIEs on v6 address. If v4
                and v6 LIEs indicate contradicting information the
                behavior is unspecified. */
```

```
     2: optional bool                           v4_forwarding_capable =
             true;
 }

 /** RIFT LIE Packet.

     @note: this node's level is already included on the packet header
 */
 struct LIEPacket {
     /** Node or adjacency name. */
     1: optional string                    name;
     /** Local link ID. */
     2: required common.LinkIDType         local_id;
     /** UDP port to which we can receive flooded TIEs. */
     3: required common.UDPPortType        flood_port =
             common.default_tie_udp_flood_port;
     /** Layer 3 MTU, used to discover to mismatch. */
     4: optional common.MTUSizeType        link_mtu_size =
             common.default_mtu_size;
     /** Local link bandwidth on the interface. */
     5: optional common.BandwithInMegaBitsType
             link_bandwidth = common.default_bandwidth;
     /** Reflects the neighbor once received to provide
         3-way connectivity. */
     6: optional Neighbor                  neighbor;
     /** Node's PoD. */
     7: optional common.PodType            pod =
             common.default_pod;
     /** Node capabilities shown in LIE. The capabilities
         MUST match the capabilities shown in the Node TIEs, otherwise
         the behavior is unspecified. A node detecting the mismatch
         SHOULD generate according error. */
    10: required NodeCapabilities          node_capabilities;
    /** Capabilities of this link. */
    11: optional LinkCapabilities          link_capabilities;
    /** Required holdtime of the adjacency, i.e. how much time
        MUST expire without LIE for the adjacency to drop. */
    12: required common.TimeIntervalInSecType
             holdtime = common.default_lie_holdtime;
    /** Unsolicited, downstream assigned locally significant label
        value for the adjacency. */
    13: optional common.LabelType          label;
     /** Indicates that the level on the LIE MUST NOT be used
         to derive a ZTP level by the receiving node. */
    21: optional bool                      not_a_ztp_offer =
             common.default_not_a_ztp_offer;
    /** Indicates to northbound neighbor that it should
        be reflooding this node's N-TIEs to achieve flood reduction and
```

```
        balancing for northbound flooding. To be ignored if received
        from a northbound adjacency. */
    22: optional bool                      you_are_flood_repeater =
              common.default_you_are_flood_repeater;
    /** Can be optionally set to indicate to neighbor that packet losses
        are seen on reception based on packet numbers or the rate is
        too high. The receiver SHOULD temporarily slow down
        flooding rates.
     */
    23: optional bool                      you_are_sending_too_quickly =
              false;
    /** Instance name in case multiple RIFT instances running on same
        interface. */
    24: optional string                    instance_name;
 }

 /** LinkID pair describes one of parallel links between two nodes. */
 struct LinkIDPair {
     /** Node-wide unique value for the local link. */
     1: required common.LinkIDType      local_id;
     /** Received remote link ID for this link. */
     2: required common.LinkIDType      remote_id;

     /** Describes the local interface index of the link. */
    10: optional common.PlatformInterfaceIndex platform_interface_index;
    /** Describes the local interface name. */
    11: optional string                        platform_interface_name;
    /** Indication whether the link is secured, i.e. protected by
        outer key, absence of this element means no indication,
        undefined outer key means not secured. */
    12: optional common.OuterSecurityKeyID
                 trusted_outer_security_key;
    /** Indication whether the link is protected by established
        BFD session. */
    13: optional bool                          bfd_up;
    /** Optional indication which address families are up on the
        interface */
    14: optional set<common.AddressFamilyType> address_families;
 }

 /** ID of a TIE.

     @note: TIEID space is a total order achieved by comparing
            the elements in sequence defined and comparing each
            value as an unsigned integer of according length.
 */
 struct TIEID {
     /** direction of TIE */
```

```
     1: required common.TieDirectionType    direction;
     /** indicates originator of the TIE */
     2: required common.SystemIDType        originator;
     /** type of the tie */
     3: required common.TIETypeType         tietype;
     /** number of the tie */
     4: required common.TIENrType           tie_nr;
 }

 /** Header of a TIE.

    @note: TIEID space is a total order achieved by comparing
           the elements in sequence defined and comparing each
           value as an unsigned integer of according length.

    @note: After sequence number the lifetime received on the envelope
           must be used for comparison before further fields.

    @note: `origination_time` and `origination_lifetime` are
           normally disregarded for comparison purposes and carried
           purely for debugging/security purposes if present.
           They may be used for comparison of last resort to
           differentiate otherwise equal ties
 */
 struct TIEHeader {
     /** ID of the tie. */
     2: required TIEID                                tieid;
     /** Sequence number of the tie. */
     3: required common.SeqNrType                     seq_nr;

     /** Absolute timestamp when the TIE
         was generated. This can be used on fabrics with
         synchronized clock to prevent lifetime modification attacks. */
    10: optional common.IEEE802_1ASTimeStampType   origination_time;
    /** Original lifetime when the TIE
        was generated. This can be used on fabrics with
        synchronized clock to prevent lifetime modification attacks. */
    12: optional common.LifeTimeInSecType          origination_lifetime;
 }

 /** Header of a TIE as described in TIRE/TIDE.
 */
 struct TIEHeaderWithLifeTime {
     1: required     TIEHeader                          header;
     /** Remaining lifetime that expires down to 0 just like in ISIS.
         TIEs with lifetimes differing by less than
         `lifetime_diff2ignore` MUST be considered EQUAL. */
     2: required     common.LifeTimeInSecType        remaining_lifetime;
```

```
}

/** TIDE with sorted TIE headers, if headers are unsorted, behavior
    is undefined. */
struct TIDEPacket {
    /** First TIE header in the tide packet. */
    1: required TIEID                       start_range;
    /** Last TIE header in the tide packet. */
    2: required TIEID                       end_range;
    /** _Sorted_ list of headers. */
    3: required list<TIEHeaderWithLifeTime> headers;
}

/** TIRE packet */
struct TIREPacket {
    1: required set<TIEHeaderWithLifeTime>  headers;
}

/** neighbor of a node */
struct NodeNeighborsTIEElement {
    /** level of neighbor */
    1: required common.LevelType                level;
    /**  Cost to neighbor.

         @note: All parallel links to same node
         incur same cost, in case the neighbor has multiple
         parallel links at different cost, the largest distance
         (highest numerical value) MUST be advertised.

         @note: any neighbor with cost <= 0 MUST be ignored
                in computations */
    3: optional common.MetricType               cost
                = common.default_distance;
    /** can carry description of multiple parallel links in a TIE */
    4: optional set<LinkIDPair>                 link_ids;

    /** total bandwith to neighbor, this will be normally sum of the
        bandwidths of all the parallel links. */
    5: optional common.BandwithInMegaBitsType
                bandwidth = common.default_bandwidth;
}

/** Indication flags of the node. */
struct NodeFlags {
    /** Indicates that node is in overload, do not transit traffic
        through it. */
    1: optional bool         overload = common.overload_default;
}
```

```
/** Description of a node.

    It may occur multiple times in different TIEs but if either
        <list>
         <t>capabilities values do not match or</t>
        <t>flags values do not match or</t>
        <t>neighbors repeat with different values</t>
        </list>

    the behavior is undefined and a warning SHOULD be generated.
    Neighbors can be distributed across multiple TIEs however if
    the sets are disjoint. Miscablings SHOULD be repeated in every
    node TIE, otherwise the behavior is undefined.

    @note: Observe that absence of fields implies defined defaults.
*/
struct NodeTIEElement {
    /** Level of the node. */
    1: required common.LevelType            level;
    /** Node’s neighbors. If neighbor systemID repeats in other
        node TIEs of same node the behavior is undefined. */
    2: required map<common.SystemIDType,
                NodeNeighborsTIEElement>    neighbors;
    /** Capabilities of the node. */
    3: required NodeCapabilities            capabilities;
    /** Flags of the node. */
    4: optional NodeFlags                   flags;
    /** Optional node name for easier operations. */
    5: optional string                      name;
    /** PoD to which the node belongs. */
    6: optional common.PodType              pod;
    /** optional startup time of the node */
    7: optional common.TimestampInSecsType  startup_time;

    /** If any local links are miscabled, the indication is flooded. */
   10: optional set<common.LinkIDType>      miscabled_links;

}

/** Attributes of a prefix. */
struct PrefixAttributes {
    /** Distance of the prefix. */
    2: required common.MetricType            metric
            = common.default_distance;
    /** Generic unordered set of route tags, can be redistributed
        to other protocols or use within the context of real time
        analytics. */
    3: optional set<common.RouteTagType>     tags;
```

```
     /** Monotonic clock for mobile addresses. */
     4: optional common.PrefixSequenceType    monotonic_clock;
     /** Indicates if the interface is a node loopback. */
     6: optional bool                         loopback = false;
     /** Indicates that the prefix is directly attached, i.e. should be
         routed to even if the node is in overload. */
     7: optional bool                         directly_attached = true;

     /** In case of locally originated prefixes, i.e. interface
         addresses this can describe which link the address
         belongs to. */
    10: optional common.LinkIDType            from_link;
 }

 /** TIE carrying prefixes */
 struct PrefixTIEElement {
     /** Prefixes with the associated attributes.
         If the same prefix repeats in multiple TIEs of same node
         behavior is unspecified. */
     1: required map<common.IPPrefixType, PrefixAttributes> prefixes;
 }

 /** Generic key value pairs. */
 struct KeyValueTIEElement {
     /** @note: if the same key repeats in multiple TIEs of same node
         or with different values, behavior is unspecified */
     1: required map<common.KeyIDType,string>    keyvalues;
 }

 /** Single element in a TIE.

     Schema enum `common.TIETypeType`
     in TIEID indicates which elements MUST be present
     in the TIEElement. In case of mismatch the unexpected
     elements MUST be ignored. In case of lack of expected
     element the TIE an error MUST be reported and the TIE
     MUST be ignored.

     This type can be extended with new optional elements
     for new `common.TIETypeType` values without breaking
     the major but if it is necessary to understand whether
     all nodes support the new type a node capability must
     be added as well.
  */
 union TIEElement {
     /** Used in case of enum common.TIETypeType.NodeTIEType. */
     1: optional NodeTIEElement     node;
     /** Used in case of enum common.TIETypeType.PrefixTIEType. */
```

```
    2: optional PrefixTIEElement          prefixes;
    /** Positive prefixes (always southbound).
        It MUST NOT be advertised within a North TIE and
        ignored otherwise.
    */
    3: optional PrefixTIEElement   positive_disaggregation_prefixes;
    /** Transitive, negative prefixes (always southbound) which
        MUST be aggregated and propagated
        according to the specification
        southwards towards lower levels to heal
        pathological upper level partitioning, otherwise
        blackholes may occur in multiplane fabrics.
        It MUST NOT be advertised within a North TIE.
    */
    5: optional PrefixTIEElement   negative_disaggregation_prefixes;
    /** Externally reimported prefixes. */
    6: optional PrefixTIEElement          external_prefixes;
    /** Positive external disaggregated prefixes (always southbound).
        It MUST NOT be advertised within a North TIE and
        ignored otherwise.
    */
    7: optional PrefixTIEElement
            positive_external_disaggregation_prefixes;
    /** Key-Value store elements. */
    9: optional KeyValueTIEElement keyvalues;
}

/** TIE packet */
struct TIEPacket {
    1: required TIEHeader  header;
    2: required TIEElement element;
}

/** Content of a RIFT packet. */
union PacketContent {
    1: optional LIEPacket     lie;
    2: optional TIDEPacket    tide;
    3: optional TIREPacket    tire;
    4: optional TIEPacket     tie;
}

/** RIFT packet structure. */
struct ProtocolPacket {
    1: required PacketHeader  header;
    2: required PacketContent content;
}
```

Appendix C. Constants

C.1. Configurable Protocol Constants

This section gathers constants that are provided in the schema files and in the document.

	Type	Value
LIE IPv4 Multicast Address	Default Value, Configurable	224.0.0.120 or all-rift-routers to be assigned in IPv4 Multicast Address Space Registry in Local Network Control Block
LIE IPv6 Multicast Address	Default Value, Configurable	FF02::A1F7 or all-rift-routers to be assigned in IPv6 Multicast Address Assignments
LIE Destination Port	Default Value, Configurable	914
Level value for TOP_OF_FABRIC flag	Constant	24
Default LIE Holdtime	Default Value, Configurable	3 seconds
TIE Retransmission Interval	Default Value	1 second
TIDE Generation Interval	Default Value, Configurable	5 seconds
MIN_TIEID signifies start of TIDEs	Constant	TIE Key with minimal values: TIEID(originator=0, tietype=TIETypeMinValue, tie_nr=0, direction=South)
MAX_TIEID signifies end of TIDEs	Constant	TIE Key with maximal values: TIEID(originator=MAX_UINT64, tietype=TIETypeMaxValue, tie_nr=MAX_UINT64, direction=North)

Table 6: all_constants

Authors' Addresses

Tony Przygienda (editor)
Juniper
1137 Innovation Way

Sunnyvale, CA

USA

Email: prz@juniper.net

Alankar Sharma
Comcast
1800 Bishops Gate Blvd
Mount Laurel, NJ 08054
US

Email: Alankar_Sharma@comcast.com

Pascal Thubert
Cisco Systems, Inc
Building D
45 Allee des Ormes - BP1200
MOUGINS - Sophia Antipolis 06254
FRANCE

Phone: +33 497 23 26 34
Email: pthubert@cisco.com

Bruno Rijsman
Individual

Email: brunorijsman@gmail.com

Dmitry Afanasiev
Yandex

Email: fl0w@yandex-team.ru

RIFT WG B. Rijsman
Internet-Draft Individual
Intended status: Standards Track Z. Zhang, Ed.
Expires: August 26, 2021 Y. Wei
ZTE Corporation
S. Ma
Google
X. Liu
Volta Networks
February 22, 2021

A YANG Data Model for Routing in Fat Trees (RIFT)
draft-ietf-rift-yang-02

Abstract

This document defines a YANG data model for the configuration and management of Routing in Fat Trees (RIFT) Protocol.

Table of Contents

1. Introduction

[I-D.ietf-rift-rift] introduces the protocol definition of RIFT. This document defines a YANG data model that can be used to configure and manage the RIFT protocol. The model is based on YANG 1.1 as defined in [RFC7950] and conforms to the Network Management Datastore Architecture (NDMA) as described in [RFC8342]

1.1. Terminology

The terminology for describing YANG data models is found in [RFC6020] and [RFC7950], including:

- o augment
- o container
- o choice
- o data model

o data node

o grouping

o identity

o leaf

o leaf-list

o list

o module

o uses

The following abbreviations are used in this document and the defined model:

RIFT: Routing in Fat Trees [I-D.ietf-rift-rift].

1.2. Conventions Used in This Document

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

1.3. Tree Diagrams

Tree diagrams used in this document follow the notation defined in [RFC8340].

1.4. Prefixes in Data Node Names

In this document, names of data nodes, actions, and other data model objects are often used without a prefix, as long as it is clear from the context in which YANG module each name is defined. Otherwise, names are prefixed using the standard prefix associated with the corresponding YANG module, as shown in Table 1.

Prefix	YANG module	Reference
yang	ietf-yang-types	[RFC6991]
inet	ietf-inet-types	[RFC6991]
rt	ietf-routing	[RFC8349]
if	ietf-interfaces	[RFC8343]
rt-types	ietf-routing-types	[RFC8294]
iana-rt-types	iana-routing-types	[RFC8294]

Table 1

2. Design of the Data Model

2.1. Scope of Model

The model covers RIFT [I-D.ietf-rift-rift].

This model can be used to configure and manage the RIFT protocol. The operational state data and statistics can be retrieved by this model. The subscription and push mechanism defined in [RFC8639] and [RFC8641] can be implemented by the user to subscribe to notifications on the data nodes in this model.

The model contains all the basic configuration parameters to operate the protocol. Depending on the implementation choices, some systems may not allow some of the advanced parameters to be configurable. The occasionally implemented parameters are modeled as optional features in this model. This model can be extended, and it has been structured in a way that such extensions can be conveniently made.

The RIFT YANG module augments the /routing/control-plane-protocols/control-plane-protocol path defined in the ietf-routing module. The ietf-rift model defines a single instance of RIFT. Multiple instances are instantiated as multiple control-plane protocols instances.

2.2. Specification

This model imports and augments ietf-routing YANG model defined in [RFC8349]. Both configuration branch and state branch of [RFC8349] are augmented. The configuration branch covers node base and policy configuration. The container "rift" is the top level container in this data model. The presence of this container is expected to enable RIFT protocol functionality.

The YANG data model defined in this document conforms to the Network Management Datastore Architecture (NMDA) [RFC8342]. The operational state data is combined with the associated configuration data in the same hierarchy [RFC8407].

2.3. Overview

The RIFT YANG module defined in this document has all the common building blocks for the RIFT protocol.

The RIFT YANG module augments the /routing/control-plane-protocols/ control-plane-protocol path defined in the ietf-routing module. The ietf-rift model defines a single instance of RIFT. Multiple instances are instantiated as multiple control-plane protocols instances.

```
module: ietf-rift
  augment /rt:routing/rt:control-plane-protocols
            /rt:control-plane-protocol:
    +--rw rift
       +--rw name?                         string
       +--ro level?                        level
       +--rw system-id                     system-id
       +--rw pod?                          uint32
       +--rw configured-level?             level
       +--rw overload?                     boolean
       +--ro protocol-major-version        uint8
       +--ro protocol-minor-version        uint16
       +--ro hierarchy-indications?        enumeration
       +--rw flood-reduction?              boolean
       +--rw nonce-increasing-interval?    uint16
       +--rw maximum-nonce-delta?          uint8 {nonce-delta-adjust}?
       +--rw rx-lie-multicast-address
       |  +--rw ipv4?   inet:ipv4-address
       |  +--rw ipv6?   inet:ipv6-address
       +--rw tx-lie-multicast-address
       |  +--rw ipv4?   inet:ipv4-address
       |  +--rw ipv6?   inet:ipv6-address
       +--rw lie-tx-port?                  inet:port-number
       +--rw global-link-capabilities
       |  +--rw bfd?                      boolean
       |  +--rw v4-forwarding-capable?    boolean
       +--rw rx-flood-port?                inet:port-number
       +--rw holdtime?
```

```
          |       rt-types:timer-value-seconds16
          +--rw tide-generation-interval?
          |       rt-types:timer-value-seconds16
          +--rw tie-security-key-id?        uint32
          +--rw interface* [name]
          |  +--ro link-id?                         linkid-type
          |  +--rw name                             if:interface-ref
          |  +--rw cost?                            uint32
          |  +--rw address-families
          |  |  +--rw address-family* [address-family]
          |  |     +--rw address-family    iana-rt-types:address-family
          |  +--rw advertised-source-addresses
          |  |  +--rw ipv4?   inet:ipv4-address
          |  |  +--rw ipv6?   inet:ipv6-address
          |  +--ro direction-type?                  enumeration
          |  +--ro was-the-last-lie-accepted?       boolean
          |  +--ro last-lie-reject-reason?          string
          |  +--ro advertised-in-lies
          |  |  +--ro you-are-flood-repeater?          boolean
          |  |  +--ro not-a-ztp-offer?                 boolean
          |  |  +--ro you-are-sending-too-quickly?     boolean
          |  +--rw link-capabilities
          |  |  +--rw bfd?                          boolean
          |  |  +--rw v4-forwarding-capable?        boolean
          |  +--ro state                            enumeration
          |  +--ro number-of-flaps?                 uint32
          |  +--ro last-state-change?               yang:date-and-time
          +--ro miscabled-links*            linkid-type
          +--rw (algorithm-type)?
          |  +--:(spf)
          |  +--:(all-path)
          +--ro hal?                        level
          +--rw instance-label?             uint32 {label-switching}?
          +--ro neighbor* [system-id]
          |  +--ro name?                            string
          |  +--ro level?                           level
          |  +--ro system-id                        system-id
          |  +--ro pod?                             uint32
          |  +--ro protocol-version?                uint16
          |  +--ro protocol-minor-version?          uint16
          |  +--ro sent-offer
          |  |  +--ro level?             level
          |  |  +--ro not-a-ztp-offer?   boolean
          |  +--ro received-offer
          |  |  +--ro level?                          level
          |  |  +--ro not-a-ztp-offer?                boolean
          |  |  +--ro best?                           boolean
          |  |  +--ro removed-from-consideration?     boolean
```

```
     |  |  +--ro removal-reason?                string
     |  +--ro received-source-addresses
     |  |  +--ro ipv4?   inet:ipv4-address
     |  |  +--ro ipv6?   inet:ipv6-address
     |  +--ro link-id-pair* [remote-id]
     |  |  +--ro local-id?    uint32
     |  |  +--ro remote-id    uint32
     |  |  +--ro if-index?    uint32
     |  |  +--ro if-name?     if:interface-ref
     |  +--ro cost?                          uint32
     |  +--ro bandwidth?                     uint32
     |  +--ro flood-reduction?               boolean
     |  +--ro received-link-capabilities
     |  |  +--ro bfd?                      boolean
     |  |  +--ro v4-forwarding-capable?   boolean
     |  +--ro received-in-lies
     |  |  +--ro you-are-flood-repeater?        boolean
     |  |  +--ro not-a-ztp-offer?               boolean
     |  |  +--ro you-are-sending-too-quickly?   boolean
     |  +--ro tx-flood-port?                 inet:port-number
     |  +--ro bfd-up?                        boolean
     |  +--ro outer-security-key-id?         uint8
     +--ro database
        +--ro tie* [direction-type originator tie-type tie-number]
           +--ro direction-type          enumeration
           +--ro originator              system-id
           +--ro tie-type                enumeration
           +--ro tie-number              uint32
           +--ro seq?                    uint64
           +--ro origination-time?       uint32
           +--ro origination-lifetime?   uint32
           +--ro node
           |  +--ro name?              string
           |  +--ro level?             level
           |  +--ro system-id          system-id
           |  +--ro pod?               uint32
           |  +--ro flood-reduction?   boolean
           |  +--ro overload?          boolean
           |  +--ro startup-time?      uint64
           |  +--ro neighbor* [system-id]
           |  |  +--ro name?                              string
           |  |  +--ro level?                             level
           |  |  +--ro system-id                          system-id
           |  |  +--ro pod?                               uint32
           |  |  +--ro link-id-pair* [remote-id]
           |  |  |  +--ro local-id?    uint32
           |  |  |  +--ro remote-id    uint32
           |  |  |  +--ro if-index?    uint32
```

```
                |  |  |  +--ro if-name?     if:interface-ref
                |  |  +--ro cost?                          uint32
                |  |  +--ro bandwidth?                     uint32
                |  |  +--ro flood-reduction?               boolean
                |  |  +--ro received-link-capabilities
                |  |     +--ro bfd?                      boolean
                |  |     +--ro v4-forwarding-capable?   boolean
                |  +--ro miscabled-links*   linkid-type
                +--ro prefix
                |  +--ro prefix?              inet:ip-prefix
                |  +--ro (type)?
                |  |  +--:(prefix)
                |  |  +--:(positive-disaggregation)
                |  |  +--:(negative-disaggregation)
                |  |  +--:(external)
                |  |  +--:(positive-external-disaggregation)
                |  |  +--:(pgp)
                |  +--ro metric?              uint32
                |  +--ro tags*                uint64
                |  +--ro monotonic-clock
                |  |  +--ro prefix-sequence-type
                |  |     +--ro timestamp
                |  |     |       ieee802-1as-timestamp-type
                |  |     +--ro transaction-id?   uint8
                |  +--ro loopback?            boolean
                |  +--ro directly-attached?   boolean
                |  +--ro from-link?           linkid-type
                +--ro key-value
                   +--ro key?     binary
                   +--ro value?   binary

     notifications:
       +---n error-set
          +--ro tie-level-error
          |  +--ro tie* [originator]
          |     +--ro direction-type?         enumeration
          |     +--ro originator              system-id
          |     +--ro tie-type?               enumeration
          |     +--ro tie-number?             uint32
          |     +--ro seq?                    uint64
          |     +--ro origination-time?       uint32
          |     +--ro origination-lifetime?   uint32
          +--ro neighbor-error
             +--ro neighbor* [system-id]
                +--ro name?                         string
                +--ro level?                        level
                +--ro system-id                     system-id
                +--ro pod?                          uint32
```

```
                +--ro protocol-version?              uint16
                +--ro protocol-minor-version?        uint16
                +--ro sent-offer
                |  +--ro level?             level
                |  +--ro not-a-ztp-offer?   boolean
                +--ro received-offer
                |  +--ro level?                          level
                |  +--ro not-a-ztp-offer?                boolean
                |  +--ro best?                           boolean
                |  +--ro removed-from-consideration?     boolean
                |  +--ro removal-reason?                 string
                +--ro received-source-addresses
                |  +--ro ipv4?   inet:ipv4-address
                |  +--ro ipv6?   inet:ipv6-address
                +--ro link-id-pair* [remote-id]
                |  +--ro local-id?    uint32
                |  +--ro remote-id    uint32
                |  +--ro if-index?    uint32
                |  +--ro if-name?     if:interface-ref
                +--ro cost?                          uint32
                +--ro bandwidth?                     uint32
                +--ro flood-reduction?               boolean
                +--ro received-link-capabilities
                |  +--ro bfd?                      boolean
                |  +--ro v4-forwarding-capable?    boolean
                +--ro received-in-lies
                |  +--ro you-are-flood-repeater?         boolean
                |  +--ro not-a-ztp-offer?                boolean
                |  +--ro you-are-sending-too-quickly?    boolean
                +--ro tx-flood-port?                 inet:port-number
                +--ro bfd-up?                        boolean
                +--ro outer-security-key-id?         uint8
```

2.4. RIFT configuration

The configuration data nodes cover node configuration attributes. RIFT configurations require node base information configurations. Some features can be used to enhance protocol, such as BFD, flooding-reducing, community attribute.

2.5. RIFT State

The state data nodes include node, neighbor, database and kv-store information.

2.6. Notifications

Unexpected TIE and neighbor's layer error should be notified.

3. RIFT YANG model

This module references [I-D.ietf-rift-rift], [RFC5881], [RFC6991], [RFC8177], [RFC8294], [RFC8343], [RFC8349], [RFC8505].

```
 <CODE BEGINS> file "ietf-rift@2021-02-20.yang"
 module ietf-rift {

   yang-version 1.1;

   namespace "urn:ietf:params:xml:ns:yang:ietf-rift";
   prefix rift;

   import ietf-inet-types {
     prefix "inet";
     reference "RFC 6991: Common YANG Data Types";
   }

   import ietf-yang-types {
     prefix "yang";
     reference "RFC 6991: Common YANG Data Types";
   }

   import ietf-routing {
     prefix "rt";
     reference
       "RFC 8349: A YANG Data Model for Routing Management
        (NMDA Version)";
   }

   import ietf-interfaces {
     prefix "if";
     reference
       "RFC 8343: A YANG Data Model for Interface Management";
   }

   import ietf-routing-types {
     prefix "rt-types";
     reference
       "RFC 8294: Common YANG Data Types for the Routing Area";
   }

   import iana-routing-types {
     prefix "iana-rt-types";
```

```
     reference
       "RFC 8294: Common YANG Data Types for the Routing Area";
   }

   organization
     "IETF RIFT (Routing In Fat Trees) Working Group";

   contact
     "WG Web:   <http://tools.ietf.org/wg/rift/>
      WG List:  <mailto:rift@ietf.org>

      Editor:   Zheng Zhang
                <mailto:zhang.zheng@zte.com.cn>

      Editor:   Yuehua Wei
                <mailto:wei.yuehua@zte.com.cn>

      Editor:   Shaowen Ma
                <mailto:mashaowen@gmail.com>

      Editor:   Xufeng Liu
                <mailto:xufeng.liu.ietf@gmail.com>";

   // RFC Ed.: replace XXXX with actual RFC number and remove
   // this note

   description
     "The module defines the YANG definitions for Routing in Fat
      Trees (RIFT).

      Copyright (c) 2020 IETF Trust and the persons identified as
      authors of the code.  All rights reserved.

      Redistribution and use in source and binary forms, with or
      without modification, is permitted pursuant to, and subject
      to the license terms contained in, the Simplified BSD
      License set forth in Section 4.c of the IETF Trust's Legal
      Provisions Relating to IETF Documents
      (https://trustee.ietf.org/license-info).

      This version of this YANG module is part of RFC XXXX
      (https://www.rfc-editor.org/info/rfcXXXX); see the RFC
      itself for full legal notices.

      The key words 'MUST', 'MUST NOT', 'REQUIRED', 'SHALL',
      'SHALL NOT', 'SHOULD', 'SHOULD NOT', 'RECOMMENDED',
      'NOT RECOMMENDED', 'MAY', and 'OPTIONAL' in this document
      are to be interpreted as described in BCP 14 (RFC 2119)
```

```
      (RFC 8174) when, and only when, they appear in all
      capitals, as shown here.";

   revision 2021-02-20 {
     description "Initial revision.";
     reference "RFC XXXX: A YANG Data Model for RIFT.";
   }

   /*
    * Features
    */

   feature nonce-delta-adjust {
     description
       "Support weak nonce delta adjusting which is used in security
        in section 4.4.";
   }

   feature label-switching {
     description
       "Support label switching for instance distinguishing in
        section 4.3.7.";
   }

   typedef system-id {
     type string {
       pattern
       '[0-9A-Fa-f]{4}\.[0-9A-Fa-f]{4}\.[0-9A-Fa-f]{4}\.[0-9A-Fa-f]{4}';
     }
     description
       "This type defines RIFT system id using pattern,
        the system id looks like: 0143.0438.0100.AeF0";
   }

   typedef level {
     type uint8 {
       range "0 .. 24";
     }
     default "0";
     description "The value of node level. The max value is 24.";
   }

   typedef linkid-type {
     type uint32;
     description "This type defines the link id of an interface.";
   }

   typedef ieee802-1as-timestamp-type {
```

```
     type uint64;
     description
       "Timestamp per 802.1AS. It is advertised with prefix to
        achieve mobility as described in section 4.3.3.";
   }

   /*
    * Identity
    */
   identity rift {
     base rt:routing-protocol;
     description "Identity for the RIFT routing protocol.";
   }

   /*
    * Groupings
    */

    grouping base-node-info {
     leaf name {
       type string;
       description
         "The name of this node. It won't be used as the key of node,
          just used for description.";
     }
     leaf level {
       type level;
       config false;
       description "The level of this node.";
     }
     leaf system-id {
       type system-id;
       mandatory true;
       description
         "Each node is identified via a system-id which is 64 bits
          wide.";
     }
     leaf pod {
       type uint32;
       description
         "Point of Delivery. The self-contained vertical slice of a
          Clos or Fat Tree network containing normally only level 0
          and level 1 nodes. It communicates with nodes in other PoDs
          via the spine. We number PoDs to distinguish them and use
          PoD #0 to denote 'undefined' PoD.";
     }
     description "The base information of a node.";
   } // base-node-info
```

```
   grouping node-flag {
     leaf overload {
       type boolean;
       description "If the overload bit in TIEs can be set.";
     }
     description "The node flag information.";
   }

   grouping link-capabilities {
     leaf bfd {
       type boolean;
       description
         "If this value is set to true, it means that
          BFD [RFC5881] function is enabled on the neighbor.";
     }
     leaf v4-forwarding-capable {
       type boolean;
       description
         "If this value is set to true, it means that
          the neighbor supports v4 forwarding.";
     }
     description "The features of neighbor.";
   } // link-capabilities

   grouping addresses {
     leaf ipv4 {
       type inet:ipv4-address;
       description "IPv4 address to be used.";
     }
     leaf ipv6 {
       type inet:ipv6-address;
       description "IPv6 address to be used.";
     }
     description "IPv4 or IPv6 address to be used.";
   }

   grouping lie-elements{
     leaf you-are-flood-repeater {
       type boolean;
       description
         "If the neighbor on this link is flooding repeater
          described in section 4.2.3.9. When this value is set
          to true, the value can be carried in exchanged
          packet.";
     }
     leaf not-a-ztp-offer {
       type boolean;
       description
```

```
         "As described in section 4.2.7. When this value is
          set to true, the flag can be carried in the LIE
          packet. When the value received in the LIE from
          neighbor, it indicates the level on the LIE MUST
          NOT be used to derive a ZTP level by the receiving
          node.";
     }
     leaf you-are-sending-too-quickly {
       type boolean;
       description
         "Can be optionally set to indicate to neighbor that
          packet losses are seen on reception based on packet
          numbers or the rate is too high. The receiver SHOULD
          temporarily slow down flooding rates. When this value
          is set to true, the flag can be carried in packet.";
     }
     description "The elements set in the LIEs.";
   } // lie-elements

   grouping link-id-pair {
     leaf local-id {
       type uint32;
       description "The local-id of link connect to this neighbor.";
     }
     leaf remote-id {
       type uint32;
       description "The remote-id to reach this neighbor.";
     }
     leaf if-index {
       type uint32;
       description "The local index of this interface.";
     }
     leaf if-name {
       type if:interface-ref;
       description "The name of this interface.";
     }
     description
       "A pair of local and remote link IDs to identify a link between
        two nodes.";
   } // link-id-pair

   grouping neighbor-node {
     list link-id-pair {
       key "remote-id";
       uses link-id-pair;
       description
         "The Multiple parallel links to this neighbor.";
     }
```

```
      leaf cost {
        type uint32;
        description "The cost value advertised by the neighbor.";
      }
      leaf bandwidth {
        type uint32;
        description
          "Total bits bandwith to neighbor, this will be
           normally sum of the bandwidths of all the
           parallel links.";
      }
      leaf flood-reduction {
        type boolean;
        description
          "If this neighbor enables the flood reduction function.";
      }
      container received-link-capabilities {
        uses link-capabilities;
        description
          "The link capabilities advertised by the neighbor.";
      }
      description "The neighbor information indicated in node TIE.";
    } // neighbor-node

    grouping neighbor {
      leaf protocol-version {
        type uint16;
        description
          "Represents the protocol encoding schema version of
           this neighbor.";
      }
      leaf protocol-minor-version {
        type uint16;
        description
          "Represents the minor protocol encoding schema
           version of this neighbor.";
      }
      container sent-offer {
        leaf level {
          type level;
          description "The level value.";
        }
        leaf not-a-ztp-offer {
          type boolean;
          description "If the neighbor needs to be offer a level.";
        }
        description
          "The level sent to the neighbor in case the neighbor
```

```
          needs to be offered.";
     }
     container received-offer {
       leaf level {
         type level;
         description "The level value.";
       }
       leaf not-a-ztp-offer {
         type boolean;
         description
           "If this interface needs to be offered a level.";
       }
       leaf best {
         type boolean;
         description
           "If level is the best level received from all
            the neighbors.";
       }
       leaf removed-from-consideration {
         type boolean;
         description
           "If the level value is considered to be used.
            If the value is not considered to be used,
            this value is set to 'TRUE'.";
       }
       leaf removal-reason {
         type string;
         description
           "The reason why this value is not considered to
            be used.";
       }
       description
         "The level offered to the interface from the neighbor.
          And if the level value is considered to be used.";
     }
     container received-source-addresses {
       uses addresses;
       description
         "The source address of LIE and TIE packets from
          the neighbor.";
     } // received-offer
     uses neighbor-node;
     container received-in-lies {
       uses lie-elements;
       description "The attributes received from this neighbor.";
     }
     leaf tx-flood-port {
       type inet:port-number;
```

```
        default "915";
        description
          "The UDP port which is used by the neighbor to flood TIEs.";
      }
      leaf bfd-up {
        type boolean;
        description
          "Indication whether the link is protected by established
           BFD session.";
      }
      leaf outer-security-key-id {
        type uint8;
        description
          "As described in section 4.4.3, the received security
           key id from the neighbor.";
      }
      description "The neighbor information.";
    } // neighbor

    grouping direction-type {
      leaf direction-type {
        type enumeration {
          enum illegal {
            description "Illegal direction.";
          }
          enum south {
            description "A link to a node one level down.";
          }
          enum north {
            description "A link to a node one level up.";
          }
          enum east-west {
            description "A link to a node in the same level.";
          }
          enum max {
            description "The max value of direction.";
          }
        }
        config false;
        description "The type of a link.";
      }
      description "The type of a link.";
    } // direction-type

    grouping tie-header {
      uses direction-type;
      leaf originator {
        type system-id;
```

```
        description "The originator’s system-id of this TIE.";
      }

      leaf tie-type {
        type enumeration {
          enum "node" {
            description "The node TIE.";
          }
          enum "prefix" {
            description "The prefix TIE.";
          }
          enum "positive-disaggregation-prefix" {
            description "The positive disaggregation prefix TIE.";
          }
          enum "negative-disaggregation-prefix" {
            description "The negative disaggregation prefix TIE.";
          }
          enum "pgp-prefix" {
            description "The policy guide prefix TIE.";
          }
          enum "key-value" {
            description "The key value TIE.";
          }
          enum "external-prefix" {
            description "The external prefix TIE.";
          }
          enum "positive-external-disaggregation-prefix" {
            description
              "The positive external disaggregation prefix TIE.";
          }
        }
        description "The types of TIE.";
      }

      leaf tie-number {
        type uint32;
        description "The number of this TIE";
      }

      leaf seq {
        type uint64;
        description
          "As described in section 4.2.3.1, the sequence number
           of a TIE.";
      }
      leaf origination-time {
        type uint32;
        description
```

```
          "Absolute timestamp when the TIE was generated. This can
          be used on fabrics with synchronized clock to prevent
          lifetime modification attacks.";
      }
      leaf origination-lifetime {
        type uint32;
        description
          "Original lifetime when the TIE was generated.
           This can be used on fabrics with synchronized clock to
           prevent lifetime modification attacks.";
      }

      description
        "TIE is the acronym for 'Topology Information Element'.
         TIEs are exchanged between RIFT nodes to describe parts
         of a network such as links and address prefixes.
         This is the TIE header information.";
    } // tie-header

    /*
     * Data nodes
     */
    augment "/rt:routing/rt:control-plane-protocols"
            + "/rt:control-plane-protocol" {
      when "derived-from-or-self(rt:type, 'rift:rift')" {
        description
          "This augment is only valid when routing protocol
           instance type is 'RIFT'.";
      }
      description "RIFT ( Routing in Fat Trees ) YANG model.";

      container rift {
        description "RIFT configuration and state data.";

        uses base-node-info;
        leaf configured-level {
          type level;
          description
            "The configured level value of this node.";
        }
        uses node-flag;

        leaf protocol-major-version {
          type uint8;
          config false;
          mandatory true;
          description
            "Represents protocol encoding schema major version.";
```

```
       }
       leaf protocol-minor-version {
         type uint16;
         config false;
         mandatory true;
         description
           "Represents protocol encoding schema minor version.";
       }
       leaf hierarchy-indications {
         type enumeration {
           enum "leaf-only" {
             description
               "The node will never leave the
                'bottom of the hierarchy'.";
           }
           enum "leaf-only-and-leaf-2-leaf-procedures" {
             description "This means leaf to leaf.";
           }
           enum "top-of-fabric" {
             description "The node is 'top of fabric'.";
           }
         }
         config false;
         description "The hierarchy indications of this node.";
       }

       leaf flood-reduction {
         type boolean;
         description
           "If the node supports flood reduction function defined in
            section 4.2.3.8. If this value is set to 'FALSE', it
            means that the flood reduction function is disabled.";
       }

       leaf nonce-increasing-interval {
         type uint16;
         units seconds;
         description
           "The configurable nonce increasing interval.";
       }
       leaf maximum-nonce-delta {
         if-feature nonce-delta-adjust;
         type uint8 {
           range "1..5";
         }
         description
           "The configurable valid nonce delta value used for
            security. It is used as vulnerability window defined
```

```
            in section 4.4.7.
            If the nonces in received packet exceeds the range
            indicated by this value, the packet MUST be discarded.";
       }
       container rx-lie-multicast-address {
         uses addresses;
         description
           "The configurable LIE receiving IPv4/IPv6 multicast
            address. '224.0.0.120' is default address value.
            Different multicast addresses can be used for receiving
            and sending.";
       }
       container tx-lie-multicast-address {
         uses addresses;
         description
           "The configurable LIE sending IPv4/IPv6 multicast
            address. 'FF02::A1F7' is default address value.
            Different multicast addresses can be used for receiving
            and sending.";
       }
       leaf lie-tx-port {
         type inet:port-number;
         description
           "The UDP port of LIE packet sending. The default port
            number is 914. The value can be set to other value
            associated with different RIFT instance.";
       }

       container global-link-capabilities {
         uses link-capabilities;
         description
           "The node default link capabilities. It can be overwrite
            by the configuration underneath interface and neighbor.";
       }

       leaf rx-flood-port {
         type inet:port-number;
         default "915";
         description
           "The UDP port which can be used to receive flooded
            TIEs. The default port number is 915. The value can
            be set to other value associated with different
            RIFT instance.";
       }
       leaf holdtime {
         type rt-types:timer-value-seconds16;
         units seconds;
         default "3";
```

```
          description "The holding time of LIE.";
        }
        leaf tide-generation-interval {
          type rt-types:timer-value-seconds16;
          units seconds;
          default "5";
          description "The TIDE generation interval.";
        }

        leaf tie-security-key-id {
          type uint32;
          description
            "As described in section 4.4.3, this value implies key
             type and algorithm.  Value 0 means that no valid
             fingerprint was computed. This key ID scope is local
             to the nodes on both ends of the adjacency.";
        }

        list interface {
          key "name";
          leaf link-id {
            type linkid-type;
            config false;
            description "The local id of this interface.";
          }
          leaf name {
            type if:interface-ref;
            description "The interface's name.";
          }
          leaf cost {
            type uint32;
            description
              "The cost from this interface to the neighbor.";
          }
          container address-families {
            description
              "Containing address families on the interface.";
            list address-family {
              key address-family;
              description
                "A list of address families enabled on the
                 interface.";
              leaf address-family {
                type iana-rt-types:address-family;
                description
                  "Indication which address families are up on the
                   interface.";
              }
```

```
          }
        }
        container advertised-source-addresses {
          uses addresses;
          description
            "The address used in the advertised LIE and TIE
             packets.";
        }
        uses direction-type;

        leaf was-the-last-lie-accepted {
          type boolean;
          config false;
          description
            "If the most recently received LIE was accepted or
             rejected. If the LIE was rejected, the neighbor error
             notifications should be used to find the reason.";
        }
        leaf last-lie-reject-reason {
          type string;
          config false;
          description
            "Description for the reject reason of the last LIE.";
        }
        container advertised-in-lies {
          config false;
          uses lie-elements;
          description
            "The attributes advertised in the LIEs from
             this interface.";
        }
        container link-capabilities {
          uses link-capabilities;
          description
            "The interface's link capabilities.";
        }
        leaf state {
          type enumeration {
            enum "OneWay" {
              description "The initial state of neighbor.";
            }
            enum "TwoWay" {
              description "This means leaf to leaf.";
            }
            enum "ThreeWay" {
              description "The node is 'top of fabric'.";
            }
            enum "Multiple-Neighbors-Wait" {
```

```
                description "The node is 'top of fabric'.";
              }
            }
            config false;
            mandatory true;
            description "The hierarchy indications of this node.";
          }
          leaf number-of-flaps {
            type uint32;
            config false;
            description
              "The number of interface state flaps.";
          }
          leaf last-state-change {
            type yang:date-and-time;
            config false;
            description "Time duration in the current state.";
          }

          description "The interface information on this node.";
        } // list interface

        leaf-list miscabled-links {
          type linkid-type;
          config false;
          description "List of miscabled links.";
        }

        choice algorithm-type {
          case spf {
            description "The algorithm is SPF.";
          }
          case all-path {
            description "The algorithm is all-path.";
          }
          description "The possible algorithm types.";
        }

        leaf hal {
          type level;
          config false;
          description
            "The highest defined level value seen from all valid
             level offers received.";
        }

        leaf instance-label {
          if-feature label-switching;
```

```
          type uint32;
          description
            "As per section 4.3.7, a locally significant, downstream
             assigned, interface specific label may be advertised in
             its LIEs. This value can be used to distinguish among
             multiple RIFT instances.";
        }

        list neighbor {
          key "system-id";
          config false;
          uses base-node-info;
          uses neighbor;
          description "The neighbor's information.";
        }

        container database {
          config false;
          list tie {
            key "direction-type originator tie-type tie-number";
            description
              "A list of TIEs (Topology Information Elements).";
            uses tie-header;

            container node {
              uses base-node-info;
              leaf flood-reduction {
                type boolean;
                description
                  "If the node enable the flood reduction function.";
              }
              uses node-flag;
              leaf startup-time {
                type uint64;
                description "Startup time of the node.";
              }
              list neighbor {
                key "system-id";
                uses base-node-info;
                uses neighbor-node;
                description "The node TIE information of a neighbor.";
              }

              leaf-list miscabled-links {
                type linkid-type;
                config false;
                  description "List of miscabled links.";
              }
```

```
             description "The node element information in this TIE.";
           } // node

           container prefix {
             leaf prefix {
               type inet:ip-prefix;
               description "The prefix information.";
             }

             choice type {
               case prefix {
                 description "It is the prefixes TIE element.";
               }
               case positive-disaggregation {
                 description
                   "It is the positive disaggregation prefixes
                    TIE element.";
               }
               case negative-disaggregation {
                 description
                   "It is the negative disaggregation prefixes
                    TIE element.";
               }
               case external {
                 description
                   "It is the external prefixes TIE element.";
               }
               case positive-external-disaggregation {
                 description
                   "It is the positive external disaggregation
                    prefixes TIE element.";
               }
               case pgp {
                 description
                   "It is the policy guide prefixes TIE element.";
               }
               description "The type of prefix TIE.";
             }

             leaf metric {
               type uint32;
               description "The metric of this prefix.";
             }
             leaf-list tags {
               type uint64;
               description "The tags of this prefix.";
             }
             container monotonic-clock {
```

```
               container prefix-sequence-type {
                 leaf timestamp {
                   type ieee802-1as-timestamp-type;
                   mandatory true;
                   description
                     "The timestamp per 802.1AS can be advertised
                      with the desired prefix North TIEs.";
                 }
                 leaf transaction-id {
                   type uint8;
                   description
                     "As per RFC 8505, a sequence number called a
                      Transaction ID (TID) with a prefix can be
                      advertised.";
                 }
                 description
                   "As described in section 4.3.3, the prefix
                    sequence attribute which can be advertised
                    for mobility.";
               }
               description
                 "The monotonic clock for mobile addresses.";
             }
             leaf loopback {
               type boolean;
               description
                 "Indicates if the interface is a node loopback.
                  According to section 4.3.10, the node's loopback
                  address can be injected into North and South
                  Prefix TIEs for node reachability.";
             }
             leaf directly-attached {
               type boolean;
               description
                 "Indicates that the prefix is directly attached,
                  i.e. should be routed to even if the node is
                  in overload.";
             }
             leaf from-link {
               type linkid-type;
               description
                 "In case of locally originated prefixes,
                  i.e. interface addresses this can describe which
                  link the address belongs to.";
             }
             description "The detail information of a prefix.";
           } // prefix
```

```
          container key-value {
            leaf key {
              type binary;
              description "The type of key value combination.";
            }
            leaf value {
              type binary;
              description "The value of key value combination.";
            }
            description
              "The information used to distinguish a Key/Value
               pair. When the type of kv is set to 'node',
               node-element is making sense. When the type of
               kv is set to other values except 'node',
               prefix-info is making sense.";
          } // kv-store
        } // ties
        description "The TIEs information in database.";
      }// container database
    }//rift
  }//augment

  /*
   * Notifications
   */
  notification error-set {
    description "The errors notification of RIFT.";
    container tie-level-error {
      list tie {
        key "originator";
        uses tie-header;
        description "The level is undefined in the LIEs.";
      }
      description "The TIE errors set.";
    }
    container neighbor-error {
      list neighbor {
        key "system-id";
        uses base-node-info;
        uses neighbor;
        description "The information of a neighbor.";
      }
      description "The neighbor errors set.";
    }
  }
}
<CODE ENDS>
```

4. Security Considerations

The YANG module specified in this document defines a schema for data that is designed to be accessed via network management protocols such as NETCONF [RFC6241] or RESTCONF [RFC8040]. The lowest NETCONF layer is the secure transport layer, and the mandatory-to-implement secure transport is Secure Shell (SSH) [RFC6242]. The lowest RESTCONF layer is HTTPS, and the mandatory-to-implement secure transport is TLS [RFC8446].

The NETCONF access control model [RFC8341] provides the means to restrict access for particular NETCONF or RESTCONF users to a preconfigured subset of all available NETCONF or RESTCONF protocol operations and content.

There are a number of data nodes defined in this YANG module that are writable/creatable/deletable (i.e., config true, which is the default). These data nodes may be considered sensitive or vulnerable in some network environments. Write operations (e.g., edit-config) to these data nodes without proper protection can have a negative effect on network operations. Writable data node represent configuration of each instance, node, interface, etc. These correspond to the following schema nodes:

- /rift

- /rift/node/

Modifying the configuration may cause all the RIFT neighborship to be rebuilt. For example, the configuration changing of level or systemid, will lead to all the neighbor connections of this node rebuilt. The incorrect modification of authentication, except for the neighbor connection broken, will lead to the permanent connection broken. The modification of interface, will lead to the neighbor state changing. In general, unauthorized modification of most RIFT configurations will pose there own set of security risks and the "Security Considerations" in the respective reference RFCs should be consulted.

Some of the readable data nodes in this YANG module may be considered sensitive or vulnerable in some network environments. It is thus important to control read access (e.g., via get, get-config, or notification) to these data nodes. These are the subtrees and data nodes and their sensitivity/vulnerability:

- /rift

- /rift/interface

- /rift/database

- /rift/kv-store

The exposure of the database will expose the detailed topology of the network. Network operators may consider their topologies to be sensitive confidential data.

For RIFT authentication, configuration is supported via the specification of key-chains [RFC8177] or the direct specification of key and authentication algorithm. Hence, authentication configuration inherits the security considerations of [RFC8177]. This includes the considerations with respect to the local storage and handling of authentication keys.

5. IANA Considerations

RFC Ed.: Please replace all occurrences of 'XXXX' with the actual RFC number (and remove this note).

This document registers a URI in the IETF XML registry [RFC3688]. Following the format in [RFC3688], the following registration is requested to be made:

URI: urn:ietf:params:xml:ns:yang:ietf-rift

Registrant Contact: The IESG

XML: N/A, the requested URI is an XML namespace.

This document also requests one new YANG module name in the YANG Module Names registry [RFC6020] with the following suggestion:

name: ietf-rift

namespace: urn:ietf:params:xml:ns:yang:ietf-rift

prefix: rift

reference: RFC XXXX

6. Acknowledgement

The authors would like to thank Tony Przygienda, Benchong Xu (xu.benchong@zte.com.cn), for their review, valuable comments and suggestions.

7. References

7.1. Normative References

[I-D.ietf-rift-rift]
Przygienda, T., Sharma, A., Thubert, P., Rijsman, B., and D. Afanasiev, "RIFT: Routing in Fat Trees", draft-ietf-rift-rift-12 (work in progress), May 2020.

[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>.

[RFC5881] Katz, D. and D. Ward, "Bidirectional Forwarding Detection (BFD) for IPv4 and IPv6 (Single Hop)", RFC 5881, DOI 10.17487/RFC5881, June 2010, <https://www.rfc-editor.org/info/rfc5881>.

[RFC6020] Bjorklund, M., Ed., "YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)", RFC 6020, DOI 10.17487/RFC6020, October 2010, <https://www.rfc-editor.org/info/rfc6020>.

[RFC6241] Enns, R., Ed., Bjorklund, M., Ed., Schoenwaelder, J., Ed., and A. Bierman, Ed., "Network Configuration Protocol (NETCONF)", RFC 6241, DOI 10.17487/RFC6241, June 2011, <https://www.rfc-editor.org/info/rfc6241>.

[RFC6242] Wasserman, M., "Using the NETCONF Protocol over Secure Shell (SSH)", RFC 6242, DOI 10.17487/RFC6242, June 2011, <https://www.rfc-editor.org/info/rfc6242>.

[RFC6991] Schoenwaelder, J., Ed., "Common YANG Data Types", RFC 6991, DOI 10.17487/RFC6991, July 2013, <https://www.rfc-editor.org/info/rfc6991>.

[RFC7950] Bjorklund, M., Ed., "The YANG 1.1 Data Modeling Language", RFC 7950, DOI 10.17487/RFC7950, August 2016, <https://www.rfc-editor.org/info/rfc7950>.

[RFC7951] Lhotka, L., "JSON Encoding of Data Modeled with YANG", RFC 7951, DOI 10.17487/RFC7951, August 2016, <https://www.rfc-editor.org/info/rfc7951>.

[RFC8040] Bierman, A., Bjorklund, M., and K. Watsen, "RESTCONF Protocol", RFC 8040, DOI 10.17487/RFC8040, January 2017, <https://www.rfc-editor.org/info/rfc8040>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC8177] Lindem, A., Ed., Qu, Y., Yeung, D., Chen, I., and J. Zhang, "YANG Data Model for Key Chains", RFC 8177, DOI 10.17487/RFC8177, June 2017, <https://www.rfc-editor.org/info/rfc8177>.

[RFC8294] Liu, X., Qu, Y., Lindem, A., Hopps, C., and L. Berger, "Common YANG Data Types for the Routing Area", RFC 8294, DOI 10.17487/RFC8294, December 2017, <https://www.rfc-editor.org/info/rfc8294>.

[RFC8340] Bjorklund, M. and L. Berger, Ed., "YANG Tree Diagrams", BCP 215, RFC 8340, DOI 10.17487/RFC8340, March 2018, <https://www.rfc-editor.org/info/rfc8340>.

[RFC8341] Bierman, A. and M. Bjorklund, "Network Configuration Access Control Model", STD 91, RFC 8341, DOI 10.17487/RFC8341, March 2018, <https://www.rfc-editor.org/info/rfc8341>.

[RFC8342] Bjorklund, M., Schoenwaelder, J., Shafer, P., Watsen, K., and R. Wilton, "Network Management Datastore Architecture (NMDA)", RFC 8342, DOI 10.17487/RFC8342, March 2018, <https://www.rfc-editor.org/info/rfc8342>.

[RFC8343] Bjorklund, M., "A YANG Data Model for Interface Management", RFC 8343, DOI 10.17487/RFC8343, March 2018, <https://www.rfc-editor.org/info/rfc8343>.

[RFC8349] Lhotka, L., Lindem, A., and Y. Qu, "A YANG Data Model for Routing Management (NMDA Version)", RFC 8349, DOI 10.17487/RFC8349, March 2018, <https://www.rfc-editor.org/info/rfc8349>.

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/info/rfc8446>.

[RFC8505] Thubert, P., Ed., Nordmark, E., Chakrabarti, S., and C. Perkins, "Registration Extensions for IPv6 over Low-Power Wireless Personal Area Network (6LoWPAN) Neighbor Discovery", RFC 8505, DOI 10.17487/RFC8505, November 2018, <https://www.rfc-editor.org/info/rfc8505>.

7.2. Informative References

[RFC3688] Mealling, M., "The IETF XML Registry", BCP 81, RFC 3688, DOI 10.17487/RFC3688, January 2004, <https://www.rfc-editor.org/info/rfc3688>.

[RFC8407] Bierman, A., "Guidelines for Authors and Reviewers of Documents Containing YANG Data Models", BCP 216, RFC 8407, DOI 10.17487/RFC8407, October 2018, <https://www.rfc-editor.org/info/rfc8407>.

[RFC8639] Voit, E., Clemm, A., Gonzalez Prieto, A., Nilsen-Nygaard, E., and A. Tripathy, "Subscription to YANG Notifications", RFC 8639, DOI 10.17487/RFC8639, September 2019, <https://www.rfc-editor.org/info/rfc8639>.

[RFC8641] Clemm, A. and E. Voit, "Subscription to YANG Notifications for Datastore Updates", RFC 8641, DOI 10.17487/RFC8641, September 2019, <https://www.rfc-editor.org/info/rfc8641>.

Authors' Addresses

Bruno Rijsman
Individual

Email: brunorijsman@gmail.com

Zheng Zhang (editor)
ZTE Corporation

Email: zhang.zheng@zte.com.cn

Yuehua Wei
ZTE Corporation

Email: wei.yuehua@zte.com.cn

Shaowen Ma
Google

Email: mashaowen@gmail.com

Xufeng Liu
Volta Networks

Email: xufeng.liu.ietf@gmail.com

RIFT   Z. Zhang
Internet-Draft   J. Tantsura
Intended status: Standards Track   J. Head
Expires: 26 August 2021   Juniper Networks
D. Fedyk
Individual
22 February 2021

SRIFT: Segment Routing in Fat Trees
draft-zzhang-rift-sr-03

Abstract

This document specifies signaling procedures for Segment Routing in RIFT. Each node's loopback address, Segment Routing Global Block (SRGB) and Node Segment Identifier (Node-SID), which are typically assigned by a configuration management system and distibuted by routing protocols, are distributed southbound from the Top Of Fabric (TOF) nodes via RIFT's Key-Value distribution mechanism, so that each node can compute how to reach a segment represented by the active SID in a packet. An SR controller signals SR policies to ingress nodes so that they can send packets with a desired segment list to steer traffic.

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 RFC2119.

Status of This Memo

Table of Contents

1. Introduction

Before we discuss the SR procedures for RIFT, let us first review how SR works with OSPF [RFC8665] and IS-IS [RFC8667].

Each node is provisioned with a loopback address as well as SRGB and Node-SID values. The loopback address and Node-SID are centrally coordinated and are unique per-node within the SR network. These values are then communicated to each node out-of-band and stored as configuration information. Communication could be done via primitive pen and paper or via modern signaling (Netconf/YANG) from a configuration management system.

SRGB information represents the label range of the "global" labels that can be allocated on a particular node for SR. SRGB could have more than one contiguous range of labeks allocated to it. It is comprised of the first available label value and the total number of available labels per range. While in modern networks it is common for each node to have identical SRGB values so that a Node-SID will correspond to the same label on each node, this is not required as to allow for flexible label allocation. In either scenario, SRGB is part of each node’s configuration. In today’s networks, it is likely pushed to nodes by a configuration management system.

Each node then signals its SRGB and Node-SID to the other nodes. A Node-SID is an index value assigned to a node (say node X), and another node (say node Y) uses the Node-SID to derive (from Y’s SRGB) the label to use when sending traffic to node X.

Consider the following example illustrating Node A’s computed IP route and label values.

```
                 B
               *   *
             *       *
           *           *
         A               D
           *           *
             *       *
               *   *
                 C
```

Node Name	Loopback	Node SID	SRGB Label Base	SRGB Label Range
A	10.1.1.1	1	100	50
B	10.1.1.2	2	100	50
C	10.1.1.3	3	200	50
D	10.1.1.4	4	100	50

```
Destination    Next Hop
-----------    --------
10.1.1.1       local
10.1.1.2       if_ab
10.1.1.3       if_ac
10.1.1.4       if_ab, if_ac

Label           Next Hop
-----           --------
100 (La_a)      pop and look up next header
101 (Lb_a)      swap to 101 (Lb_b), via if_ab
102 (Lc_a)      swap to 202 (Lc_c), via if_ac
103 (Ld_a)      swap to 103 (Ld_b), via if_ab
                swap to 203 (Ld_c), via if_ac
```

The specific notation Lb_a refers to the label derived for node B, using B's Node-SID as index into A's SRGB. Similarly, Ld_c refers to the label derived for Node D, using D's Node-SID as index into C's SRGB.

Node A computes the route to Node D's loopback address. The next hops are Node B (via if_ab) and Node C (via if_ac). Node A uses Node D's Node-SID (which was advertised along with the loopback address) to index into its local SRGB to obtain a label value of 103 (Ld_a). Furthermore, Node A also uses Node D's Node-SID to derive label values for Node B and Node C, 103 (Ld_b) and 203 (Ld_c) respectively, using D's Node-SIDs as index into B and C' SRGBs respectively. Notice that Node C's SRGB is different from the other nodes. Node A can now program its label forwarding state with (Ld_a --> (via if_ab swap to Ld_b, via if_ac swap to Ld_c)).

Similarly, Node B computes the route to Node D's loopback address, but this time finds that the next hop is Node D itself (via if_bd). Node B will also use Node D's Node-SID (again, advertised with the loopback address) to index into its local SRGB and obtain a label value of 103 (Ld_b) and index into Node D's SRGB and obtain a label value of 103 (Ld_d). The label forwarding state can be programmed with (Ld_b --> via if_bd swap to Ld_d). Finally, Node D programs its label forwarding state with (Ld_d -> pop and lookup next header).

2. SR in RIFT (SRIFT)

In referring to the previous section, it is clear that each RIFT node participating in a SR domain requires the following information:

* SRGB values of all adjacent nodes

* Node-SID values of all nodes participating in the routing domain

* Loopback addresses or System IDs of all other nodes

In OSPF and IS-IS, each node's SR information is simply flooded. With RIFT, Node TIEs could be used to flood SR information, but each node would have to learn its own SR information first. With RIFT's Key-Value mechanism, KV-TIEs can be used for TOF nodes to flood all nodes' SR information that it learns from an SR controller, therefore accommodating both provisioning and signalling of SR. The non-TOF nodes do not need any SR related provisioning, which goes very well with RIFT's ZTP concept.

ToF nodes in an SR domain MUST populate KV South TIEs with the minimum required SR information for each node. Specifically SRGB Label Base, SRGB Label Range, Node-SID, RIFT System ID, and Loopback Address. While the Loopback Address must be included, it MAY be set to an empty value in cases if loopbacks are not configured for nodes.

Traffic forwarding in an SR network is typically done in two ways.

The first option is to use Prefix-SIDs and allow traffic to follow the shortest paths for the prefixes. Prefix-SIDs for node prefixes, i.e. Node-SIDs (for loopback addresses), can be used both for encapsulating service traffic to service nodes (e.g. VPN PEs) and for SR-TE traffic steering purposes (see below), but the benefits of other Prefix-SIDs are not clear, so currently only Node-SIDs are supported with RIFT.

The second option is to use SR-TE and follow a specific segment list in the packet header. Each node in the path steers the packet to the currently active segment in the list, following the natural path for that segment (see above). Since a node only has the full topology south of it, and a leaf node does not have any south topology, the traffic steering information (i.e. the segment list) must be programmed by controllers into ingress nodes via SR policies.

Support for Adjacency SIDs will be considered in future revisions.

Consider the following 4-level topology:

```
                  ToF1                       ToF2

        Spine1_11    Spine1_21  |  Spine1_21    Spine1_22
                                |
        Spine2_11    Spine2_21  |  Spine2_21    Spine2_22
                                |
          Leaf11       Leaf12   |   Leaf21       Leaf22
```

Suppose the TE controller instructs Leaf11 to send a packet to Spine2_11 with label stack (Label_TOF2, Label_Spine2_21, Label_Leaf21). Spine2_11 recognizes that Label_TOF2 maps to node TOF2 and it should not simply follow the default route (because the default route could lead to an unintended path via TOF1). In other words, each node needs to have a specific route to every node (that may appear in the segment list). That means for RIFT the southbound distance vector routing needs to additionally advertise routes for the nodes in the north, and they must be propagated all the way down. Each node originates a route for its own loopback address and advertises it southbound, with a special marking that allows a south node to re-advertise it further south.

If loopback addresses are not used, similar "routes" for System IDs must be used. It is RECOMMENDED to use loopback addresses to reuse existing mechanisms.

3. Well-Known KV Registry Values

This section requests an entry from the RIFT Well-Known Key-Type Registry for networks that use SR along with suggested values in accordance with RIFT-KV-REGISTRY [RIFT-KV-REGISTRY].

Name	Value	Description
SRIFT Node	TBD	Key-Type describing a SRIFT node

Table 1: Requested Entries

3.1. SRIFT Node Key-Type

```
 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
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      TBD2     |                SRIFT Node                       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     (System ID,                                               |
|      Loopback Address,                                        |
|      SRGB Label Base,                                         |
|      SRGB Label Range,                                        |
|      Node-SID,)                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

where:

System ID:
A node's 64-bit RIFT System ID.

Loopback Address:
A node's loopback address. This MAY be set to 0 if loopback addresses are not used.

SRGB Label Base:
The first valid label within the corresponding node's SRGB.

SRGB Label Range:
The total number of valid labels in the corresponding node's SRGB.

Node-SID:
The corresponding node's Node-SID value.

4. Security Considerations

This document does not introduce any new security concerns with RIFT or any other referenced protocols. RIFT KV TIEs are already extensively secured via RIFT's specification.

5. Acknowledgements

The authors thank Bruno Rijsman and Antoni Przygenda for their review and suggestions.

6. References

6.1. 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>.

[RFC8402] Filsfils, C., Ed., Previdi, S., Ed., Ginsberg, L., Decraene, B., Litkowski, S., and R. Shakir, "Segment Routing Architecture", RFC 8402, DOI 10.17487/RFC8402, July 2018, <https://www.rfc-editor.org/info/rfc8402>.

[RIFT] Przygienda, T., Sharma, A., Thubert, P., Rijsman, B., and D. Afanasiev, "RIFT: Routing in Fat Trees", Work in Progress, draft-ietf-rift-rift-12, May 2020.

[RIFT-KV-REGISTRY] Przygienda, T., "RIFT Keys Structure and Well-Known Registry in Key Value TIE", Work in Progress, draft-przygienda-rift-kv-registry-00, December 2020.

6.2. Informative References

[RFC8665] Psenak, P., Ed., Previdi, S., Ed., Filsfils, C., Gredler, H., Shakir, R., Henderickx, W., and J. Tantsura, "OSPF Extensions for Segment Routing", RFC 8665, DOI 10.17487/RFC8665, December 2019, <https://www.rfc-editor.org/info/rfc8665>.

[RFC8667] Previdi, S., Ed., Ginsberg, L., Ed., Filsfils, C., Bashandy, A., Gredler, H., and B. Decraene, "IS-IS Extensions for Segment Routing", RFC 8667, DOI 10.17487/RFC8667, December 2019, <https://www.rfc-editor.org/info/rfc8667>.

Authors' Addresses

Zhaohui Zhang
Juniper Networks

Email: zzhang@juniper.net

Jeff Tantsura
Juniper Networks

Email: jefftant.ietf@gmail.com

Jordan Head
Juniper Networks

Email: jhead@juniper.net

Don Fedyk
Individual

Email: don.fedyk@gmail.com