

RIFT Working Group
Internet-Draft
Intended status: Standards Track
Expires: September 2, 2018

T. Przygienda, Ed.
Juniper Networks
A. Sharma
Comcast
A. Atlas
J. Drake
Juniper Networks
Mar 01, 2018

RIFT: Routing in Fat Trees
draft-przygienda-rift-05

Abstract

This document outlines a specialized, dynamic routing protocol for Clos and fat-tree network topologies. The protocol (1) deals with automatic construction of fat-tree topologies based on detection of links, (2) minimizes the amount of routing state held at each level, (3) automatically prunes the topology distribution exchanges to a sufficient subset of links, (4) supports automatic disaggregation of prefixes on link and node failures to prevent black-holing and suboptimal routing, (5) allows traffic steering and re-routing policies, (6) allows non-ECMP forwarding, (7) automatically re-balances traffic towards the spines based on bandwidth available and ultimately (8) 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 September 2, 2018.

Copyright Notice

Copyright (c) 2018 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	4
1.1. Requirements Language	5
2. Reference Frame	5
2.1. Terminology	5
2.2. Topology	8
3. Requirement Considerations	10
4. RIFT: Routing in Fat Trees	12
4.1. Overview	12
4.2. Specification	13
4.2.1. Transport	13
4.2.2. Link (Neighbor) Discovery (LIE Exchange)	13
4.2.3. Topology Exchange (TIE Exchange)	14
4.2.3.1. Topology Information Elements	14
4.2.3.2. South- and Northbound Representation	15
4.2.3.3. Flooding	18
4.2.3.4. TIE Flooding Scopes	18
4.2.3.5. Initial and Periodic Database Synchronization	20
4.2.3.6. Purging	20
4.2.3.7. Southbound Default Route Origination	21
4.2.3.8. Optional Automatic Flooding Reduction and Partitioning	21
4.2.4. Policy-Guided Prefixes	23
4.2.4.1. Ingress Filtering	24
4.2.4.2. Applying Policy	24
4.2.4.3. Store Policy-Guided Prefix for Route Computation and Regeneration	25
4.2.4.4. Re-origination	26
4.2.4.5. Overlap with Disaggregated Prefixes	26
4.2.5. Reachability Computation	26
4.2.5.1. Northbound SPF	27
4.2.5.2. Southbound SPF	27

4.2.5.3. East-West Forwarding Within a Level	28
4.2.6. Attaching Prefixes	28
4.2.7. Attaching Policy-Guided Prefixes	29
4.2.8. Automatic Disaggregation on Link & Node Failures . .	30
4.2.9. Optional Autoconfiguration	33
4.2.9.1. Terminology	34
4.2.9.2. Automatic SystemID Selection	35
4.2.9.3. Generic Fabric Example	35
4.2.9.4. Level Determination Procedure	36
4.2.9.5. Resulting Topologies	37
4.2.10. Stability Considerations	39
4.3. Further Mechanisms	40
4.3.1. Overload Bit	40
4.3.2. Optimized Route Computation on Leafs	40
4.3.3. Key/Value Store	40
4.3.3.1. Southbound	40
4.3.3.2. Northbound	41
4.3.4. Interactions with BFD	41
4.3.5. Fabric Bandwidth Balancing	41
4.3.5.1. Northbound Direction	42
4.3.5.2. Southbound Direction	43
4.3.6. Segment Routing Support with RIFT	43
4.3.6.1. Global Segment Identifiers Assignment	43
4.3.6.2. Distribution of Topology Information	44
4.3.7. Leaf to Leaf Procedures	44
4.3.8. Other End-to-End Services	45
4.3.9. Address Family and Multi Topology Considerations . .	45
4.3.10. Reachability of Internal Nodes in the Fabric	45
4.3.11. One-Hop Healing of Levels with East-West Links . . .	45
5. Examples	46
5.1. Normal Operation	46
5.2. Leaf Link Failure	47
5.3. Partitioned Fabric	48
5.4. Northbound Partitioned Router and Optional East-West Links	50
6. Implementation and Operation: Further Details	51
6.1. Considerations for Leaf-Only Implementation	51
6.2. Adaptations to Other Proposed Data Center Topologies . .	51
6.3. Originating Non-Default Route Southbound	52
7. Security Considerations	52
8. Information Elements Schema	53
8.1. common.thrift	53
8.2. encoding.thrift	57
9. IANA Considerations	63
10. Acknowledgments	63
11. References	63
11.1. Normative References	63
11.2. Informative References	65

Authors' Addresses	66
------------------------------	----

1. Introduction

Clos [CLOS] and Fat-Tree [FATTREE] 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 routing protocols were geared towards a network with an irregular topology and low degree of connectivity originally but given they were the only available mechanisms, consequently several attempts to apply those to Clos have been made. Most successfully BGP [RFC4271] [RFC7938] has been extended to this purpose, not as much due to its inherent suitability to solve the problem but rather because the perceived capability to modify it "quicker" and the immanent difficulties with link-state [DIJKSTRA] based protocols to perform in large scale densely meshed topologies.

In looking at the problem through the lens of its requirements an optimal approach does not seem however to be a simple modification of either a link-state (distributed computation) or distance-vector (diffused computation) approach but rather a mixture of both, colloquially best described as "link-state towards the spine" and "distance vector towards the leafs". In other words, "bottom" levels are flooding their link-state information in the "northern" direction while each switch generates under normal conditions a default route and floods it in the "southern" direction. Obviously, such aggregation can blackhole in cases of misconfiguration or failures and this has to be addressed somehow.

For the visually oriented reader, Figure 1 presents a first 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 routes to them. In the second row of the database we indicate that a partial information of other nodes in the same level is available as well; the details of how this is achieved should be postponed for the moment. Whereas when we look at the "bottom" of the fabric we see that the topology of the leafs is basically empty and they only hold a load balanced default route to the next level.

The balance of this document details the resulting protocol and fills in the missing details.

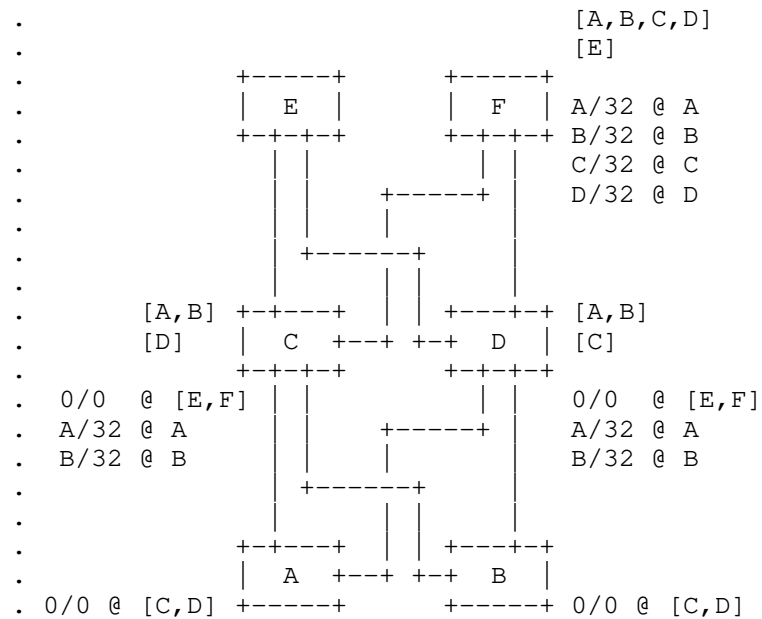


Figure 1: RIFT information distribution

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. Reference Frame

2.1. 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 [RFC1142], [ISO10589] as well as the according graph theoretical concepts of shortest path first (SPF) [DIJKSTRA] computation and directed acyclic graphs (DAG).

Level: Clos and Fat Tree networks are trees and 'level' denotes the set of nodes at the same height in such a network, where the bottom level is level 0. 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.

Spine/Aggregation/Edge Levels: Traditional names for Level 2, 1 and 0 respectively. Level 0 is often called leaf as well.

Point of Delivery (PoD): A 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.

Spine: The set of nodes that provide inter-PoD communication. These nodes are also organized into levels (typically one, three, or five levels). Spine nodes do not belong to any PoD and are assigned the PoD value 0 to indicate this.

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 4.2.9).

Connected Spine: In case a spine level represents a connected graph (discounting links terminating at different levels), we call it a "connected spine", in case a spine level consists of multiple partitions, we call it a "disconnected" or "partitioned spine". In other terms, a spine without east-west links is disconnected and is the typical configuration for Clos and Fat Tree networks.

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.

Southbound representation: Information sent towards a lower level representing only limited amount of information.

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. It can be thought of as largely equivalent to ISIS LSPs or OSPF LSA. We will talk about N-TIEs when talking about TIEs in the northbound representation and S-TIEs for the southbound equivalent.

Node TIE: This is an acronym for a "Node Topology Information Element", largely equivalent to OSPF Node LSA, i.e. it contains all neighbors the node discovered and information about node itself.

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 N-TIE and in case of S-TIE the necessary default and de-aggregated prefixes the node passes southbound.

Policy-Guided Information: Information that is passed in either southbound direction or north-bound direction by the means of diffusion and can be filtered via policies. Policy-Guided Prefixes and KV Ties are examples of Policy-Guided Information.

Key Value TIE: A S-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.

PGP: Policy-Guided Prefixes allow to support traffic engineering that cannot be achieved by the means of SPF computation or normal node and prefix S-TIE origination. S-PGPs are propagated in south direction only and N-PGPs follow northern direction strictly.

De-aggregation/Disaggregation: Process in which a node decides to advertise certain prefixes it received in N-TIEs to prevent black-holing and suboptimal routing upon link failures.

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 adjacencies.

FL: Flooding Leader for a specific system has a dedicated role to flood TIEs of that system.

BAD: This is an acronym for Bandwidth Adjusted Distance. RIFT calculates the amount of northbound bandwidth available for a node compared to other nodes at the same level and adjusts the default route distance accordingly to allow for the lower level to weight their forwarding load balancing.

Overloaded: Applies to a node advertising 'overload' attribute as set. The semantics closely follow the meaning of the same attribute in [RFC1142].

2.2. Topology

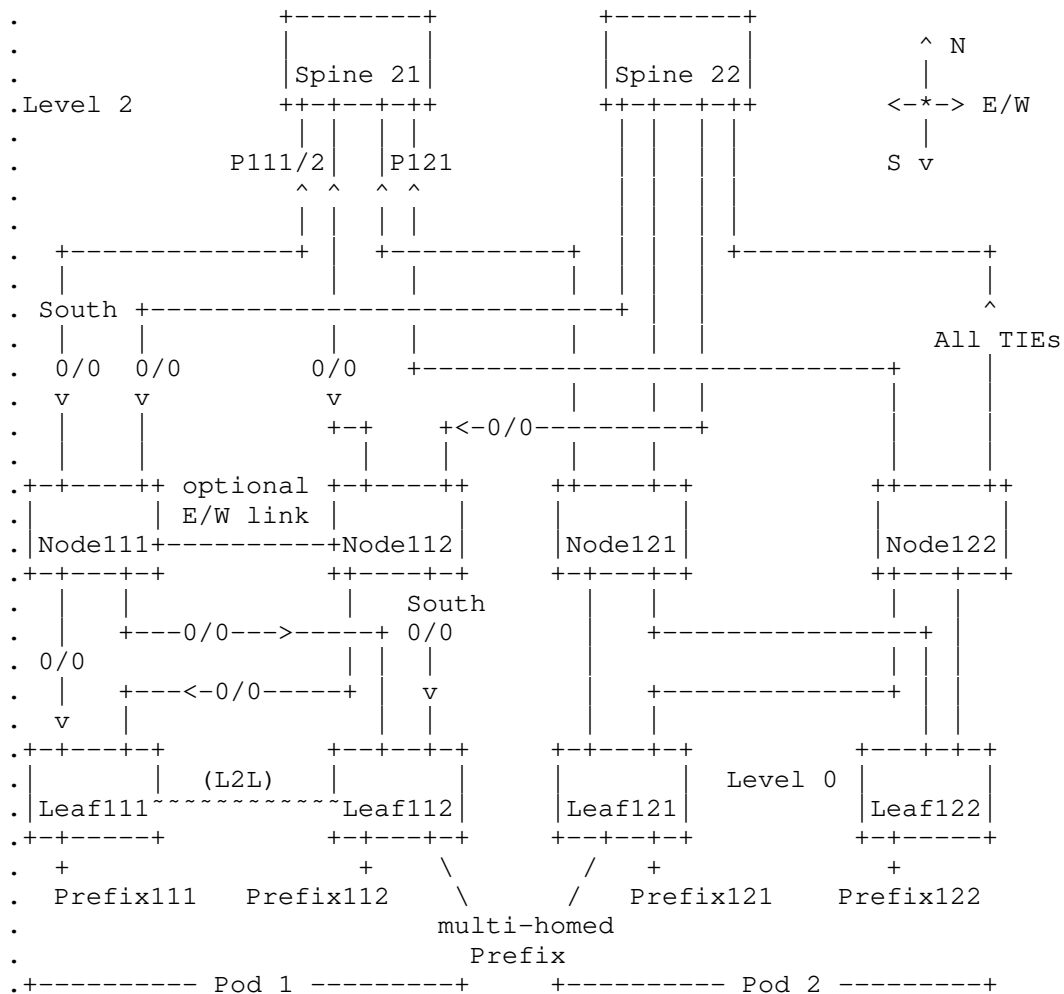


Figure 2: A two level spine-and-leaf topology

We will use this topology (called commonly a fat tree/network in modern DC considerations [VAHDAT08] as homonym to the original definition of the term [FATTREE]) in all further considerations. It depicts a generic "fat-tree" and the concepts explained in three levels here carry by induction for further levels and higher degrees of connectivity. However, this document will deal with designs that provide only sparser connectivity as well.

3. Requirement Considerations

[RFC7938] gives the original set of requirements augmented here based upon recent experience in the operation of fat-tree networks.

- REQ1: The control protocol should discover the physical links automatically and be able to detect cabling that violates fat-tree topology constraints. It must react 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. E.g. connecting a leaf to a spine at level 2 should be detected and ideally prevented.
- REQ2: A node without any configuration beside default values should come up at the correct level in any PoD it is introduced into. Optionally, it must be possible to configure nodes to restrict their participation to the PoD(s) targeted at any level.
- REQ3: Optionally, the protocol should allow to provision data centers where the individual switches carry no configuration information and are all deriving their level from a "seed". Observe that this requirement may collide with the desire to detect cabling misconfiguration and with that only one of the requirements can be fully met in a chosen configuration mode.
- REQ4: The solution should allow for minimum size routing information base and forwarding tables at leaf level for speed, cost and simplicity reasons. Holding excessive amount of information away from leaf nodes simplifies operation and lowers cost of the underlay.
- REQ5: Very high degree of ECMP must be supported. Maximum ECMP is currently understood as the most efficient routing approach to maximize the throughput of switching fabrics [MAKSIC2013].
- REQ6: Non equal cost anycast must be supported to allow for easy and robust multi-homing of services without regressing to careful balancing of link costs.
- REQ7: Traffic engineering should be allowed by modification of prefixes and/or their next-hops.
- REQ8: The solution should allow for access to link states of the whole topology to enable efficient support for modern

control architectures like SPRING [RFC7855] or PCE [RFC4655].

- REQ9: The solution should easily accommodate opaque data to be carried throughout the topology to subsets of nodes. This can be used for many purposes, one of them being a key-value store that allows bootstrapping of nodes based right at the time of topology discovery.
- REQ10: Nodes should be taken out and introduced into production with minimum wait-times and minimum of "shaking" of the network, i.e. radius of propagation (often called "blast radius") of changed information should be as small as feasible.
- REQ11: The protocol should allow for maximum aggregation of carried routing information while at the same time automatically de-aggregating the prefixes to prevent black-holing in case of failures. The de-aggregation should support maximum possible ECMP/N-ECMP remaining after failure.
- REQ12: Reducing the scope of communication needed throughout the network on link and state failure, as well as reducing advertisements of repeating, idiomatic or policy-guided information in stable state is highly desirable since it leads to better stability and faster convergence behavior.
- REQ13: Once a packet traverses a link in a "southbound" direction, it must not take any further "northbound" steps along its path to delivery to its destination under normal conditions. Taking a path through the spine in cases where a shorter path is available is highly undesirable.
- REQ14: Parallel links between same set of nodes must be distinguishable for SPF, failure and traffic engineering purposes.
- REQ15: The protocol must not rely on interfaces having discernible unique addresses, i.e. it must operate in presence of unnumbered links (even parallel ones) or links of a single node having same addresses.
- REQ16: It would be desirable to achieve fast re-balancing of flows when links, especially towards the spines are lost or provisioned without regressing to per flow traffic engineering which introduces significant amount of complexity while possibly not being reactive enough to account for short-lived flows.

Following list represents possible requirements and requirements under discussion:

PEND1: Supporting anything but point-to-point links is a non-requirement. Questions remain: for connecting to the leaves, is there a case where multipoint is desirable? One could still model it as point-to-point links; it seems there is no need for anything more than a NBMA-type construct.

PEND2: What is the maximum scale of number leaf prefixes we need to carry. Is 500'000 enough ?

Finally, following are the non-requirements:

NONREQ1: Broadcast media support is unnecessary.

NONREQ2: Purging is unnecessary given its fragility and complexity and today's large memory size on even modest switches and routers.

NONREQ3: Special support for layer 3 multi-hop adjacencies is not part of the protocol specification. Such support can be easily provided by using tunneling technologies the same way IGP's today are solving the problem.

4. RIFT: Routing in Fat Trees

Derived from the above requirements we present 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][RFC1142] when "pointing north" and path-vector [RFC4271] protocol when "pointing south". Albeit an unusual combination, it does quite naturally exhibit the desirable properties we seek.

4.1. Overview

The singular property of RIFT is that it floods northbound "flat" link-state information so that each level understands the full topology of levels south of it. In contrast, in the southbound direction the protocol operates like a path vector protocol or rather a distance vector with implicit split horizon since the topology constraints make a diffused computation front propagating in all directions unnecessary.

To account for the "northern" and the "southern" information split the link state database is partitioned into "north representation" and "south representation" TIEs, whereas in simplest terms the N-TIEs contain a link state topology description of lower levels and and

S-TIEs carry simply default routes. This oversimplified view will be refined gradually in following sections while introducing protocol procedures aimed to fulfill the described requirements.

4.2. Specification

4.2.1. Transport

All protocol elements are carried over UDP. Once QUIC [QUIC] achieves the desired stability in deployments it may prove a valuable candidate for TIE transport.

All packet formats are defined in Thrift models in Section 8.

Future versions may include a [PROTOBUF] schema.

4.2.2. Link (Neighbor) Discovery (LIE Exchange)

LIE exchange happens over well-known administratively locally scoped IPv4 multicast address [RFC2365] or link-local multicast scope for IPv6 [RFC4291] and SHOULD be sent with a TTL of 1 to prevent RIFT information reaching beyond a single L3 next-hop in the topology. LIEs are exchanged over all links running RIFT.

Unless Section 4.2.9 is used, each node is provisioned with the level at which it is operating and its PoD (or otherwise a default level and "undefined" PoD are assumed; meaning that leafs do not need to be configured at all). Nodes in the spine are configured with an "undefined" PoD. This information is propagated in the LIEs exchanged.

A node tries to form a three way adjacency if and only if (definitions of LEAF_ONLY are found in Section 4.2.9)

1. the node is in the same PoD or either the node or the neighbor advertises "undefined" 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 with level higher than the neighboring node OR

ii) the neighboring node is at level 0 OR

iii) both nodes are at level 0 AND both indicate support for Section 4.3.7 OR

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

].

Rule in Paragraph 3 MAY be optionally disregarded by a node if PoD detection is undesirable or has to be disregarded.

A node configured with "undefined" PoD membership MUST, after building first northbound adjacency making it participant in a PoD, advertise that PoD as part of its LIEs.

LIEs arriving with a TTL 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.

LIE exchange uses three-way handshake mechanism [RFC5303]. Precise finite state machines will be provided in later versions of this specification. LIE packets contain nonces and may contain an SHA-1 [RFC6234] over nonces and some of the LIE data which prevents corruption and replay attacks. TIE flooding reuses those nonces to prevent mismatches and can use those for security purposes in case it is using QUIC [QUIC]. Section 7 will address the precise security mechanisms in the future.

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.

TIE exchange mechanism uses 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.

TIEs contain sequence numbers, lifetimes and a type. Each type has a large 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 point of the design space is a prefix per TIE which leads to BGP-like behavior vs. dense packing into few TIEs leading to more traditional IGP trade-off with fewer TIEs. An implementation may even rehash at the cost of significant amount of re-advertisements of TIEs.

More information about the TIE structure can be found in the schema in Section 8.

4.2.3.2. South- and Northbound Representation

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

The N-TIEs hold all of the node's adjacencies, local prefixes and northbound policy-guided prefixes while the S-TIEs hold only all of the node's adjacencies and the default prefix with necessary disaggregated prefixes and southbound policy-guided prefixes. We will explain this in detail further in Section 4.2.8 and Section 4.2.4.

The TIE types are symmetric in both directions and Table 1 provides a quick reference to the different TIE types including direction and their function.

TIE-Type	Content
node N-TIE	node properties, adjacencies and information helping in complex disaggregation scenarios
node S-TIE	same content as node N-TIE except the information to help disaggregation
Prefix N-TIE	contains nodes' directly reachable prefixes
Prefix S-TIE	contains originated defaults and de-aggregated prefixes
PGP N-TIE	contains nodes north PGPs
PGP S-TIE	contains nodes south PGPs
KV N-TIE	contains nodes northbound KVs
KV S-TIE	contains nodes southbound KVs

Table 1: TIE Types

As an example illustrating a databases holding both representations, consider the topology in Figure 2 with the optional link between node 111 and node 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 and the PGP elements which may be included in their S-TIEs or N-TIEs are not shown.

Spine21 S-TIEs:

Node S-TIE:

```
NodeElement(layer=2, neighbors((Node111, layer 1, cost 1),
(Node112, layer 1, cost 1), (Node121, layer 1, cost 1),
(Node122, layer 1, cost 1)))
```

Prefix S-TIE:

```
SouthPrefixesElement(prefixes(0/0, cost 1), (::/0, cost 1))
```

Node111 S-TIEs:


```
Node S-TIE:
  NodeElement(layer=1, neighbors((Spine21, layer 2, cost 1, links(...)),
    (Spine22, layer 2, cost 1, links(...)),
    (Node112, layer 1, cost 1, links(...)),
    (Leaf111, layer 0, cost 1, links(...)),
    (Leaf112, layer 0, cost 1, links(...))))
Prefix S-TIE:
  SouthPrefixesElement(prefixes(0/0, cost 1), (::/0, cost 1))

Node111 N-TIEs:
Node N-TIE:
  NodeElement(layer=1,
    neighbors((Spine21, layer 2, cost 1, links(...)),
      (Spine22, layer 2, cost 1, links(...)),
      (Node112, layer 1, cost 1, links(...)),
      (Leaf111, layer 0, cost 1, links(...)),
      (Leaf112, layer 0, cost 1, links(...))))
Prefix N-TIE:
  NorthPrefixesElement(prefixes(Node111.loopback)

Node121 S-TIEs:
Node S-TIE:
  NodeElement(layer=1, neighbors((Spine21, layer 2, cost 1),
    (Spine22, layer 2, cost 1), (Leaf121, layer 0, cost 1),
    (Leaf122, layer 0, cost 1)))
Prefix S-TIE:
  SouthPrefixesElement(prefixes(0/0, cost 1), (::/0, cost 1))

Node121 N-TIEs:
Node N-TIE:
  NodeLinkElement(layer=1,
    neighbors((Spine21, layer 2, cost 1, links(...)),
      (Spine22, layer 2, cost 1, links(...)),
      (Leaf121, layer 0, cost 1, links(...)),
      (Leaf122, layer 0, cost 1, links(...))))
Prefix N-TIE:
  NorthPrefixesElement(prefixes(Node121.loopback)

Leaf112 N-TIEs:
Node N-TIE:
  NodeLinkElement(layer=0,
    neighbors((Node111, layer 1, cost 1, links(...)),
      (Node112, layer 1, cost 1, links(...))))
Prefix N-TIE:
  NorthPrefixesElement(prefixes(Leaf112.loopback, Prefix112,
    Prefix_MH))
```

Figure 3: example TIES generated in a 2 level spine-and-leaf topology

4.2.3.3. Flooding

The mechanism used to distribute TIES is the well-known (albeit modified in several respects to address fat tree requirements) flooding mechanism used by today's link-state protocols. Albeit initially more demanding to implement it avoids many problems with diffused computation update style used by path vector. As described before, TIES themselves are transported over UDP with the ports indicates in the LIE exchanges and using the destination address (for unnumbered IPv4 interfaces same considerations apply as in equivalent OSPF case) on which the LIE adjacency has been formed.

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.

Precise finite state machines and procedures will be provided in later versions of this specification.

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 N-TIE is flooded northbound, providing a node at a given level with the complete topology of the Clos or Fat Tree network underneath 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 may 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 S-TIES, consisting of all node's adjacencies and prefix S-TIES with 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 a E-W disconnected node in a given level to receive the S-TIES of other nodes at its level, every *NODE* S-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; those TIES need to be flooded to satisfy algorithms in Section 4.2.5. 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 flooding scopes are given in Table 2. Those rules govern as well

what SHOULD be included in TIDEs towards neighbors. East-West flooding scopes are identical to South flooding scopes.

Node S-TIE "reflection" allows to support disaggregation on failures describes in Section 4.2.8 and flooding reduction in Section 4.2.3.8.

Packet Type vs. Peer Direction	South	North
node S-TIE	flood self-originated only	flood if TIE originator's level is higher than own level
non-node S-TIE	flood self-originated only	flood only if TIE originator is equal peer
all N-TIEs	never flood	flood always
TIDE	include TIEs in flooding scope	include TIEs in flooding scope
TIRE	include all N-TIEs and all peer's self-originated TIEs and all node S-TIEs	include only if TIE originator is equal peer

Table 2: Flooding Scopes

As an example to illustrate these rules, consider using the topology in Figure 2, with the optional link between node 111 and node 112, and the associated TIEs given in Figure 3. The flooding from particular nodes of the TIEs is given in Table 3.

Router floods to	Neighbor	TIEs
Leaf111	Node112	Leaf111 N-TIEs, Node111 node S-TIE
Leaf111	Node111	Leaf111 N-TIEs, Node112 node S-TIE
Node111	Leaf111	Node111 S-TIEs
Node111	Leaf112	Node111 S-TIEs
Node111	Node112	Node111 S-TIEs
Node111	Spine21	Node111 N-TIEs, Leaf111 N-TIEs, Leaf112 N-TIEs, Spine22 node S-TIE
Node111	Spine22	Node111 N-TIEs, Leaf111 N-TIEs, Leaf112 N-TIEs, Spine21 node S-TIE
...
Spine21	Node111	Spine21 S-TIEs
Spine21	Node112	Spine21 S-TIEs
Spine21	Node121	Spine21 S-TIEs
Spine21	Node122	Spine21 S-TIEs
...

Table 3: Flooding some TIEs from example topology

4.2.3.5. 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 2.

4.2.3.6. Purging

RIFT does not purge information that has been distributed by the protocol. Purging mechanisms in other routing protocols have proven to be complex and fragile over many years of experience. Abundant amounts of memory are available today even on low-end platforms. The information will age out and all computations will deliver correct results if a node leaves the network due to the new information distributed by its adjacent nodes.

Once a RIFT node issues a TIE with an ID, it MUST preserve the ID as long as feasible (also when the protocol restarts), even if the TIE loses 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 will 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 supercede 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.

4.2.3.7. Southbound Default Route Origination

Under certain conditions nodes issue a default route in their South Prefix TIEs with metrics as computed in Section 4.3.5.1.

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 layer (otherwise the node S-TIEs cannot be reflected and the nodes in e.g. POD 1 nad 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.

4.2.3.8. Optional Automatic Flooding Reduction and Partitioning

Several nodes can, but strictly only under conditions defined below, run a hashing function based on TIE originator value and partition flooding between them.

Steps for flooding reduction and partitioning:

1. select all nodes in the same level for which
 - A. node S-TIEs have been received AND

- B. which have precisely the same non-empty sets of respectively north and south neighbor adjacencies AND
 - C. have at least one shared southern neighbor including backlink verification and
 - D. support flooding reduction (overload bits are ignored)
- and then
- 2. run on the chosen set a hash algorithm using nodes flood priorities and IDs to select flooding leader and backup per TIE originator ID, i.e. each node floods immediately through to all its necessary neighbors TIEs that it received with an originator ID that makes it the flooding leader or backup for this originator. The preference (higher is better) is computed as $\text{XOR}(\text{TIE-ORIGINATOR-ID} \ll 1, \sim \text{OWN-SYSTEM-ID})$, whereas \ll is a non-circular shift and \sim is bit-wise NOT.
 - 3. In the very unlikely case of hash collisions on either of the two nodes with highest values (i.e. either does NOT produce unique hashes as compared to all other hash values), the node running the election does not attempt to reduce flooding.

Additional rules for flooding reduction and partitioning:

- 1. A node always floods its own TIEs
- 2. A node generates TIDEs as usual but when receiving TIREs with requests for TIEs for a node for which it is not a flooding leader or backup it ignores such TIDEs on first request only. Normally, the flooding leader should satisfy the requestor and with that no further TIREs for such TIEs will be generated. Otherwise, the next set of TIDEs and TIREs will lead to flooding independent of the flooding leader status.
- 3. A node receiving a TIE originated by a node for which it is not a flooding leader floods such TIEs only when receiving an out-of-date TIDE for them, except for the first one.

The mechanism can be implemented optionally in each node. The capability is carried in the node S-TIE (and for symmetry purposes in node N-TIE as well but it serves no purpose there currently).

Obviously flooding reduction does NOT apply to self originated TIEs. Observe further that all policy-guided information consists of self-originated TIEs.

4.2.4. Policy-Guided Prefixes

In a fat tree, it can be sometimes desirable to guide traffic to particular destinations or keep specific flows to certain paths. In RIFT, this is done by using policy-guided prefixes with their associated communities. Each community is an abstract value whose meaning is determined by configuration. It is assumed that the fabric is under a single administrative control so that the meaning and intent of the communities is understood by all the nodes in the fabric. Any node can originate a policy-guided prefix.

Since RIFT uses distance vector concepts in a southbound direction, it is straightforward to add a policy-guided prefix to an S-TIE. For easier troubleshooting, the approach taken in RIFT is that a node's southbound policy-guided prefixes are sent in its S-TIE and the receiver does inbound filtering based on the associated communities (an egress policy is imaginable but would lead to different S-TIEs per neighbor possibly which is not considered in RIFT protocol procedures). A southbound policy-guided prefix can only use links in the south direction. If an PGP S-TIE is received on an east-west or northbound link, it must be discarded by ingress filtering.

Conceptually, a southbound policy-guided prefix guides traffic from the leaves up to at most the north-most layer. It is also necessary to have northbound policy-guided prefixes to guide traffic from the north-most layer down to the appropriate leaves. Therefore, RIFT includes northbound policy-guided prefixes in its N PGP-TIE and the receiver does inbound filtering based on the associated communities. A northbound policy-guided prefix can only use links in the northern direction. If an N PGP TIE is received on an east-west or southbound link, it must be discarded by ingress filtering.

By separating southbound and northbound policy-guided prefixes and requiring that the cost associated with a PGP is strictly monotonically increasing at each hop, the path cannot loop. Because the costs are strictly increasing, it is not possible to have a loop between a northbound PGP and a southbound PGP. If east-west links were to be allowed, then looping could occur and issues such as counting to infinity would become an issue to be solved. If complete generality of path - such as including east-west links and using both north and south links in arbitrary sequence - then a Path Vector protocol or a similar solution must be considered.

If a node has received the same prefix, after ingress filtering, as a PGP in an S-TIE and in an N-TIE, then the node determines which policy-guided prefix to use based upon the advertised cost.

A policy-guided prefix is always preferred to a regular prefix, even if the policy-guided prefix has a larger cost. Section 8 provides normative indication of prefix preferences.

The set of policy-guided prefixes received in a TIE is subject to ingress filtering and then re-originated to be sent out in the receiver's appropriate TIE. Both the ingress filtering and the re-origination use the communities associated with the policy-guided prefixes to determine the correct behavior. The cost on re-advertisement MUST increase in a strictly monotonic fashion.

4.2.4.1. Ingress Filtering

When a node X receives a PGP S-TIE or a PGP N-TIE that is originated from a node Y which does not have an adjacency with X, all PGPs in such a TIE MUST be filtered. Similarly, if node Y is at the same layer as node X, then X MUST filter out PGPs in such S- and N-TIEs to prevent loops.

Next, policy can be applied to determine which policy-guided prefixes to accept. Since ingress filtering is chosen rather than egress filtering and per-neighbor PGPs, policy that applies to links is done at the receiver. Because the RIFT adjacency is between nodes and there may be parallel links between the two nodes, the policy-guided prefix is considered to start with the next-hop set that has all links to the originating node Y.

A policy-guided prefix has or is assigned the following attributes:

cost: This is initialized to the cost received

community_list: This is initialized to the list of the communities received.

next_hop_set: This is initialized to the set of links to the originating node Y.

4.2.4.2. Applying Policy

The specific action to apply based upon a community is deployment specific. Here are some examples of things that can be done with communities. The length of a community is a 64 bits number and it can be written as a single field M or as a multi-field (S = M[0-31], T = M[32-63]) in these examples. For simplicity, the policy-guided prefix is referred to as P, the processing node as X and the originator as Y.

Prune Next-Hops: Community Required: For each next-hop in P.next_hop_set, if the next-hop does not have the community, prune that next-hop from P.next_hop_set.

Prune Next-Hops: Avoid Community: For each next-hop in P.next_hop_set, if the next-hop has the community, prune that next-hop from P.next_hop_set.

Drop if Community: If node X has community M, discard P.

Drop if not Community: If node X does not have the community M, discard P.

Prune to ifIndex T: For each next-hop in P.next_hop_set, if the next-hop's ifIndex is not the value T specified in the community (S,T), then prune that next-hop from P.next_hop_set.

Add Cost T: For each appearance of community S in P.community_list, if the node X has community S, then add T to P.cost.

Accumulate Min-BW T: Let bw be the sum of the bandwidth for P.next_hop_set. If that sum is less than T, then replace (S,T) with (S, bw).

Add Community T if Node matches S: If the node X has community S, then add community T to P.community_list.

4.2.4.3. Store Policy-Guided Prefix for Route Computation and Regeneration

Once a policy-guided prefix has completed ingress filtering and policy, it is almost ready to store and use. It is still necessary to adjust the cost of the prefix to account for the link from the computing node X to the originating neighbor node Y.

There are three different policies that can be used:

Minimum Equal-Cost: Find the lowest cost C next-hops in P.next_hop_set and prune to those. Add C to P.cost.

Minimum Unequal-Cost: Find the lowest cost C next-hop in P.next_hop_set. Add C to P.cost.

Maximum Unequal-Cost: Find the highest cost C next-hop in P.next_hop_set. Add C to P.cost.

The default policy is Minimum Unequal-Cost but well-known communities can be defined to get the other behaviors.

Regardless of the policy used, a node MUST store a PGP cost that is at least 1 greater than the PGP cost received. This enforces the strictly monotonically increasing condition that avoids loops.

Two databases of PGPs - from N-TIEs and from S-TIEs are stored. When a PGP is inserted into the appropriate database, the usual tie-breaking on cost is performed. Observe that the node retains all PGP TIEs due to normal flooding behavior and hence loss of the best prefix will lead to re-evaluation of TIEs present and re-advertisement of a new best PGP.

4.2.4.4. Re-origination

A node must re-originate policy-guided prefixes and retransmit them. The node has its database of southbound policy-guided prefixes to send in its S-TIE and its database of northbound policy-guided prefixes to send in its N-TIE.

Of course, a leaf does not need to re-originate southbound policy-guided prefixes.

4.2.4.5. Overlap with Disaggregated Prefixes

PGPs may overlap with prefixes introduced by automatic de-aggregation. The topic is under further discussion. The break in connectivity that leads to infeasibility of a PGP is mirrored in adjacency tear-down and according removal of such PGPs. Nevertheless, the underlying link-state flooding will be likely reacting significantly faster than a hop-by-hop redistribution and with that the preference for PGPs may cause intermittent black-holes.

4.2.5. Reachability Computation

A node has three sources of relevant information. A node knows the full topology south from the received N-TIEs. A node has the set of prefixes with associated distances and bandwidths from received S-TIEs. A node can also have a set of PGPs.

To compute reachability, a node runs conceptually a northbound and a southbound SPF. We call that N-SPF and S-SPF.

Since neither computation can "loop" (with due considerations given to PGPs), it is possible to compute non-equal-cost or even k-shortest paths [EPPSTEIN] and "saturate" the fabric to the extent desired.

4.2.5.1. Northbound SPF

N-SPF uses northbound and east-west adjacencies in North Node TIEs when progressing Dijkstra. Observe that this is really just a one hop variety since South Node TIEs are not re-flooded southbound beyond a single level (or east-west) and with that the computation cannot progress beyond adjacent nodes.

Default route found when crossing an E-W link is 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.

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

1. no north neighbors are advertising same or superssuming non-default prefix AND
2. the node does not originate a non-default superssuming prefix itself.

i.e. the E-W link can be used as the 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.

For N-SPF we are using the South Node TIEs to find according adjacencies to verify backlink connectivity. Just as in case of IS-IS or OSPF, two unidirectional links are associated together to confirm bidirectional connectivity.

4.2.5.2. Southbound SPF

S-SPF uses only the southbound adjacencies in the south node 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 uses northbound adjacencies in north node TIEs to verify backlink connectivity.

4.2.5.3. East-West Forwarding Within a Level

Ultimately, it should be observed that in presence of a "ring" of E-W links in a level neither SPF will provide a "ring protection" scheme since such a computation would have to deal necessarily with breaking of "loops" in generic Dijkstra sense; an application for which RIFT is not intended. It is outside the scope of this document how an underlay can be used to provide a full-mesh connectivity between nodes in the same layer that would allow for N-SPF to provide protection for a single node loosing all its northbound adjacencies (as long as any of the other nodes in the level are northbound connected).

Using south prefixes over horizontal links is optional and 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.

4.2.6. Attaching Prefixes

After the SPF is run, it is necessary to attach according prefixes. For S-SPF, prefixes from an N-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, type=spf, path_distance, next-hop set), accumulates these results. Obviously, the prefix retains its type which is used to tie-break between the same prefix advertised with different types.

In case of N-SPF prefixes from each S-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 S-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 S-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 cost next-hop to that neighbor. 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. RIFT route preferences are normalized by the according thrift model type.

An exemplary implementation for node X follows:

```

for each S-TIE
  if S-TIE.layer > X.layer
    next_hop_set = set of minimum cost links to the S-TIE.originator
    next_hop_cost = minimum cost link to S-TIE.originator
  end if
  for each prefix P in the S-TIE
    P.cost = P.cost + next_hop_cost
    if P not in route_database:
      add (P, type=DistVector, P.cost, next_hop_set) to route_database
    end if
    if (P in route_database) and
      (route_database[P].type is not PolicyGuided):
      if route_database[P].cost > P.cost:
        update route_database[P] with (P, DistVector, P.cost, next_hop_set)
      else if route_database[P].cost == P.cost
        update route_database[P] with (P, DistVector, P.cost,
          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 4: Adding Routes from S-TIE Prefixes

4.2.7. Attaching Policy-Guided Prefixes

Each policy-guided prefix P has its cost and next_hop_set already stored in the associated database, as specified in Section 4.2.4.3; the cost stored for the PGP is already updated to considering the cost of the link to the advertising neighbor. By definition, a policy-guided prefix is preferred to a regular prefix.

```

for each policy-guided prefix P:
  if P not in route_database:
    add (P, type=PolicyGuided, P.cost, next_hop_set)
  end if
  if P in route_database :
    if (route_database[P].type is not PolicyGuided) or
      (route_database[P].cost > P.cost):
      update route_database[P] with (P, PolicyGuided, P.cost, next_hop_set
)
    else if route_database[P].cost == P.cost
      update route_database[P] with (P, PolicyGuided, P.cost,
        merge(next_hop_set, route_database[P].next_hop_set))
    else
      // Not preferred route so ignore
    end if
  end if
end for

```

Figure 5: Adding Routes from Policy-Guided Prefixes

4.2.8. Automatic Disaggregation on Link & Node Failures

Under normal circumstances, node's S-TIEs contain just the adjacencies, a default route and policy-guided prefixes. 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 S-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 as 'de-aggregation' or 'dis-aggregation'.

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 N-TIEs are used to find all of prefixes it can reach and the set of next-hops in the lower level for each. 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 $|H(r)$. Observe that policy-guided prefixes are NOT affected since their scope is controlled by configuration.

2. The node uses reflected S-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 S-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 S-TIE, a node at the lower level that receives this S-TIE will not propagate it south-bound. Neither is it necessary for the receiving node to 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. 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 or east-west 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 layer from X 's perspective. Each entry is a list of south neighbor of X and a list of nodes of X .layer that can't reach that neighbor.
2. A node X determines its set of southbound neighbors `X.south_neighbors`.
3. For each S-TIE originated from a node Y that X has which is at X .layer, 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 to disaggregate any prefixes. If node X is advertising disaggregated prefixes in its S-TIE, X SHOULD remove them and re-advertise its according S-TIEs.

A node X computes its SPF based upon the received N-TIEs. 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_layer = { empty }
for each S-TIE
  if (S-TIE.layer == X.layer and
      X shares at least one S-neighbor with X)
    add S-TIE.originator to nodes_same_layer
  end if
end for

for each next-hop-set NHS
  isolated_nodes = nodes_same_layer
  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 S-TIE
if X's S-TIE is different
  schedule S-TIE for flooding
end if

```

Figure 6: Computation to Disaggregate Prefixes

Each disaggregated prefix is sent with the accurate path_distance. This allows a node to send the same S-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 S-TIE per node and complicate things unnecessarily. The PoD containing the prefix will prefer southbound anyway.
4. 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 only.
5. disaggregated prefix S-TIEs are not "reflected" by the lower layer, 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.

Ultimately, complex partitions of superspine on sparsely connected fabrics can lead to necessity of transitive disaggregation through multiple layers. The topic will be described and standardized in later versions of this document.

4.2.9. Optional Autoconfiguration

Each RIFT node can optionally operate in zero touch provisioning (ZTP) mode, i.e. it has no configuration (unless it is a superspine at the top of the topology or it MUST operate 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. This section describes the necessary concepts and procedures.

4.2.9.1. Terminology

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

UNDEFINED_LEVEL: An imaginary 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". SUPERSPINE_FLAG and CONFIGURED_LEVEL cannot be defined at the same time as this flag. It implies CONFIGURED_LEVEL value of 0.

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. SUPERSPINE_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 4.3.7. SUPERSPINE_FLAG is ignored when set at the same time as this flag. LEAF_2_LEAF implies LEAF_ONLY and the according restrictions.

LEVEL_VALUE: In ZTP case the original definition of "level" in Section 2.1 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). Offers from LIEs with 'not_a_ztp_offer' being true are not VOLs either.

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

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

SUPERSPINE_FLAG: Configuration flag provided to all superspines. LEAF_FLAG and CONFIGURED_LEVEL cannot be defined at the same time as this flag. It implies CONFIGURED_LEVEL value of 16. In fact, it is basically a shortcut for configuring same level at all superspine nodes which is unavoidable since an initial 'seed' is needed for other ZTP nodes to derive their level in the topology.

4.2.9.2. Automatic SystemID Selection

RIFT identifies each node via a SystemID which is a 64 bits wide integer. It is relatively simple to derive a, for all practical purposes collision free, value for each node on startup. As simple examples either system MAC and two random bytes can be used or an IPv4/IPv6 router ID interface address recycled as System ID. The router MUST ensure that such identifier is not changing very frequently (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 suggested in Section 7 are implemented).

4.2.9.3. 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 7. We assume all nodes being in the same PoD.

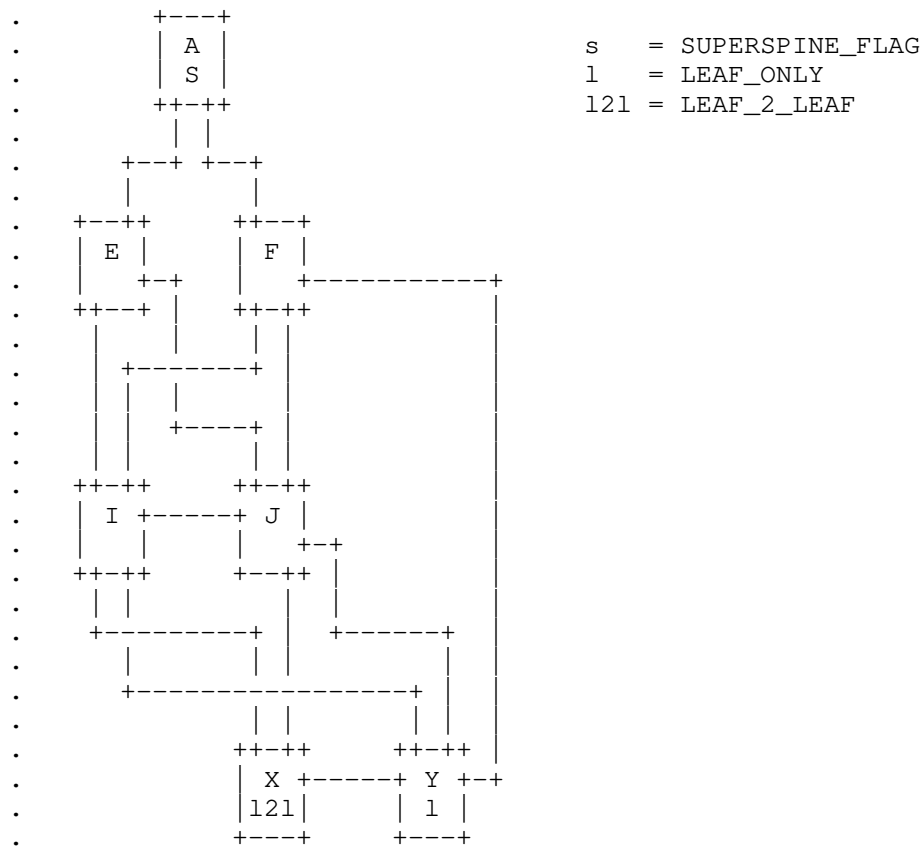


Figure 7: Generic ZTP Cabling Considerations

First, we need to anchor the "top" of the cabling and that's what the SUPERSPINE_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 or at the same level as X and consequently, X is south of it. This is unresolvable here until we "nail down the bottom" of the topology. To achieve that we use the leaf flags. We will see further then whether Y chooses to form adjacencies to F or I, J successively.

4.2.9.4. Level Determination Procedure

A node starting up with UNDEFINED_VALUE (i.e. without a CONFIGURED_LEVEL or any leaf or superspine 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 chooses on an ongoing basis from all VOLs the value of $\text{MAX}(\text{HAL}-1, 0)$ as its DERIVED_LEVEL. The node then starts to advertise this derived level.
3. 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.
4. A node MUST reset any adjacency that has changed the level it is offering and is in three way state.
5. 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.
6. After a level has been derived the node MUST set the 'not_a_ztp_offer' on LIEs towards all systems extending a VOL for HAL.

A node starting with LEVEL_VALUE being 0 (i.e. it assumes a leaf function 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.

Precise finite state machines will be provided in later versions of this specification.

4.2.9.5. Resulting Topologies

The procedures defined in Section 4.2.9.4 will lead to the RIFT topology and levels depicted in Figure 8.

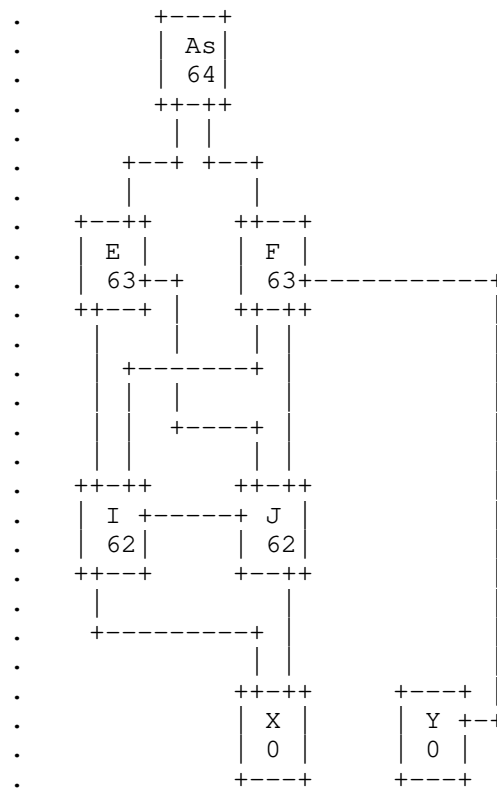


Figure 8: Generic ZTP Topology Autoconfigured

In case we imagine the LEAF_ONLY restriction on Y is removed the outcome would be very different however and result in Figure 9. This demonstrates basically that auto configuration prevents miscabling detection and with that can lead to undesirable effects when leafs are not "nailed" and arbitrarily cabled.

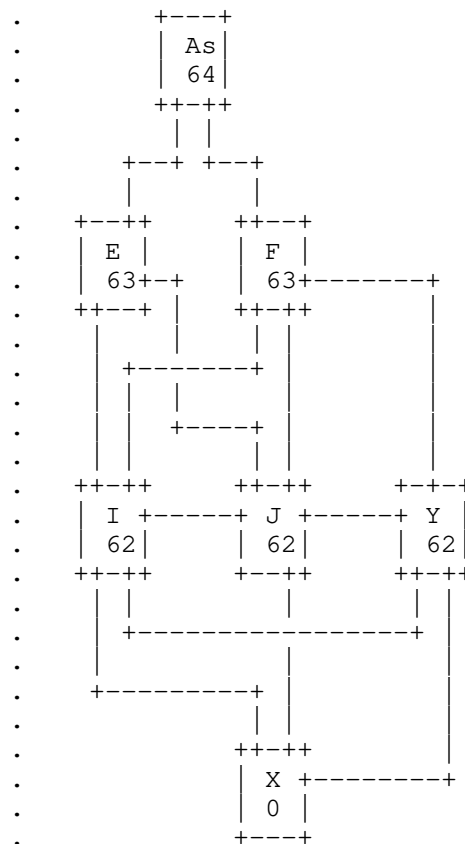


Figure 9: Generic ZTP Topology Autoconfigured

4.2.10. 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

Overload Bit MUST be respected in all according reachability computations. A node with overload bit set SHOULD NOT advertise any reachability prefixes southbound except locally hosted ones.

The leaf node SHOULD set the 'overload' bit on its node TIEs, since if the spine nodes were to forward traffic not meant for the local node, the leaf node does not have the topology information to prevent a routing/forwarding loop.

4.3.2. Optimized Route Computation on Leafs

Since the leafs do see only "one hop away" they do not need to run a full SPF but can simply gather prefix candidates from their neighbors and build the according routing table.

A leaf will have no N-TIEs except its own and optionally from its east-west neighbors. A leaf will have S-TIEs from its neighbors.

Instead of creating a network graph from its N-TIEs and neighbor's S-TIEs and then running an SPF, a leaf node can simply compute the minimum cost and next_hop_set to each leaf neighbor by examining its local interfaces, determining bi-directionality from the associated N-TIE, and specifying the neighbor's next_hop_set set and cost from the minimum cost local interfaces to that neighbor.

Then a leaf attaches prefixes as in Section 4.2.6 as well as the policy-guided prefixes as in Section 4.2.7.

4.3.3. Key/Value Store

4.3.3.1. Southbound

The protocol supports a southbound distribution of key-value pairs that can be used to e.g. distribute configuration information during topology bring-up. The KV S-TIEs can arrive from multiple nodes and hence need tie-breaking per key. We use the following rules

1. Only KV TIEs originated by a node to which the receiver has an adjacency are considered.
2. Within all valid KV S-TIEs containing the key, the value of the KV S-TIE for which the according node S-TIE is present, has the highest level and within the same level has highest originator ID

is preferred. If keys in the most preferred TIEs are overlapping, the behavior is undefined.

Observe that if a node goes down, the node south of it loses adjacencies to it and with that the KVs will be disregarded and on tie-break changes new KV re-advertised to prevent stale information being used by nodes further south. KV information in southbound direction is not result of independent computation of every node but a diffused computation.

4.3.3.2. Northbound

Certain use cases seem to necessitate distribution of essentially KV information that is generated in the leafs in the northbound direction. Such information is flooded in KV N-TIEs. Since the originator of northbound KV is preserved during northbound flooding, overlapping keys could be used. However, to omit further protocol complexity, only the value of the key in TIE tie-broken in same fashion as southbound KV TIEs is used.

4.3.4. Interactions with BFD

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

After RIFT 3-way hello adjacency convergence a BFD session MAY be formed automatically between the RIFT endpoints without further configuration.

In case RIFT loses 3-way hello adjacency, the BFD session should be brought down until 3-way adjacency is formed again.

In case established BFD session goes Down after it was Up, RIFT adjacency should be re-initialized from scratch.

In case of parallel links between nodes each link may run its own independent BFD session.

In case RIFT changes link identifiers both the hello as well as the BFD sessions will be brought down and back up again.

4.3.5. Fabric Bandwidth Balancing

A well understood problem in fabrics is that in case of link losses it would be ideal to rebalance how much traffic is offered to switches in the next layer based on the ingress and egress bandwidth they have. Current attempts rely mostly on specialized traffic

engineering via controller or leafs being aware of complete topology with according cost and complexity.

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

4.3.5.1. Northbound Direction

In a first step, a node can compare the amount of northbound bandwidth available to neighbors at the same level and modify metric on its advertised default route (or even other routes) to present a different distance leading to e.g. e.g. weighted ECMP forwarding on leafs. We call such a distance Bandwidth Adjusted Distance or BAD. This is best illustrated by a simple example.

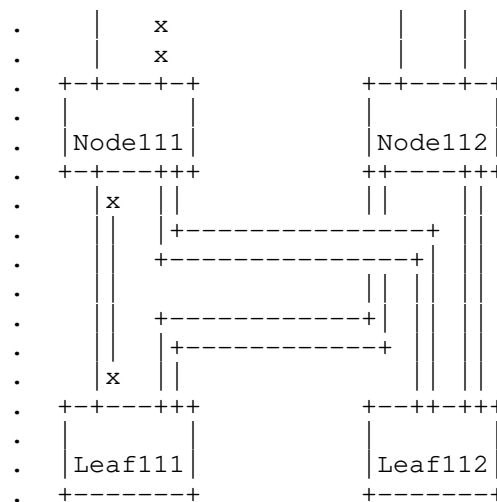


Figure 10: Balancing Bandwidth

All links in Figure 10 are assumed to have the same bandwidth for simplicity. Node 111 sees in the node S-TIE of 112 that Node 112 has twice the amount of bandwidth going northbound and therefore Node 111 will advertise its default route cost (BAD) as twice the default which without further failures would lead to Leaf 111 and Leaf 112 distributing 2/3 of the traffic to Node 111 and 1/3 to Node 112.

Further, in Figure 10 we assume that Leaf111 lost one of the parallel links to Node 111 and with that wants to push more traffic onto Node 112. This leads to local modification of the received BADs and each

node can choose the ratio here independently based on understanding of e.g. traffic distribution between E-W and N-S or queue occupancy. If we assume that 50% of the leaf's traffic is for Leaf112 and 50% exits northbound we would modify the BADs accordingly to the bandwidth available towards each of them and end in Leaf 111 with a weight of 4 to Node 111 and weight of 1 to Node 112 which gives us roughly 4/5 of the traffic going to Node 112.

Future version of this document will provide the precise algorithm to compute BADs from all other nodes at the same level using the same algorithm as Section 4.2.3.8 while ignoring overloaded nodes.

Observe that since BAD is only computed for default routes any disaggregated prefixes or PGP are not affected.

Observe further that a change in available bandwidth will only affect one level down in the fabric, i.e. blast radius of bandwidth changes is contained.

4.3.5.2. Southbound Direction

Due to its loop free properties a node could take during S-SPF into account the available bandwidth on the nodes in lower layers and modify the amount of traffic offered to next level's "southbound" nodes based as what it sees is the total achievable maximum flow through those nodes. It is worth observing that such computations will work better if standardized but does not have to be necessarily. As long the packet keeps on heading south it will take one of the available paths and arrive at the intended destination.

Future versions of this document will fill in more details.

4.3.6. Segment Routing Support with RIFT

Recently, alternative architecture to reuse labels as segment identifiers [I-D.ietf-spring-segment-routing] has gained traction and may present use cases in DC fabric that would justify its deployment. Such use cases will either precondition an assignment of a label per node (or other entities where the mechanisms are equivalent) or a global assignment and a knowledge of topology everywhere to compute segment stacks of interest. We deal with the two issues separately.

4.3.6.1. Global Segment Identifiers Assignment

Global segment identifiers are normally assumed to be provided by some kind of a centralized "controller" instance and distributed to other entities. This can be performed in RIFT by attaching a controller to the superspine nodes at the top of the fabric where the

whole topology is always visible, assign such identifiers and then distribute those via the KV mechanism towards all nodes so they can perform things like probing the fabric for failures using a stack of segments.

4.3.6.2. Distribution of Topology Information

Some segment routing use cases seem to precondition full knowledge of fabric topology in all nodes which can be performed albeit at the loss of one of highly desirable properties of RIFT, namely minimal blast radius. Basically, RIFT can function as a flat IGP by switching off its flooding scopes. All nodes will end up with full topology view and albeit the N-SPF and S-SPF are still performed based on RIFT rules, any computation with segment identifiers that needs full topology can use it.

Beside blast radius problem, excessive flooding may present significant load on implementations. RIFT can be extended beside the mechanism in Section 4.2.3.8 to provide an algorithm for globally optimized flooding minimalization should demand for such a use case solidify.

4.3.7. Leaf to Leaf Procedures

RIFT can optionally allow special leaf East-West adjacencies under additional set of rules. The leaf supporting those procedures MUST:

- advertise the LEAF_2_LEAF flag in node capabilities AND
- set the overload bit on all leaf's node TIEs AND
- flood only node's own north and south TIEs over E-W leaf adjacencies AND
- always use E-W leaf adjacency in both north as well as south computation AND
- install a discard route for any advertised aggregate in leaf's TIEs 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).

4.3.8. Other End-to-End Services

Losing full, flat topology information at every node will have an impact on some of the end-to-end network services. This is the price paid for minimal disturbance in case of failures and reduced flooding and memory requirements on nodes lower south in the level hierarchy.

4.3.9. Address Family and Multi Topology Considerations

Multi-Topology (MT) [RFC5120] and Multi-Instance (MI) [RFC6822] is 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 4.3.4 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 precondition that its nodes have reachable addresses albeit for operational purposes this is clearly desirable. Under normal operating conditions this can be easily achieved by e.g. injecting the node's loopback address into North Prefix TIEs.

Things get more interesting in case a node loses all its northbound adjacencies but is not at the top of the fabric. In such a case a node that detects that some other members at its level are advertising northbound adjacencies MAY inject its loopback address into southbound PGP TIE and become reachable "from the south" that way. Further, a solution may be implemented where based on e.g. a "well known" community such a southbound PGP is reflected at level 0 and advertised as northbound PGP again to allow for "reachability from the north" at the cost of additional flooding.

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

Based on the rules defined in Section 4.2.5, Section 4.2.3.7 and given presence of E-W links, RIFT can provide a one-hop protection of nodes that lost all their northbound links or in other complex link set failure scenarios. Section 5.4 explains the resulting behavior based on one such example.

5. Examples

5.1. Normal Operation

This section describes RIFT deployment in the example topology without any node or link failures. We disregard flooding reduction for simplicity's sake.

As first step, the following bi-directional adjacencies will be created (and any other links that do not fulfill LIE rules in Section 4.2.2 disregarded):

1. Spine 21 (PoD 0) to Node 111, Node 112, Node 121, and Node 122
2. Spine 22 (PoD 0) to Node 111, Node 112, Node 121, and Node 122
3. Node 111 to Leaf 111, Leaf 112
4. Node 112 to Leaf 111, Leaf 112
5. Node 121 to Leaf 121, Leaf 122
6. Node 122 to Leaf 121, Leaf 122

Consequently, N-TIEs would be originated by Node 111 and Node 112 and each set would be sent to both Spine 21 and Spine 22. N-TIEs also would be originated by Leaf 111 (w/ Prefix 111) and Leaf 112 (w/ Prefix 112 and the multi-homed prefix) and each set would be sent to Node 111 and Node 112. Node 111 and Node 112 would then flood these N-TIEs to Spine 21 and Spine 22.

Similarly, N-TIEs would be originated by Node 121 and Node 122 and each set would be sent to both Spine 21 and Spine 22. N-TIEs also would be originated by Leaf 121 (w/ Prefix 121 and the multi-homed prefix) and Leaf 122 (w/ Prefix 122) and each set would be sent to Node 121 and Node 122. Node 121 and Node 122 would then flood these N-TIEs to Spine 21 and Spine 22.

At this point both Spine 21 and Spine 22, as well as any controller to which they are connected, would have the complete network topology. At the same time, Node 111/112/121/122 hold only the N-ties of level 0 of their respective PoD. Leafs hold only their own N-TIEs.

S-TIEs with adjacencies and a default IP prefix would then be originated by Spine 21 and Spine 22 and each would be flooded to Node 111, Node 112, Node 121, and Node 122. Node 111, Node 112, Node 121, and Node 122 would each send the S-TIE from Spine 21 to Spine 22 and

the S-TIE from Spine 22 to Spine 21. (S-TIEs are reflected up to level from which they are received but they are NOT propagated southbound.)

An S Tie with a default IP prefix would be originated by Node 111 and Node 112 and each would be sent to Leaf 111 and Leaf 112. Leaf 111 and Leaf 112 would each send the S-TIE from Node 111 to Node 112 and the S-TIE from Node 112 to Node 111.

Similarly, an S Tie with a default IP prefix would be originated by Node 121 and Node 122 and each would be sent to Leaf 121 and Leaf 122. Leaf 121 and Leaf 122 would each send the S-TIE from Node 121 to Node 122 and the S-TIE from Node 122 to Node 121. At this point IP connectivity with maximum possible ECMP has been established between the leafs while constraining the amount of information held by each node to the minimum necessary for normal operation and dealing with failures.

5.2. Leaf Link Failure

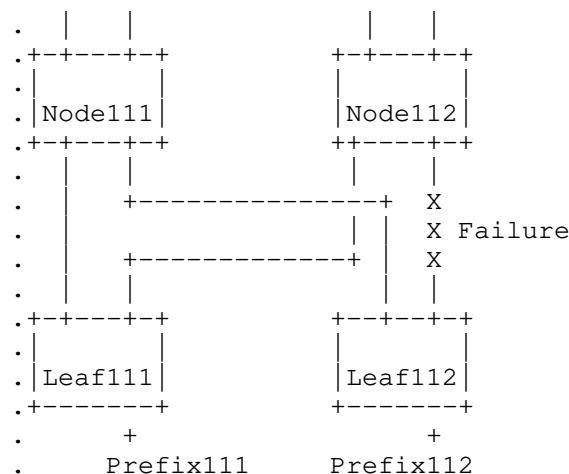


Figure 11: Single Leaf link failure

In case of a failing leaf link between node 112 and leaf 112 the link-state information will cause re-computation of the necessary SPF and the higher levels will stop forwarding towards prefix 112 through node 112. Only nodes 111 and 112, as well as both spines will see control traffic. Leaf 111 will receive a new S-TIE from node 112 and reflect back to node 111. Node 111 will de-aggregate prefix 111 and prefix 112 but we will not describe it further here since de-aggregation is emphasized in the next example. It is worth observing

however in this example that if leaf 111 would keep on forwarding traffic towards prefix 112 using the advertised south-bound default of node 112 the traffic would end up on spine 21 and spine 22 and cross back into pod 1 using node 111. This is arguably not as bad as black-holing present in the next example but clearly undesirable. Fortunately, de-aggregation prevents this type of behavior except for a transitory period of time.

5.3. Partitioned Fabric

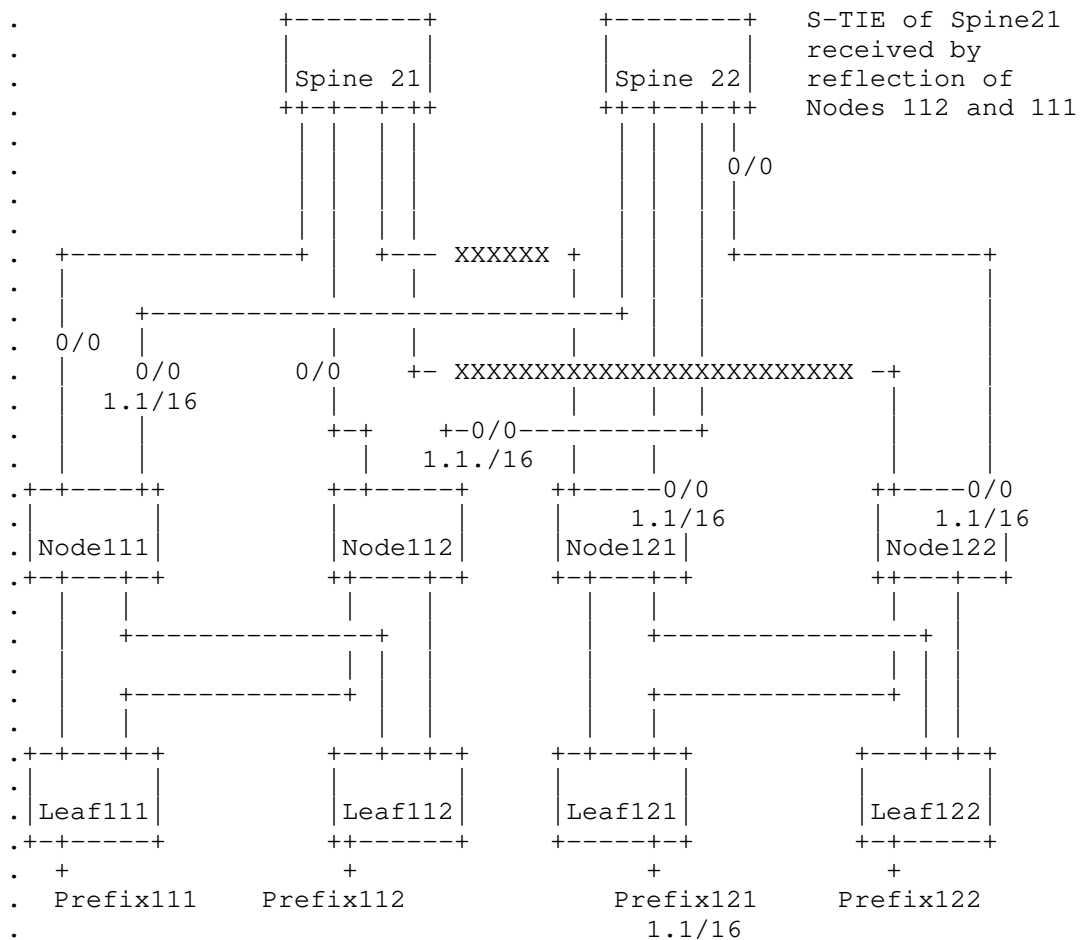


Figure 12: Fabric partition

Figure 12 shows the arguably most catastrophic but also the most interesting case. Spine 21 is completely severed from access to

Prefix 121 (we use in the figure 1.1/16 as example) by double link failure. However unlikely, if left unresolved, forwarding from leaf 111 and leaf 112 to prefix 121 would suffer 50% black-holing based on pure default route advertisements by spine 21 and spine 22.

The mechanism used to resolve this scenario is hinging on the distribution of southbound representation by spine 21 that is reflected by node 111 and node 112 to spine 22. Spine 22, having computed reachability to all prefixes in the network, advertises with the default route the ones that are reachable only via lower level neighbors that spine 21 does not show an adjacency to. That results in node 111 and node 112 obtaining a longest-prefix match to prefix 121 which leads through spine 22 and prevents black-holing through spine 21 still advertising the 0/0 aggregate only.

The prefix 121 advertised by spine 22 does not have to be propagated further towards leafs since they do no benefit from this information. Hence the amount of flooding is restricted to spine 21 reissuing its S-TIEs and reflection of those by node 111 and node 112. The resulting SPF in spine 22 issues a new prefix S-TIEs containing 1.1/16. None of the leafs become aware of the changes and the failure is constrained strictly to the level that became partitioned.

To finish with an example of the resulting sets computed using notation introduced in Section 4.2.8, spine 22 constructs the following sets:

|R = Prefix 111, Prefix 112, Prefix 121, Prefix 122

|H (for r=Prefix 111) = Node 111, Node 112

|H (for r=Prefix 112) = Node 111, Node 112

|H (for r=Prefix 121) = Node 121, Node 122

|H (for r=Prefix 122) = Node 121, Node 122

|A (for Spine 21) = Node 111, Node 112

With that and |H (for r=prefix 121) and |H (for r=prefix 122) being disjoint from |A (for spine 21), spine 22 will originate an S-TIE with prefix 121 and prefix 122, that is flooded to nodes 112, 121 and 122.

5.4. Northbound Partitioned Router and Optional East-West Links

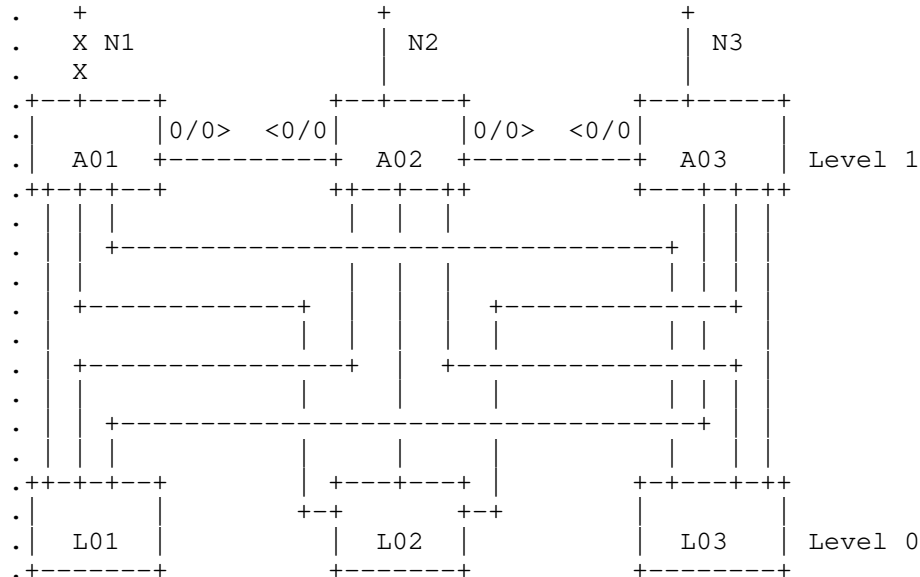


Figure 13: North Partitioned Router

Figure 13 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 4.2.5.1 A01 will compute northbound reachability by using the link A01 to A02 (whereas A02 will NOT use this link during N-SPF). Hence A01 will still advertise the default towards level 0 and route unidirectionally using the horizontal link. Moreover, based on Section 4.3.10 it may advertise its loopback address as south PGP to remain reachable "from the south" for operational purposes. This is necessary since A02 will NOT route towards A01 using the E-W link (doing otherwise may form routing loops).

As further consideration, the moment A02 loses link N2 the situation evolves again. A01 will have no more northbound reachability while still seeing A03 advertising northbound adjacencies in its south node tie. With that it will stop advertising a default route due to Section 4.2.3.7. Moreover, A02 may now inject its loopback address as south PGP.

6. Implementation and Operation: Further Details

6.1. Considerations for Leaf-Only Implementation

Ideally RIFT can be stretched out to the lowest level in the IP fabric to integrate ToRs or even servers. Since those entities would run as leafs only, it is worth to observe that a leaf only version is significantly simpler to implement and requires much less resources:

1. Under normal conditions, the leaf needs to support a multipath default route only. In worst partitioning case it has to be capable of accommodating all the leaf routes in its own POD to prevent black-holing.
2. Leaf nodes hold only their own N-TIEs and S-TIEs of Level 1 nodes they are connected to; so overall few in numbers.
3. Leaf node does not have to support flooding reduction and de-aggregation.
4. Unless optional leaf-2-leaf procedures are desired default route origination, S-TIE origination is unnecessary.

6.2. Adaptations to Other Proposed Data Center Topologies

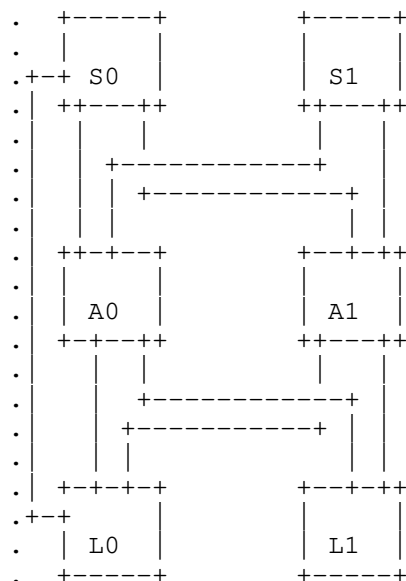


Figure 14: Level Shortcut

Strictly speaking, RIFT is not limited to Clos variations only. The protocol preconditions only a sense of 'compass rose direction' achieved by configuration (or derivation) of levels and other topologies are possible within this framework. So, conceptually, one could include leaf to leaf links and even shortcut between layers but certain requirements in Section 3 will not be met anymore. As an example, shortcutting levels illustrated in Figure 14 will lead either to suboptimal routing when L0 sends traffic to L1 (since using S0's default route will lead to the traffic being sent back to A0 or A1) or the leafs need each other's routes installed to understand that only A0 and A1 should be used to talk to each other.

Whether such modifications of topology constraints make sense is dependent on many technology variables and the exhausting treatment of the topic is definitely outside the scope of this document.

6.3. Originating Non-Default Route Southbound

Obviously, an implementation may choose to originate southbound instead of a strict default route (as described in Section 4.2.3.7) a shorter prefix P' but in such a scenario all addresses carried within the RIFT domain must be contained within P'.

7. Security Considerations

The protocol has provisions for nonces and can include authentication mechanisms in the future comparable to [RFC5709] and [RFC7987].

One can consider additionally 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 vanishes. Those attack vectors are not unique to RIFT. Given large memory footprints available today those attacks should be relatively benign. Otherwise a node can implement a strategy of e.g. discarding contents of all TIEs of nodes that were not present in the SPF tree over a certain 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.

Section 4.2.9 presents many attack vectors in untrusted environments, starting with nodes that oscillate their level offers to the possibility of a node offering a three way adjacency with the highest possible level value with a very long holdtime trying to put itself "on top of the lattice" and with that gaining access to the whole

southbound topology. Session authentication mechanisms are necessary in environments where this is possible.

8. Information Elements Schema

This section introduces the schema for information elements.

On schema changes that

1. change field numbers or
2. add new required fields or
3. remove fields or
4. change lists into sets, unions into structures or
5. change multiplicity of fields or
6. changes name of any field
7. change datatypes of any field or
8. changes default value of any field

major version of the schema MUST increase. All other changes MUST increase minor version within the same major.

Thrift serializer/deserializer MUST not discard optional, unknown fields but preserve and serialize them again when re-flooding whereas missing optional fields MAY be replaced with according default values if present.

All signed integer as forced by 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.

8.1. common.thrift

```
/**  
    Thrift file with common definitions for RIFT  
*/  
  
namespace * common
```

```
/** @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 of length long enough to accomodate prefix */
typedef binary   IPv6Address
/** @note MUST be interpreted in implementation as unsigned 16 bits */
typedef i16      UDPPortType
/** @note MUST be interpreted in implementation as unsigned 32 bits */
typedef i32      TIENrType
/** @note MUST be interpreted in implementation as unsigned 32 bits */
typedef i32      MTUSizeType
/** @note MUST be interpreted in implementation as unsigned 32 bits */
typedef i32      SeqNrType
/** @note MUST be interpreted in implementation as unsigned 32 bits */
typedef i32      LifeTimeInSecType
/** @note MUST be interpreted in implementation as unsigned 16 bits */
typedef i16      LevelType
/** @note MUST be interpreted in implementation as unsigned 32 bits */
typedef i32      PodType
/** @note MUST be interpreted in implementation as unsigned 16 bits */
typedef i16      VersionType
/** @note MUST be interpreted in implementation as unsigned 32 bits */
typedef i32      MetricType
/** @note MUST be interpreted in implementation as unsigned 32 bits */
typedef i32      BandwithInMegaBitsType
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 */
typedef i64      NonceType
/** adjacency holdtime */
typedef i16      HoldTimeInSecType

/** Flags indicating nodes behavior in case of ZTP and support
 * for special optimization procedures. It will force level to 'leaf_level'
 */
enum LeafIndications {
    leaf_only = 0,
    leaf_only_and_leaf_2_leaf_procedures = 1,
```

```
}

/** default bandwidth on a link */
const BandwidthInMegaBitsType default_bandwidth = 10
/** fixed leaf level when ZTP is not used */
const LevelType leaf_level = 0
const LevelType default_level = leaf_level
/** This MUST be used when node is configured as superspine in ZTP.
    This is kept reasonably low to allow for fast ZTP convergence on
    failures. */
const LevelType default_superspine_level = 24
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 = 0x70000000
/** any element with 0 distance will be ignored,
    * missing metrics will be replaced with default_distance
    */
const MetricType invalid_distance = 0
const bool overload_default = false
const bool flood_reduction_default = true
const HoldTimeInSecType default_holdtime = 3
/** by default LIE levels are ZTP offers */
const bool default_not_a_ztp_offer = false
/** 0 is illegal for SystemID */
const SystemIDType IllegalSystemID = 0
/** empty set of nodes */
const set<SystemIDType> empty_set_of_nodeids = {}

/** normalized bandwidth metric maximum, i.e. node with lowest northbound bandwidth
    * at its level uses this metric to advertise its default route */
const MetricType normalized_bw_metric_max = 0x1fff
/** normalized bandwidth metric minimum, i.e. node with highest northbound bandwidth
    * at its level uses this metric to advertise its default route */
const MetricType normalized_bw_metric_min = 0x00ff

/** default UDP port to run LIEs on */
const UDPPortType default_lie_udp_port = 6949
const UDPPortType default_tie_udp_flood_port = 6950

/** default MTU size to use */
const MTUSizeType default_mtu_size = 1400
/** default mcast is v4 224.0.1.150, we make it i64 to
    * help languages struggling with highest bit */
const i64 default_lie_v4_mcast_group = 3758096790
```

```
/** indicates whether the direction is northbound/east-west
 * or southbound */
enum TieDirectionType {
    Illegal          = 0,
    South            = 1,
    North            = 2,
    DirectionMaxValue = 3,
}

enum AddressFamilyType {
    Illegal          = 0,
    AddressFamilyMinValue = 1,
    IPv4             = 2,
    IPv6             = 3,
    AddressFamilyMaxValue = 4,
}

struct IPv4PrefixType {
    1: required IPv4Address    address;
    2: required PrefixLenType  prefixlen;
}

struct IPv6PrefixType {
    1: required IPv6Address    address;
    2: required PrefixLenType  prefixlen;
}

union IPAddressType {
    1: optional IPv4Address    ipv4address;
    2: optional IPv6Address    ipv6address;
}

union IPPrefixType {
    1: optional IPv4PrefixType  ipv4prefix;
    2: optional IPv6PrefixType  ipv6prefix;
}

enum TIETypeType {
    Illegal          = 0,
    TIETypeMinValue  = 1,
    /** first legal value */
    NodeTIEType      = 2,
    PrefixTIEType     = 3,
    TransitivePrefixTIEType = 4,
    PGPPrefixTIEType  = 5,
    KeyValueTIEType   = 6,
    TIETypeMaxValue   = 7,
}
```



```
/** @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
 */
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,
    /** advertised in S-TIEs */
    SouthPrefix            = 7,
    /** transitive southbound are least preferred */
    TransitiveSouthPrefix  = 8,
    RouteTypeMaxValue      = 9
}
```

8.2. encoding.thrift

```
/**
 * Thrift file for packet encodings for RIFT
 */

include "common.thrift"

/** represents protocol encoding schema major version */
const i32 protocol_major_version = 8
/** represents protocol encoding schema minor version */
const i32 protocol_minor_version = 0

/** common RIFT packet header */
struct PacketHeader {
```

```
1: required common.VersionType major_version = protocol_major_version;
2: required common.VersionType minor_version = protocol_minor_version;
/** this is the 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;
}

/** Community serves as community for PGP purposes */
struct Community {
    1: required i32 top;
    2: required i32 bottom;
}

/** Neighbor structure */
struct Neighbor {
    1: required common.SystemIDType originator;
    2: required common.LinkIDType remote_id;
}

/** Capabilities the node supports */
struct NodeCapabilities {
    /** can this node participate in flood reduction,
        only relevant at level > 0 */
    1: optional bool flood_reduction =
        common.flood_reduction_default;
    /** does this node restrict itself to be leaf only (in ZTP) and
        does it support leaf-2-leaf procedures */
    2: optional common.LeafIndications leaf_indications;
}

/** RIFT LIE packet

    @note this node's level is already included on the packet header */
struct LIEPacket {
    /** optional 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 */
    4: optional common.MTUSizeType link_mtu_size =
```

```

        common.default_mtu_size;
    /** this will reflect the neighbor once received to provid
        3-way connectivity */
    5: optional Neighbor                neighbor;
    6: optional common.PodType          pod = common.default_pod;
    /** optional nonce used for security computations */
    7: optional common.NonceType        nonce;
    /** optional node capabilities shown in the 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.
    */
    8: optional NodeCapabilities         capabilities;
    /** required holdtime of the adjacency, i.e. how much time
        MUST expire without LIE for the adjacency to drop
    */
    9: required common.HoldTimeInSecType holdtime =
        common.default_holdtime;
    /** indicates that the level on the LIE MUST NOT be used
        to derive a ZTP level by the receiving node. */
    10: optional bool                   not_a_ztp_offer =
        common.default_not_a_ztp_offer;
}

/** 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;
    /** more properties of the link can go in here */
}

/** 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 {
    /** indicates direction of the TIE */
    1: required common.TieDirectionType direction;
    /** indicates originator of the TIE */
    2: required common.SystemIDType      originator;
    3: required common.TIETypeType       tietype;
    4: required common.TIENrType         tie_nr;
}

```

```
/** Header of a TIE */
struct TIEHeader {
    2: required TIEID                tieid;
    3: required common.SeqNrType      seq_nr;
    /** lifetime expires down to 0 just like in ISIS */
    4: required common.LifeTimeInSecType lifetime;
}

/** A sorted TIDE packet, if unsorted, behavior is undefined */
struct TIDEPacket {
    /** all 00s marks starts */
    1: required TIEID                start_range;
    /** all FFs mark end */
    2: required TIEID                end_range;
    /** _sorted_ list of headers */
    3: required list<TIEHeader> headers;
}

/** A TIRE packet */
struct TIREPacket {
    1: required set<TIEHeader> headers;
}

/** Neighbor of a node */
struct NodeNeighborsTIEElement {
    2: 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 bandwidth to neighbor, this will be normally sum of the
     *   bandwidths of all the parallel links.
     */
    5: optional common.BandwidthInMegaBitsType bandwidth =
        common.default_bandwidth;
}

/** Flags the node sets */
struct NodeFlags {
    /** 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
    * capabilities values do not match or
    * flags values do not match or
    * neighbors repeat with different values or
    * visible in same level/having partition upper do not match
    the behavior is undefined and a warning SHOULD be generated.
    Neighbors can be distributed across multiple TIEs however if
    the sets are disjoint.

    @note: observe that absence of fields implies defined defaults
*/
struct NodeTIEElement {
    1: required common.LevelType          level;
    /** if neighbor systemID repeats in other node TIEs of same node
        the behavior is undefined. Equivalent to  $|A_{(n,s)}(N)$  in spec. */
    2: required map<common.SystemIDType,
        NodeNeighborsTIEElement>         neighbors;
    3: optional NodeCapabilities           capabilities;
    4: optional NodeFlags                  flags;
    /** optional node name for easier operations */
    5: optional string                     name;

    /** Nodes seen an the same level through reflection through nodes
        having backlink to both nodes. They are equivalent to  $|V(N)$  in
        future specifications. Ignored in Node S-TIEs if present.
    */
    6: optional set<common.SystemIDType>   visible_in_same_level
        = common.empty_set_of_nodeids;
    /** Non-overloaded nodes in  $|V$  seen as attached to another north
        * level partition due to the fact that some nodes in its  $|V$  have
        * adjacencies to higher level nodes that this node doesn't see.
        * This may be used in the computation at higher levels to prevent
        * blackholing. Ignored in Node S-TIEs if present.
        * Equivalent to  $|PUL(N)$  in spec. */
    7: optional set<common.SystemIDType>   same_level_unknown_north_partitions
        = common.empty_set_of_nodeids;
}

struct PrefixAttributes {
    /** Observe that in default metric case the node is supposed to advertise
        * metric calculated from comparison of bandwidths at all nodes at its
        * level. */
    2: required common.MetricType          metric = common.default_distance;
}

```

```
/** multiple 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;
}

/** keys with their values */
struct KeyValueTIEElement {
    /** 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. enum common.TIETypeType
    in TIEID indicates which elements MUST be present
    in the TIEElement. In case of mismatch the unexpected
    elements MUST be ignored.
    */
union TIEElement {
    /** in case of enum common.TIETypeType.NodeTIEType */
    1: optional NodeTIEElement node;
    /** in case of enum common.TIETypeType.PrefixTIEType */
    2: optional PrefixTIEElement prefixes;
    /** transitive prefixes (always southbound) which SHOULD be propagated
        * southwards towards lower levels to heal
        * pathological upper level partitioning, otherwise
        * blackholes may occur. MUST NOT be advertised within a North TIE.
        */
    3: optional PrefixTIEElement transitive_prefixes;
    4: optional KeyValueTIEElement keyvalues;
    /** @todo: policy guided prefixes */
}

/** @todo: flood header separately in UDP to allow caching to TIEs
    while changing lifetime?
    */
struct TIEPacket {
    1: required TIEHeader header;
    2: required TIEElement element;
}

union PacketContent {
    1: optional LIEPacket lie;
    2: optional TIDEPacket tide;
    3: optional TIREPacket tire;
    4: optional TIEPacket tie;
```

```
}

/** protocol packet structure */
struct ProtocolPacket {
    1: required PacketHeader header;
    2: required PacketContent content;
}
```

9. IANA Considerations

This specification will request at an opportune time multiple registry points to exchange protocol packets in a standardized way, amongst them multicast address assignments and standard port numbers. The schema itself defines many values and codepoints which can be considered registries themselves.

10. Acknowledgments

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 here. Adrian Farrel, Joel Halpern and Jeffrey Zhang 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. 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 on a (clean) napkin in Singapore.

11. References

11.1. Normative References

- [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.

- [RFC1142] Oran, D., Ed., "OSI IS-IS Intra-domain Routing Protocol", RFC 1142, DOI 10.17487/RFC1142, February 1990, <<https://www.rfc-editor.org/info/rfc1142>>.
- [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>>.
- [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>>.
- [RFC4655] Farrel, A., Vasseur, J., and J. Ash, "A Path Computation Element (PCE)-Based Architecture", RFC 4655, DOI 10.17487/RFC4655, August 2006, <<https://www.rfc-editor.org/info/rfc4655>>.
- [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>>.
- [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>>.
- [RFC6234] Eastlake 3rd, D. and T. Hansen, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)", RFC 6234, DOI 10.17487/RFC6234, May 2011, <<https://www.rfc-editor.org/info/rfc6234>>.
- [RFC6822] Previdi, S., Ed., Ginsberg, L., Shand, M., Roy, A., and D. Ward, "IS-IS Multi-Instance", RFC 6822, DOI 10.17487/RFC6822, December 2012, <<https://www.rfc-editor.org/info/rfc6822>>.
- [RFC7855] Previdi, S., Ed., Filsfils, C., Ed., Decraene, B., Litkowski, S., Horneffer, M., and R. Shakir, "Source Packet Routing in Networking (SPRING) Problem Statement and Requirements", RFC 7855, DOI 10.17487/RFC7855, May 2016, <<https://www.rfc-editor.org/info/rfc7855>>.
- [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>>.
- [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>>.

11.2. 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.
- [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.
- [I-D.ietf-spring-segment-routing]
Filsfils, C., Previdi, S., Ginsberg, L., Decraene, B., Litkowski, S., and R. Shakir, "Segment Routing Architecture", draft-ietf-spring-segment-routing-15 (work in progress), January 2018.
- [MAKSIC2013]
Maksic et al., N., "Improving Utilization of Data Center Networks", IEEE Communications Magazine, Nov 2013.
- [PROTOBUF]
Google, Inc., "Protocol Buffers", <https://developers.google.com/protocol-buffers>.
- [QUIC] Iyengar et al., J., "QUIC: A UDP-Based Multiplexed and Secure Transport", 2016.
- [VAHDAT08]
Al-Fares, M., Loukissas, A., and A. Vahdat, "A Scalable, Commodity Data Center Network Architecture", SIGCOMM , 2008.

Authors' Addresses

Tony Przygienda (editor)
Juniper Networks
1194 N. Mathilda Ave
Sunnyvale, CA 94089
US

Email: prz@juniper.net

Alankar Sharma
Comcast
1800 Bishops Gate Blvd
Mount Laurel, NJ 08054
US

Email: Alankar_Sharma@comcast.com

Alia Atlas
Juniper Networks
10 Technology Park Drive
Westford, MA 01886
US

Email: akatlas@juniper.net

John Drake
Juniper Networks
1194 N. Mathilda Ave
Sunnyvale, CA 94089
US

Email: jdrake@juniper.net

RIFT WG
Internet-Draft
Intended status: Standards Track
Expires: November 5, 2019

Zheng. Zhang
Yuehua. Wei
ZTE Corporation
Shaowen. Ma
Mellanox
Xufeng. Liu
Volta Networks
May 4, 2019

RIFT YANG Model
draft-zhang-rift-yang-02

Abstract

This document defines a YANG data model for the configuration and management of RIFT Protocol.

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 5, 2019.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of

the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	2
2. Design of the Data Model	2
3. RIFT configuration	7
4. RIFT State	7
5. RPC	7
6. Notifications	7
7. RIFT YANG model	7
8. Security Considerations	19
9. IANA Considerations	20
10. Contributors	20
11. Normative References	20
Authors' Addresses	22

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]

2. Design of the Data Model

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 neighbor state will be added in later version. The container "rift" is the top level container in this data model. The presence of this container is expected to enable RIFT protocol functionality.

```

module: ietf-rift
  augment /rt:routing/rt:control-plane-protocols/rt:control-plane-protocol:
    +--rw rift!
      +--rw node-info
        +--rw level?                level
        +--rw systemid             systemid
        +--rw name?                string
        +--rw pod?                 uint32
        +--rw overload?            boolean
        +--rw flood-reducing?      boolean {flood-reducing}?
        +--rw hierarchy-indications? enumeration
        +--rw interfaces* [local-id]

```

```

+--rw local-id                               linkidtype
+--rw name?                                 if:interface-ref
+--rw if-index?                             if:interface-ref
+--rw bfd?                                  boolean {bfd}?
+--rw direction-type?                       enumeration
+--rw you_are_flood_repeater?              boolean
+--rw not_a_ztp_offer?                     boolean
+--rw flood-port?                           inet:port-number
+--rw lie-rx-port?                         inet:port-number
+--rw holdtime?                             rt-types:timer-value-seconds16
+--rw local-nonce?                         uint16
+--rw (algorithgm-type)?
|   +--:(spf)
|   +--:(all-path)
+--ro hal?                                  level
+--ro vol-list
|   +--ro vol* [systemid]
|       +--ro offered-level?                level
|       +--ro level?                       level
|       +--ro systemid                     systemid
|       +--ro name?                        string
|       +--ro pod?                        uint32
|       +--ro overload?                   boolean
|       +--ro flood-reducing?             boolean {flood-reducing}?
|       +--ro hierarchy-indications?      enumeration
|       +--ro interfaces* [local-id]
|           +--ro local-id                 linkidtype
|           +--ro name?                   if:interface-ref
|           +--ro if-index?               if:interface-ref
|           +--ro bfd?                    boolean {bfd}?
|           +--ro direction-type?         enumeration
|           +--ro you_are_flood_repeater? boolean
|           +--ro not_a_ztp_offer?        boolean
|           +--ro flood-port?             inet:port-number
|           +--ro lie-rx-port?            inet:port-number
|           +--ro holdtime?               rt-types:timer-value-seconds1
|
|           +--ro local-nonce?             uint16
+--ro miscabled-links*                     linkidtype
+--ro neighbor
+--ro nbrs* [systemid remote-id]
|   +--ro level?                          level
|   +--ro systemid                        systemid
|   +--ro name?                          string
|   +--ro pod?                          uint32
|   +--ro overload?                      boolean
|   +--ro flood-reducing?                boolean {flood-reducing}?
|   +--ro hierarchy-indications?         enumeration
|   +--ro interfaces* [local-id]

```

```

    +---ro local-id                linkidtype
    +---ro name?                   if:interface-ref
    +---ro if-index?               if:interface-ref
    +---ro bfd?                    boolean {bfd}?
    +---ro direction-type?         enumeration
    +---ro you_are_flood_repeater? boolean
    +---ro not_a_ztp_offer?        boolean
    +---ro flood-port?             inet:port-number
    +---ro lie-rx-port?            inet:port-number
    +---ro holdtime?               rt-types:timer-value-seconds16
    +---ro local-nonce?            uint16
+---ro remote-id                  uint32
+---ro local-id?                  uint32
+---ro distance?                  uint32
+---ro remote-nonce?              uint16
+---ro miscabled-links*           linkidtype
+---ro database
  +---ro ties* [tie-index]
    +---ro tie-index              uint32
    +---ro database-tie
      +---ro originator?          systemid
      +---ro direction-type?      enumeration
      +---ro tie-number?          uint32
      +---ro seq?                  uint32
      +---ro lifetime?            uint16
      +---ro tie-node
        +---ro level?              level
        +---ro systemid            systemid
        +---ro name?               string
        +---ro pod?                uint32
        +---ro overload?           boolean
        +---ro flood-reducing?     boolean {flood-reducing}?
        +---ro hierarchy-indications? enumeration
        +---ro interfaces* [local-id]
          +---ro local-id          linkidtype
          +---ro name?              if:interface-ref
          +---ro if-index?          if:interface-ref
          +---ro bfd?                boolean {bfd}?
          +---ro direction-type?    enumeration
          +---ro you_are_flood_repeater? boolean
          +---ro not_a_ztp_offer?   boolean
          +---ro flood-port?        inet:port-number
          +---ro lie-rx-port?       inet:port-number
          +---ro holdtime?          rt-types:timer-value-secon
ds16
    +---ro local-nonce?            uint16
  +---ro tie-prefix
    +---ro prefixes
      +---ro prefix?              inet:ip-prefix

```

```

+---ro metric?                uint32
+---ro tag?                   uint64
+---ro monotonic_clock?      PrefixSequenceType
+---ro from-link?            linkidtype
+---ro positive_disaggregation_prefixes
+---ro prefix?               inet:ip-prefix
+---ro metric?               uint32
+---ro tag?                  uint64
+---ro monotonic_clock?      PrefixSequenceType
+---ro from-link?            linkidtype
+---ro negative_disaggregation_prefixes
+---ro prefix?               inet:ip-prefix
+---ro metric?               uint32
+---ro tag?                  uint64
+---ro monotonic_clock?      PrefixSequenceType
+---ro from-link?            linkidtype
+---ro external_prefixes
+---ro prefix?               inet:ip-prefix
+---ro metric?               uint32
+---ro tag?                  uint64
+---ro monotonic_clock?      PrefixSequenceType
+---ro from-link?            linkidtype
+---ro kvs
+---ro key?                   uint16
+---ro value?                 uint32
+---rw kv-store
+---rw kvs* [kvs-index]
+---rw kvs-index              uint32
+---rw kvs-tie
+---rw originator?            systemid
+---rw direction-type?        enumeration
+---rw tie-number?            uint32
+---rw key?                   uint16
+---rw value?                 uint32

notifications:
+---n error-set
+---ro tie-level-error
+---ro originator?            systemid
+---ro direction-type?        enumeration
+---ro tie-number?            uint32
+---ro seq?                   uint32
+---ro lifetime?              uint16
+---ro tie-node
+---ro level?                  level
+---ro systemid                systemid
+---ro name?                   string
+---ro pod?                    uint32

```



```

+--ro overload?                boolean
+--ro flood-reducing?          boolean {flood-reducing}?
+--ro hierarchy-indications?   enumeration
+--ro interfaces* [local-id]
  +--ro local-id                linkidtype
  +--ro name?                   if:interface-ref
  +--ro if-index?               if:interface-ref
  +--ro bfd?                    boolean {bfd}?
  +--ro direction-type?        enumeration
  +--ro you_are_flood_repeater? boolean
  +--ro not_a_ztp_offer?        boolean
  +--ro flood-port?             inet:port-number
  +--ro lie-rx-port?            inet:port-number
  +--ro holdtime?               rt-types:timer-value-seconds16
  +--ro local-nonce?            uint16
+--ro tie-prefix
  +--ro prefixes
    +--ro prefix?               inet:ip-prefix
    +--ro metric?               uint32
    +--ro tag?                  uint64
    +--ro monotonic_clock?      PrefixSequenceType
    +--ro from-link?            linkidtype
  +--ro positive_disaggregation_prefixes
    +--ro prefix?               inet:ip-prefix
    +--ro metric?               uint32
    +--ro tag?                  uint64
    +--ro monotonic_clock?      PrefixSequenceType
    +--ro from-link?            linkidtype
  +--ro negative_disaggregation_prefixes
    +--ro prefix?               inet:ip-prefix
    +--ro metric?               uint32
    +--ro tag?                  uint64
    +--ro monotonic_clock?      PrefixSequenceType
    +--ro from-link?            linkidtype
  +--ro external_prefixes
    +--ro prefix?               inet:ip-prefix
    +--ro metric?               uint32
    +--ro tag?                  uint64
    +--ro monotonic_clock?      PrefixSequenceType
    +--ro from-link?            linkidtype
+--ro kvs
  +--ro key?                    uint16
  +--ro value?                  uint32
+--ro nbr-error
  +--ro nbrs* [systemid remote-id]
    +--ro level?                level
    +--ro systemid               systemid
    +--ro name?                  string

```

```

+--ro pod?                               uint32
+--ro overload?                           boolean
+--ro flood-reducing?                     boolean {flood-reducing}?
+--ro hierarchy-indications?              enumeration
+--ro interfaces* [local-id]
|   +--ro local-id                         linkidtype
|   +--ro name?                           if:interface-ref
|   +--ro if-index?                       if:interface-ref
|   +--ro bfd?                            boolean {bfd}?
|   +--ro direction-type?                 enumeration
|   +--ro you_are_flood_repeater?         boolean
|   +--ro not_a_ztp_offer?                boolean
|   +--ro flood-port?                     inet:port-number
|   +--ro lie-rx-port?                     inet:port-number
|   +--ro holdtime?                       rt-types:timer-value-seconds16
|   +--ro local-nonce?                     uint16
+--ro remote-id                           uint32
+--ro local-id?                           uint32
+--ro distance?                           uint32
+--ro remote-nonce?                       uint16
+--ro miscabled-links*                     linkidtype

```

3. RIFT configuration

RIFT configurations require node base information configurations. Some features can be used to enhance protocol, such as BFD, flooding-reducing, community attribute.

4. RIFT State

RIFT states are composed of RIFT node state, neighbor state, database.

5. RPC

TBD.

6. Notifications

Unexpected TIE and neighbor's layer error should be notified.

7. RIFT YANG model

```

<CODE BEGINS> file "ietf-rift.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 "RFC6991";
}

import ietf-routing {
  prefix "rt";
  reference "RFC8349";
}

import ietf-interfaces {
  prefix "if";
  reference "RFC7223";
}

import ietf-routing-types {
  prefix "rt-types";
  reference "RFC8294";
}

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:zzhang_ietf@hotmail.com>

  Editor:     Yuehua Wei
               <mailto:wei.yuehua@zte.com.cn>

  Editor:     Shaowen Ma
               <mailto:mashaowen@gmail.com>

  Editor:     Xufeng Liu
               <mailto:Xufeng_Liu@jabil.com>";

description
  "The module defines the YANG definitions for RIFT.

  Copyright (c) 2018 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 (<http://trustee.ietf.org/license-info>). This version of this YANG module is part of RFC 3618; see the RFC itself for full legal notices.";

```
revision 2019-05-05 {
  description "Initial revision.";
  reference
    "RFC XXXX: A YANG Data Model for RIFT.
    draft-ietf-rift-rift: RIFT: Routing in Fat Trees.";
}

/*
 * Features
 */

/*feature overload {
  description "A node with overload bit set SHOULD NOT advertise any reach
ability
                                prefixes southbound except locally hosted ones. The leaf no
de SHOULD
                                set the 'overload' bit on its node TIEs.";
}*/

feature bfd {
  description "Support BFD (RFC5881) function to react quickly to link fai
lures.";
}

feature flood-reducing {
  description "Support flood reducing function defined in section 4.2.3.8.
";
}

feature policy {
  description "Support policy guide information.";
}

typedef systemid {
  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,
    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 linkidtype {
    type uint32;
    description
        "This type defines the link id of an interface.";
}

typedef PrefixSequenceType {
    type uint64;
    description
        "This type defines the link id of an interface.";
}

/*
 * Identity
 */
identity rift {
    base rt:routing-protocol;
    description "Identity for the RIFT routing protocol.";
}

/*
 * Groupings
 */
grouping node-key {
    leaf systemid {
        type systemid;
        mandatory true;
        description "Each node is identified via a SystemID which is 64 bits
wide.";
    }
    description "The key information used to distinguish a node.";
}

grouping base-node-info {
    leaf level {
        type level;
        description "The level of this node.";
    }
    uses node-key;

    leaf name {
```

```

        type string;
        description "The name of this node. It won't be used as the key of n
ode,
                                just used for description.";
    }
    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 distingu
ish
                                them and use PoD #0 to denote 'undefined' PoD.";
    }
    leaf overload {
        type boolean;
        description "If the overload bit in TIEs should be set.";
    }
    leaf flood-reducing {
        if-feature flood-reducing;
        type boolean;
        description "If the node support flood reducing function defined in
section 4.2.3.8.";
    }
    uses hierarchy-indications;
    uses interface;

    description "The base information of a node.";
}

grouping neighbor {
    leaf remote-id {
        type uint32;
        description "The remote-id to reach this neighbor.";
    }
    leaf local-id {
        type uint32;
        description "The local-id of link connect to this neighbor.";
    }
    leaf distance {
        type uint32;
        description "The cost value to arrive this neighbor.";
    }
    leaf remote-nonce {
        type uint16;
        description "Remote Nonce of the adjacency as received
in LIEs. In case of LIE packet this MUST
correspond to the value in the serialized
object otherwise the packet MUST be discarded.";
    }
    leaf-list miscabled-links {

```

```

        type linkidtype;
        description "List of miscabled links.";
    }
    description "The neighbor information.";
}

grouping hierarchy-indications {
    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'.";
            }
        }
        description "The hierarchy indications of this node.";
    }
    description "The hierarchy indications of this node.";
}

grouping node {
    uses base-node-info;
    uses algorithm;

    leaf hal {
        type level;
        config false;
        description "The highest defined level value seen from all
            valid level offers received.";
    }
    container vol-list {
        config false;
        list vol {
            key "systemid";
            leaf offered-level {
                type level;
                description "The level type value offered by this neighbor."
            }
        }
        uses base-node-info;
        description "The valid offered level information.";
    }
    description "The valid offered level information.";
}
;

```

```
    leaf-list miscabled-links {
        type linkidtype;
        config false;
        description
            "List of miscabled links.";
    }
    description "The information of local node. Includes base information,
        configurable parameters and features.";
}

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.";
            }
        }
        description "The type of a link.";
    }
    description "The type of a link.";
}

grouping interface {
    list interfaces {
        key "local-id";
        leaf local-id {
            type linkidtype;
            mandatory true;
            description "The local id of this interface.";
        }
        leaf name {
            type if:interface-ref;
            description "The interface's name.";
        }
        leaf if-index {
            type if:interface-ref;
        }
    }
}
```



```
        description "The index of this interface.";
    }
    leaf bfd {
        if-feature bfd;
        type boolean;
        description "If BFD function is enabled to react link failures
                    after neighbor's detection.";
    }
    uses direction-type;

    leaf you_are_flood_repeater {
        type boolean;
        description "If the neighbor on this link is flooding repeater."
;
    }
    leaf not_a_ztp_offer {
        type boolean;
        description "If the neighbor on this link offers ZTP.";
    }
    leaf flood-port {
        type inet:port-number;
        description "The flooding port.";
    }
    leaf lie-rx-port {
        type inet:port-number;
        description "The port of LIE packet receiving.";
    }
    leaf holdtime {
        type rt-types:timer-value-seconds16;
        units seconds;
        description "The holding time of this adjacency.";
    }
    leaf local-nonce {
        type uint16;
        description "Local Nonce of the adjacency as advertised in LIEs.
                    In case of LIE packet this MUST correspond to the
                    value in the serialized object otherwise the packet
                    MUST be discarded.";
    }

    description "The interface information on this node.";
}
description "The interface information.";
}

grouping prefix-info {
    leaf prefix {
        type inet:ip-prefix;
        description "The prefix information.";
```

```
    }
    leaf metric {
        type uint32;
        description "The metric of this prefix.";
    }
    leaf tag {
        type uint64;
        description "The tag of this prefix.";
    }
    leaf monotonic_clock {
        type PrefixSequenceType;
        description "The monotonic clock for mobile addresses.";
    }
    leaf from-link {
        type linkidtype;
        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 prefix.";
}

grouping tie-id {
    leaf originator {
        type systemid;
        description "The originator's systemid of this TIE.";
    }
    uses direction-type;
    leaf tie-number {
        type uint32;
        description "The number of this TIE";
    }
    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 identification information.";
}

grouping key-value {
    leaf key {
        type uint16;
        description "The type of key value combination.";
    }
    leaf value {
        type uint32;
        description "The value of key value combination.";
    }
}
```

```

        description "The key-value store information.";
    }

    grouping tie-info {
        leaf seq {
            type uint32;
            description "The sequence number of a TIE.";
        }
        leaf lifetime {
            type uint16 {
                range "1 .. 65535";
            }
            description "The lifetime of a TIE.";
        }
    }

    container tie-node {
        uses base-node-info;
        description "The node element information in this TIE.";
    }

    container tie-prefix {
        container prefixes {
            uses prefix-info;
            description "It is the prefixes TIE element.";
        }
        container positive_disaggregation_prefixes {
            uses prefix-info;
            description "It is the positive disaggregation prefixes TIE element.";
        }
        container negative_disaggregation_prefixes {
            uses prefix-info;
            description "It is the negative disaggregation prefixes element.";
        }
        container external_prefixes {
            uses prefix-info;
            description "It is the external prefixes element.";
        }
        description "The prefix information in this TIE.";
    }

    container kvs {
        uses key-value;
        description "The key/values in the database.";
    }

    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 TIE
        info is used to indicate the state of this TIE. When the
        type of this TIE is set to 'node', the node-element is

```

```

        making sense. When the type of this TIE is set to other
        types except for 'node', the prefix-info is making sense.";
    }

    grouping algorithm {
        choice algorighm-type {
            case spf {
                description "The algorithm is SPF.";
            }
            case all-path {
                description "The algorithm is all-path.";
            }
            description "The possible algorithm types.";
        }
        description "The computation algorithm types.";
    }

    /*
     * Data nodes
     */
    augment "/rt:routing/rt:control-plane-protocols/rt:control-plane-protocol" {
        when "derived-from-or-self(rt:type, 'rift:rft')" {
            description "This augment is only valid for a routing protocol insta
nce of RIFT.";
        }
        description "RIFT ( Routing in Fat Trees ) YANG model.";
        container rift {
            presence "Container for RIFT protocol.";
            description "RIFT configuration data.";

            container node-info {
                description "The node information about RIFT.";
                uses node;
            }
            container neighbor {
                config false;
                list nbrs {
                    key "systemid remote-id";
                    uses base-node-info;
                    uses neighbor;
                    description "The information of a neighbor.";
                }
                description "The neighbor's information.";
            } //neighbor

            container database {
                config false;
                list ties {
                    key "tie-index";

```

```

        leaf tie-index {
            type uint32;
            description "The index of a TIE.";
        }
        container database-tie {
            uses tie-id;
            uses tie-info;

            description "The TIEs in the database.";
        }
        description "The detail information of a TIE.";
    }
    description "The TIEs information in database.";
} // database

container kv-store {
    list kvs {
        key "kvs-index";
        leaf kvs-index {
            type uint32;
            description "The index of a kv pair.";
        }

        container kvs-tie {
            uses tie-id;
            uses key-value;
            description "The TIEs in the kv-store.";
        }
        description "The information used to distinguish a Key/Value
pair.
ent is
er values
When the type of kv is set to 'node', node-elem
making sense. When the type of kv is set to oth
except 'node', prefix-info is making sense.";
    }
    description "The Key/Value store information.";
} // kv-store

} // rift
} // augment

/*
 * RPCs
 */

/*
 * Notifications
 */

```

```
notification error-set {
  description "The errors notification of RIFT.";
  container tie-level-error {
    uses tie-id;
    uses tie-info;
    description "The level is undefined in the LIEs.";
  }
  container nbr-error {
    list nbrs {
      key "systemid remote-id";
      uses base-node-info;
      uses neighbor;
      description "The information of a neighbor.";
    }
    description "The neighbor errors set.";
  }
}
}
<CODE ENDS>
```

8. 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 [RFC5246].

The NETCONF access control model [RFC6536] 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.

The RPC operations in this YANG module may be considered sensitive or vulnerable in some network environments. It is thus important to control access to these operations.

9. IANA Considerations

The IANA is requested to assign two new URIs from the IETF XML registry ([RFC3688]). Authors are suggesting the following URI:

URI: urn:ietf:params:xml:ns:yang:ietf-rift

Registrant Contact: RIFT WG

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

10. Contributors

The authors would like to thank Tony Przygienda, Benchong Xu (xu.benchong@zte.com.cn), for their review and valuable contributions.

11. Normative References

- [I-D.ietf-rift-rift]
Team, T., "RIFT: Routing in Fat Trees", draft-ietf-rift-rift-05 (work in progress), April 2019.
- [I-D.ietf-rtgwg-policy-model]
Qu, Y., Tantsura, J., Lindem, A., and X. Liu, "A YANG Data Model for Routing Policy Management", draft-ietf-rtgwg-policy-model-06 (work in progress), March 2019.
- [RFC3688] Mealling, M., "The IETF XML Registry", BCP 81, RFC 3688, DOI 10.17487/RFC3688, January 2004, <<https://www.rfc-editor.org/info/rfc3688>>.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <<https://www.rfc-editor.org/info/rfc5246>>.

- [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>>.
- [RFC6087] Bierman, A., "Guidelines for Authors and Reviewers of YANG Data Model Documents", RFC 6087, DOI 10.17487/RFC6087, January 2011, <<https://www.rfc-editor.org/info/rfc6087>>.
- [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>>.
- [RFC6536] Bierman, A. and M. Bjorklund, "Network Configuration Protocol (NETCONF) Access Control Model", RFC 6536, DOI 10.17487/RFC6536, March 2012, <<https://www.rfc-editor.org/info/rfc6536>>.
- [RFC6991] Schoenwaelder, J., Ed., "Common YANG Data Types", RFC 6991, DOI 10.17487/RFC6991, July 2013, <<https://www.rfc-editor.org/info/rfc6991>>.
- [RFC7223] Bjorklund, M., "A YANG Data Model for Interface Management", RFC 7223, DOI 10.17487/RFC7223, May 2014, <<https://www.rfc-editor.org/info/rfc7223>>.
- [RFC7277] Bjorklund, M., "A YANG Data Model for IP Management", RFC 7277, DOI 10.17487/RFC7277, June 2014, <<https://www.rfc-editor.org/info/rfc7277>>.
- [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>>.
- [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>>.
- [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>>.

- [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>>.
- [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>>.
- [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>>.

Authors' Addresses

Zheng Zhang
ZTE Corporation

Email: zzhang_ietf@hotmail.com

Yuehua Wei
ZTE Corporation

Email: wei.yuehua@zte.com.cn

Shaowen Ma
Mellanox

Email: mashaowen@gmail.com

Xufeng Liu
Volta Networks

Email: xufeng.liu.ietf@gmail.com

BIER
Internet-Draft
Intended status: Standards Track
Expires: September 6, 2018

Zhaohui. Zhang
Shaowen. Ma
Juniper Networks
Zheng. Zhang
ZTE Corporation
March 5, 2018

Supporting BIER with RIFT
draft-zzhang-bier-rift-00

Abstract

This document specifies extensions to RIFT protocol to support BIER.

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

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 September 6, 2018.

Copyright Notice

Copyright (c) 2018 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect

to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Terminologies	2
2. Introduction	2
3. Advertising BIER Information For non-MPLS Encapsulation . . .	3
4. Advertising BIER Information Northbound	4
5. Advertising BIER Information Southbound	4
5.1. Local BIER Information	4
5.2. Proxied BFR-ID Ranges	4
6. Information Elements Schema	5
6.1. bier.thrift	5
6.2. Additions to encoding.thrift	6
7. IANA Considerations	6
8. Acknowledgements	6
9. References	6
9.1. Normative References	6
9.2. Informative References	6
Authors' Addresses	7

1. Terminologies

Familiarity with BIER and RIFT protocols and procedures is assumed. Some terminologies are listed below for convenience.

[To be added.]

2. Introduction

BIER [RFC8279] ... (to be expanded)

RIFT [I-D.przygienda-rift] is a new protocol specifically designed for CLOS and fat-tree network topologies. As a hybrid between Link State Routing and Distance Vector Routing, it does LSR in northbound (towards the spine) and DVR in southbound (towards the leaves).

[I-D.ietf-bier-isis-extensions] and [I-D.ietf-bier-ospf-bier-extensions] specify ISIS/OSPF extensions to support BIER in an ISIS/OSPF domain. The same approach applies to RIFT in the northbound LSR.

[I-D.zwzw-bier-prefix-redistribute] specifies methods to advertise BIER information via default or summary/aggregate routes advertised

from one IGP area/domain to another. Similar approach applies to RIFT in the southbound DVR.

BIER encapsulation, whether it is based on MPLS or not, is covered in [RFC8296]. However, the OSPF/ISIS extensions for BIER only covers signaling needed for MPLS encapsulation. RIFT is targeted at DC deployments, where MPLS may not be used. This document covers signaling for both BIER MPLS and non-MPLS encapsulation with RIFT.

The details are provided in following sections.

3. Advertising BIER Information For non-MPLS Encapsulation

In the BIER architecture, a BIER sub-domain may have multiple BitStringLengths (BSLs) and multiple Encapsulations (Encaps). A single multicast packet coming from outside the BIER sub-domain may be sent as multiple BIER packets, one for each set that is identified by a SetID (SI). An incoming BIER packet is forwarded according to a BIFT for the <SD,Encap,BSL,SI> tuple. Each BIFT is identified by a 20-bit opaque number (BIFT-ID) in the packet.

With MPLS encapsulation, the BIFT-ID for an incoming BIER packet is simply an MPLS label allocated by the receiving BFR for the BIFT. For each <SD,BSL> tuple, OSPF/ISIS advertises a block of contiguous labels, one label for each SI needed for the tuple, in the MPLS Encapsulation sub-sub-TLV as part of the BIER sub-TLV, which is attached to the Extended Reachability TLV (ISIS case) or the Extended Prefix TLV (OSPF case) for the BFR's BIER Prefix.

With non-MPLS encapsulation, the BIFT-ID in the packet is at the same position as the label in MPLS encapsulation case. Its semantics is no different from the MPLS case in that as an 20-bit opaque value, it leads to the BIFT according to which the BIER packet is forwarded. Beyond the semantics, there are two differences from the MPLS case though:

- o MPLS infrastructure is not needed.
- o While each BFR could allocate local BIFT-IDs independently and advertise them just like in MPLS case, for the same <SD,Encap,BSL,SI> tuple all BFRs could optionally auto-derive or be provisioned with the same BIFT-ID and no signaling is needed in that case.

One may consider that if MPLS would allow to use consistently provisioned BIER labels on all BFRs, then the second difference listed above does not exist anymore.

In this specification, if locally significant BIFT-IDs are to be used with non-MPLS encapsulation, the BIFT-IDs are advertised the same way as in the MPLS case - by a BIFT-ID block, which is a block of contiguous labels in MPLS case or a block of contiguous opaque 20-bit values in non-MPLS case. The only difference is the type of encapsulation.

If consistently provisioned or auto-derived BIFT-IDs are used with non-MPLS encapsulation, then no BIFT-ID block is signaled. Just the encapsulation type is signaled.

4. Advertising BIER Information Northbound

Nothing special here compared to OSPF/ISIS. A node's local BIER information as described in the previous section is attached to a local BIER Prefix. Details to be added.

5. Advertising BIER Information Southbound

5.1. Local BIER Information

Similar to the northbound case, a node's local BIER information is attached to a local BIER prefix that is advertised southbound.

5.2. Proxied BFR-ID Ranges

On the southbound, a node advertises a default route, plus certain prefixes to prevent blackholing or suboptimal routing upon link failures. Those prefixes and default route are like the summary routes and default route in [I-D.zwzw-bier-prefix-redistribute], and similarly they carry BFR-IDs corresponding to the covered BIER Prefixes.

Consider a RIFT network with a BIER sub-domain of 200 BFIR/BFERS. Each non-leaf node is provisioned that BFR-ID 1-200 are used. Suppose a node X advertise southbound a default route RT1 and disaggregation routes RT2/RT3. RT2 and RT3 MUST advertise BFR-IDs covered by them (e.g. BFR-ID 100/102/150 covered by RT2 and BFR-ID 101/103 covered by RT3), while the default route RT1 can always advertise that all BFR-ID 1~200 are covered by it and does not need to exclude BFR-ID 100/102/150 and 101/103 that are covered by RT2/RT3. When a southern node receives RT1 and RT2/RT3, it installs BFR-ID 100/102/150 in its BIFT according to RT2, 101/103 in its BIFT according to RT3, and installs other BFR-IDs (or just a default route) in its BIFT according to RT1.

6. Information Elements Schema

This document introduces a bier.thrift schema with definitions to be used in RIFT encoding.thrift.

6.1. bier.thrift

```
typedef i8      SubdomainIdType
typedef i16     BfrIdType
typedef i8      BARType
typedef i8      IPAType
typedef i16     BSLType      /* Number of bits */
typedef i32     BiftIdType   /* Only the most significant 20 bits are used */
/

enum EncapsulationType {
    mpls      = 0;
    non-mpls   = 1;
}

/* Similar to the label range in OSPF/ISIS extensions for BIER */
struct BiftIdBlock {
    1: required BiftIdType      bift_id_base;
    2: required i8              bift_id_range;
}

/* Similar to the MPLS Encapsulation sub-sub-TLV in OSPF/ISIS */
struct EncapStruct {
    1: required EncapsulationType encap_type;
    2: required BSLType           bsl;
    3: optional BiftIdBlock       bift_id_block;
}

/*BIER node information. Similar to BIER sub-TLV in OSPF/ISIS. */
struct BierInfo {
    1: required SubdomainIdType  subdomain_id;
    2: required BfrIdType        bfr_id;
    3: required BARType          bar;
    4: required IPAType          ipa;
    5: required EncapStruct      encaps;      /* one or more */
}

struct ProxyBfrIdRange {
    1: required SubdomainIdType  subdomain_id;
    2: required BfrIdType        bfr_id_base;
    3: required BSLType          bfr_id_range;
}
```

6.2. Additions to encoding.thrift

The PrefixAttributes in encoding.rift now has two optional elements:

```
struct PrefixAttributes {  
    ...  
    2: optional BierInfo      bier_info;    /* BIER info for a  
                                           * local BIER Prefix */  
    3: optional ProxyBfrIdRange proxy_bfr_id; /* one or more proxy  
                                           * BFR-ID ranges covered  
                                           * by this prefix */  
}
```

7. IANA Considerations

8. Acknowledgements

9. References

9.1. Normative References

- [I-D.przygienda-rift]
Przygienda, T., Sharma, A., Atlas, A., and J. Drake,
"RIFT: Routing in Fat Trees", draft-przygienda-rift-05
(work in progress), March 2018.
- [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>>.
- [RFC8279] Wijnands, IJ., Ed., Rosen, E., Ed., Dolganow, A.,
Przygienda, T., and S. Aldrin, "Multicast Using Bit Index
Explicit Replication (BIER)", RFC 8279,
DOI 10.17487/RFC8279, November 2017,
<<https://www.rfc-editor.org/info/rfc8279>>.
- [RFC8296] Wijnands, IJ., Ed., Rosen, E., Ed., Dolganow, A.,
Tantsura, J., Aldrin, S., and I. Meilik, "Encapsulation
for Bit Index Explicit Replication (BIER) in MPLS and Non-
MPLS Networks", RFC 8296, DOI 10.17487/RFC8296, January
2018, <<https://www.rfc-editor.org/info/rfc8296>>.

9.2. Informative References

[I-D.ietf-bier-isis-extensions]

Ginsberg, L., Przygienda, T., Aldrin, S., and Z. Zhang,
"BIER support via ISIS", draft-ietf-bier-isis-
extensions-09 (work in progress), February 2018.

[I-D.ietf-bier-ospf-bier-extensions]

Psenak, P., Kumar, N., Wijnands, I., Dolganow, A.,
Przygienda, T., Zhang, Z., and S. Aldrin, "OSPF Extensions
for BIER", draft-ietf-bier-ospf-bier-extensions-15 (work
in progress), February 2018.

[I-D.zwzw-bier-prefix-redistribute]

Zhang, Z., Bo, W., Zhang, Z., and I. Wijnands, "BIER
Prefix Redistribute", draft-zwzw-bier-prefix-
redistribute-00 (work in progress), January 2018.

Authors' Addresses

Zhaohui Zhang
Juniper Networks

EMail: z Zhang@juniper.net

Shaowen Ma
Juniper Networks

EMail: mashao@juniper.net

Zheng Zhang
ZTE Corporation

EMail: zhang.zheng@zte.com.cn