LSR Working Group
Internet Draft
Intended status: Standards Track
Expires: April 2019

Dave Allan
Ericsson

October 2018

A Distributed Algorithm for Constrained Flooding of IGP Advertisements
draft-allan-lsr-flooding-algorithm-00

Abstract

This document describes a distributed algorithm that can be applied to the problem of constraining IGP flooding in dense mesh topologies. The flooding topology utilizes two node-diverse spanning trees in order to provide complete coverage in the presence of any single failure while constraining the number of LSAs received by any IGP speaker connected to the flooding topology.

Table of Contents

1. Introduction

This memo describes an algorithm suitable for reducing the quantity of IGP flooding in dense mesh networks. The only property that the algorithm is dependent upon is that there are at least two equal and diverse shortest paths between any pair of IGP speakers in order to meet the requirements elucidated in [Li]. The algorithm uses a re-purposing of the tie breaking algorithm used in 802.1aq Shortest Path Bridging as an element of construction of the flooding topology. It is not the intention of this memo to specify a complete solution, but to offer a foundation of an eventual solution.

1.1. Authors

David Allan

1.2. 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 [RFC2119].

2. Conventions used in this document

2.1. Terminology

Member Adjacency - An adjacency that has been determined to part of the flooding topology.

Member Node - A Participant node that is connected to the flooding topology.

Participant Adjacency - An adjacency between two participating nodes. It may be a member adjacency or a non-member adjacency

Non-Participant Adjacency - An adjacency where at least one of the two nodes is a not a Participating Node

Participating Node - An IGP speaker that has advertised the capability, and hence the intention, to participate in a flooding topology

3. Solution Overview

3.1. The Flooding Topology

A flooding topology is composed of a contiguously connected set of participating nodes.

The flooding topology constructed from two diversely rooted spanning trees. A participating node that is connected to the physical topology with a degree of two or greater and has at least two participating adjacencies will be bi-connected to the flooding topology.

The resulting flooding topology diameter will typically be two times the depth of the tree hierarchy. The compromise in this approach is that a subset of nodes in the network will not see a reduction of the replication burden from current practice when flooding LSAs as the degree of a subset of nodes in the flooding topology will correspond to the degree of the physical topology.

The protocol structure of flooded information is unmodified. A participant node may relay a received LSA onto member links of both spanning trees. Specific forwarding rules prevent undue flooding, the result being that every participant node that is bi-connected to the flooding topology will receive two copies of any flooded LSA in a fault free network. Participating nodes that due to network degradation are only singly connected will receive one copy. The forwarding rules are described in section 4.4.

3.2. Solution Applicability

This algorithm has been considered in the context of pure bipartite graphs, bipartite graphs modified with the addition of intra-tier adjacencies, and hierarchical variations of the above. Applicability to other network designs is for further study.

For all graphs the link costs are assumed to be common for all inter-tier links and common for any intra-tier links. Inter-tier and intra-tier links do not have to have the same cost.

3.3. Algorithm

The algorithm borrows from 802.1aq for the construction of the spanning trees used in this application. This is described in clause 28.5 of [802.1Q].

3.3.1. Algorithm Basics

The key component of the 802.1aq employed is the tie breaking algorithm. The original application of the algorithm was to produce a symmetrically congruent mesh of multicast trees and unicast forwarding whereby the path between any two nodes in the network was symmetric in both directions and congruent for both unicast and multicast traffic.

For this application the algorithm is used in the generation of two diversely rooted spanning trees that define the flooding topology.

As part of tree construction, the algorithm tie breaks between equal cost paths. When a tie is identified as part of a Dijkstra computation, a path-id is constructed for each equal cost path. A path-id is expressed as a lexicographically sorted list of the node-ids in the path. The set of equal cost paths is ranked, and the lowest selected. As an example:

Path-id 23-39-44-68-85 is ranked lower than

Path-id 23-44-59-63-90

When the path-ids are of unequal length, the path-ids with the fewest hops are ranked superior to the longer paths, and tie breaking is applied to select between the shorter path-ids. This is not expected to apply in the general case of the dense graphs this application is targeted at.

The node-ids used would be the loopback address of each node, therefore each path-id will be unique.

3.3.2. Generating Diverse Trees

The algorithm includes the concept of an "algorithm-mask", which is a value XOR'd with the node-ids prior to sorting into path IDs and ranking the paths. This permits the construction of diverse trees in a dense topology.

Two algorithm masks are used (zero and -1). When computing two trees from the same root, when there are at least two nodes to choose from at each distance from the root, fully diverse trees will be generated. When computing two trees from diverse roots in a tree architecture, diverse nodes will be selected in each tier in the hierarchy as the relay nodes to the next tier.

3.3.3. Desirable Properties Computation Wise

The algorithm has the property of permitting the pruning of intermediate state as a Dijkstra progresses as ties can be immediately evaluated, and the all but the selected path removed from further consideration. This is desirable when computing a Dijkstra in a dense graph as all path permutations do not need to be carried forward during computation. This permits the computation to be quite fast.

The resulting computational complexity would still be expressed as 2N(lnN).

4. Applying the Algorithm

4.1. Tree Generation

Each IGP speaker in the network has knowledge of each of the two spanning tree roots and the algorithm mask associated with each. This memo does not specify how root selection is performed and disseminated through the network, but does discuss selection requirements in section 4.5.

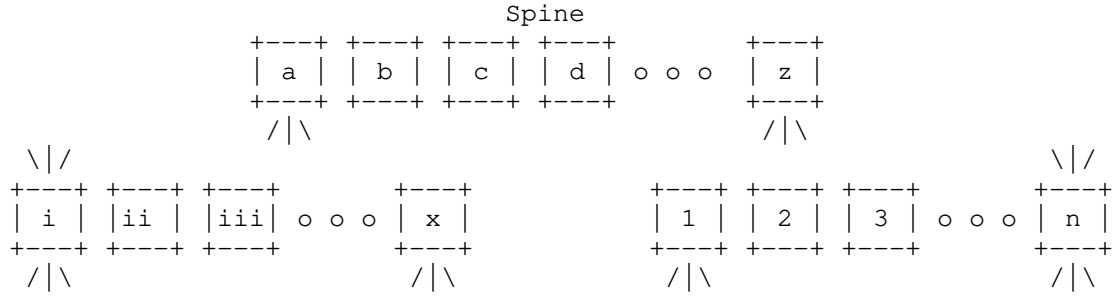
Each root has one of the two algorithm masks associated with it.

Each participating IGP speaker in the network computes a spanning tree from each the two roots (using the algorithm mask associated with each root) and from that can determine its own role in the flooding topology. The two spanning trees are designated the "low spanning tree" and the "high spanning tree".

The spanning trees are a starting point for a redundant topology. Unlike the commonly accepted operation of a spanning tree, in this application the distinction between upstream and downstream adjacencies is important and is an input to how a member node further relays any LSAs received. Upstream member adjacencies are in the direction of a root, and downstream member adjacencies are in the direction away from the root.

4.2. Illustrating the result

The following diagram illustrates the general layout of the flooding graph constructed using the algorithm as applied to a bi-partite style of tree (no intra tier links):



In the example, there are two tiers of switches. The spine (nodes a..z), and the next tier with two groups of nodes (i..x) and (1..n). The algorithm will select the node with the lowest node ID in each tier as the replicating node for the low spanning tree; 'a' and 'i' for the set of nodes connecting the spine and the next tier. The algorithm will select the nodes with highest node ID in the same set of nodes for the high spanning tree; 'z' and 'n' for the same set of nodes.

In the flooding topology:

- Node 'a' is connected to nodes i..x and 1..n for the low spanning tree.

- Node 'z' is connected to the same set of nodes for the high spanning tree.

- Node 'i' is connected to nodes 'a'..'z' for the low spanning tree, and

- Node 'n' is connected to the same nodes for the high spanning tree.

- All other nodes are bi-connected to the flooding topology

If there was a further tier added below nodes i..x, then 'i' and 'x' would be selected as the replicating nodes for the low and high spanning tree respectively. This is similarly true for nodes 1..n.

4.3. Interactions between Participating and Non-Participating Nodes

This solution proposes primarily only nodal behaviors with respect to constraining flooding to member adjacencies. To address the scenario

where the participating nodes were a subset of a larger network, it would be necessary to advertise the capability to participate in flood reduction.

This would then require that each participating node use this information to be able to identify the set of participating adjacencies and confine the spanning tree computation to the set of participating adjacencies in order to identify local set of member adjacencies. Interactions with non-participant adjacencies would conform to current practice.

4.4. Flooding of LSAs

The design of the protocol elements that are flooded is unmodified by this solution. Therefore, there is no additional information available to associate a received LSA with a given tree, nor is such information needed; the two spanning trees are not treated as unique entities in the flooding topology.

As per current practice, a node does not relay LSAs that it has already seen.

A new LSA received from an upstream member adjacency is flooded on:

- All downstream member adjacencies exclusive of the adjacency of arrival, irrespective of which tree the adjacencies are part of.

- All non-participant adjacencies

A new LSA received from a downstream member adjacency is flooded on:

- All other member adjacencies exclusive of the adjacency of arrival irrespective of which tree the adjacencies are part of.

- All non-participant adjacencies

A new LSA received from a member adjacency where upstream and downstream is ambiguous (it is an upstream member on one of the spanning trees and a downstream member on the other), is flooded on:

- All other member adjacencies exclusive of the adjacency of arrival irrespective of which adjacency the links are part of.

- All Non-Participant adjacencies

A new LSA received from a non-member adjacency is flooded on all member adjacency irrespective of which tree the adjacencies are part of (see sections 5.1 and 5.5).

4.5. Root Selection

The algorithm depends on tie breaking between sets of node IDs to produce diverse paths, therefore it does place some restrictions on root selection.

A root SHOULD be selected so that the root"s node-id when XORd with the associated algorithm mask is the lowest ranked node in the local tier in the tree hierarchy. This would be analogous to path-id ranking where the paths were all of length 1.

The root MUST NOT be selected such that the node-ID when XORd with the other root"s algorithm mask is the lowest ranked node. This would result in the root also being a transit node for the other spanning tree and produce a scenario whereby a single failure could render both spanning trees incomplete.

Roots MUST NOT be directly connected for either of the low or high spanning trees. If the topology does not permit this to be satisfied purely by root selection, then the inter-root adjacency must be pruned from the graph prior to spanning tree computation to ensure that diverse paths between the roots are used.

For a true bipartite graph, there are no other restrictions on node selection.

For a bipartite graph modified with inter-tier links, the roots MUST be placed in different tiers to ensure a pathological combination of link weights and node-ids does not result in a scenario where a single failure would render the flooding topology incomplete.

Other sources of failure may exist that may require an administrative component to root selection. This, for example, would ensure that both roots were not selected from a common shared risk group.

See also section 5.5.

4.6. Node Additions

A participating node that is added to the topology will initially not be served by the flooding topology. A participating node adjacent to that node is required to treat it as a non-participating node until such time as tree re-optimization has completed. At the end of tree

optimization, typically two adjacent participating nodes will have member adjacencies with the new node, so the ability to flood LSAs between the new node and the flooding topology will have been uninterrupted during the process.

5. Further work

5.1. Thoughts on Coexistence in the Context of a Larger Network

A node that had a combination of participating and non-participating adjacencies would be required to do the following:

- For any new LSA received on a participating adjacency, in addition to the rules for member adjacencies, it would also flood the LSA on all non-participating adjacencies.

- For any new LSA received on a non-participating adjacency, it would flood the LSA on all member adjacencies.

This is reflected in the forwarding rules described in section 4.4.

5.1.1. Multiple flooding Domains and the Severing of Flooding Domains

It is possible to envision several scenarios whereby there are sets of participating nodes that are not contiguously connected via participating adjacencies in a given IGP domain.

1. A node has been incorrectly configured as a participating node but has no participating adjacencies.

2. A participating node or set of nodes has become severed from the flooding topology but is still connected to other nodes in the network. Nodes in this set would still be able to compute a local extension of the flooding topology, but it would only be useful if the set was sufficiently large that a majority of the nodes were not connected to non-participants.

3. Procedures are designed to permit more than one flooding topology in an IGP domain. In which case participating nodes would have to be administratively configured to associate with a flooding topology instance.

5.2. Thoughts on Flooding Topology Re-Optimization

After a topology change, it is desirable that the flooding topology remain stable until the network has stabilized. However a single failure may render one of the spanning trees incomplete, such that a

further single failure could make the flooding topology incomplete. Therefore procedures should include re-optimization of the flooding topology after a topology change. In order to maintain complete coverage it would make sense not to recompute the spanning trees simultaneously.

One approach that would appear to make sense to separate in time network convergence, re-optimization of the low spanning tree and re-optimization of the high spanning tree.

The ideal would be to reoptimize an incomplete tree first, however this would require the participating nodes to maintain a complete map of all member adjacencies so that a common determination of the most degraded spanning tree and hence the order of re-optimization could be made.

5.3. Thoughts on Node and Network Initialization

A participating node at power up will be not be able to establish member links until it has synchronized with the network and the network is stable in the new topology. This suggests it simply treats power up similarly to how a topology change and network re-optimization is treated. The only difference being that it will flood all LSAs received or originated as per current practice until both spanning trees have stabilized.

5.4. Thoughts on Loop Prevention

802.1aq included additional mechanisms to prevent looping, a reverse path forwarding check, and digest exchange across adjacencies to ensure IGP synchronization.

Routing LSAs are not relayed if they are a duplicate, therefore destructive looping cannot occur and additional mitigation mechanisms are not required.

5.5. Thoughts on Pathological Failure Scenarios

While in a stable fault free network with sufficient mesh density of the types considered, the flooding topology used by this solution would ensure that no single failure rendered both spanning trees incomplete, it is also useful to consider multiple failure scenarios and if they can be mitigated.

Preliminary analysis suggests that in a tree network of sufficient mesh density, the only dual link failure that can render the flooding topology incomplete is if a participant node has failures in both

upstream member adjacencies. This can be partially mitigated if the node recognizes this scenario and reverts to flooding on all adjacencies. If the suggested procedures of 5.1.1 above are adopted, surrounding participating nodes that receive the LSA on a non-member adjacency will introduce the LSA into the flooding topology.

The pathological scenario is the simultaneous failure of both roots. This does suggest that root selection should place the roots two hops apart so there will be a constituency of participants that would observe a simultaneous failure of both upstream member adjacencies and revert to normal flooding.

6. Acknowledgements

The author would like to acknowledge Jerome Chiabaut for his original algorithm work that underpins this memo.

7. Security Considerations

For a future version of this document.

8. IANA Considerations

This memo requires no IANA allocations

9. References

9.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

9.2. Informative References

[802.1Q] 802.1Q (2014) IEEE Standard for Local and Metropolitan Area Networks--Media Access Control (MAC) Bridges and Virtual Bridged Local Area Networks

[Li] Li, T., Psenak, P., "Dynamic Flooding on Dense Graphs", IETF work in progress, draft-li-dynamic-flooding-05, June 2018

10. Author's Address

Dave Allan
Ericsson
2455 Augustine Drive
Santa Clara, CA 95054
USA
Email: david.i.allan@ericsson.com

RIFT Working Group
Internet-Draft
Intended status: Standards Track
Expires: 3 October 2024

A. Przygienda, Ed.
J. Head, Ed.
Juniper Networks
A. Sharma
Hudson River Trading
P. Thubert
Bruno. Rijsman
Individual
Dmitry. Afanasiev
Yandex
1 April 2024

RIFT: Routing in Fat Trees
draft-ietf-rift-rift-21

Abstract

This document defines a specialized, dynamic routing protocol for Clos, fat tree, and variants thereof. These topologies were initially used within crossbar interconnects, and consequently router and switch backplanes, but their characteristics make them ideal for constructing IP fabrics as well. The protocol specified by this document is optimized toward the minimization of control plane state to support very large substrates as well as the minimization of configuration and operational complexity to allow for simplified deployment of said topologies.

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 3 October 2024.

Table of Contents

1. Introduction

Clos [CLOS] topologies (called commonly a fat tree/network in modern IP fabric considerations [VAHDAT08] as homonym to the original definition of the term [FATTREE]) have gained prominence in today's networking, primarily as a result of the paradigm shift towards a centralized data-center architecture that is poised to deliver a majority of computation and storage services in the future. Many builders of such IP fabrics desire a protocol that auto-configures itself and deals with failures and mis-configurations with a minimum of human intervention. Such a solution would allow local IP fabric bandwidth to be consumed in a 'standard component' fashion, i.e. provision it much faster and operate it at much lower costs than today, much like compute or storage is consumed already.

In looking at the problem through the lens of such IP fabric requirements, RIFT (Routing in Fat Trees) addresses those challenges not through an incremental modification of either a link-state (distributed computation) or distance-vector (diffused computation) techniques but rather a mixture of both, briefly described as "link-state towards the spines" and "distance vector towards the leaves". In other words, "bottom" levels are flooding their link-state information in the "northern" direction while each node generates under normal conditions a "default route" and floods it in the "southern" direction. This type of protocol allows naturally for highly desirable address aggregation. Alas, such aggregation could drop traffic in cases of misconfiguration or while failures are being resolved or even cause persistent network partitioning and this has to be addressed by some adequate mechanism. The approach RIFT takes is described in Section 6.5 and is based on automatic, sufficient disaggregation of prefixes in case of link and node failures.

The protocol does further provide:

* optional fully automated construction of fat tree topologies based on detection of links without any configuration (Section 6.7), while allowing for conventional configuration methods or an arbitrary mix of both,

* minimum amount of routing state held by nodes,

* automatic pruning and load balancing of topology flooding exchanges over a sufficient subset of links (Section 6.3.9),

* automatic address aggregation (Section 6.3.8) and consequently automatic disaggregation (Section 6.5) of prefixes on link and node failures to prevent traffic loss and suboptimal routing,

* loop-free non-ECMP forwarding due to its inherent valley-free nature,

* fast mobility (Section 6.8.4),

* re-balancing of traffic towards the spines based on bandwidth available (Section 6.8.7.1), and finally

* mechanisms to synchronize a limited key-value data-store (Section 6.8.5.1) that can be used after protocol convergence to e.g. bootstrap higher levels of functionality on nodes.

Figure 1 illustrates a simplified, conceptual view of a RIFT fabric with its routing tables and topology databases. The top of the fabric's link-state database holds information about the nodes below it and the routes to them. When referring to Figure 1, the /32 notation corresponds to each node's loopback address (e.g. A/32 is node A's loopback, etc.) and 0/0 indicates a default route. The first row of database information represents the nodes for which full topology information is available. The second row of database information indicates that partial information of other nodes in the same level is also available. Such information will be necessary to perform certain algorithms necessary for correct protocol operation. When the "bottom" of the fabric is considered, or in other words the leaves, the topology is basically empty and, under normal conditions, the leaves hold a load balanced default route to the next level.

The remainder of this document fills in the protocol specification details.

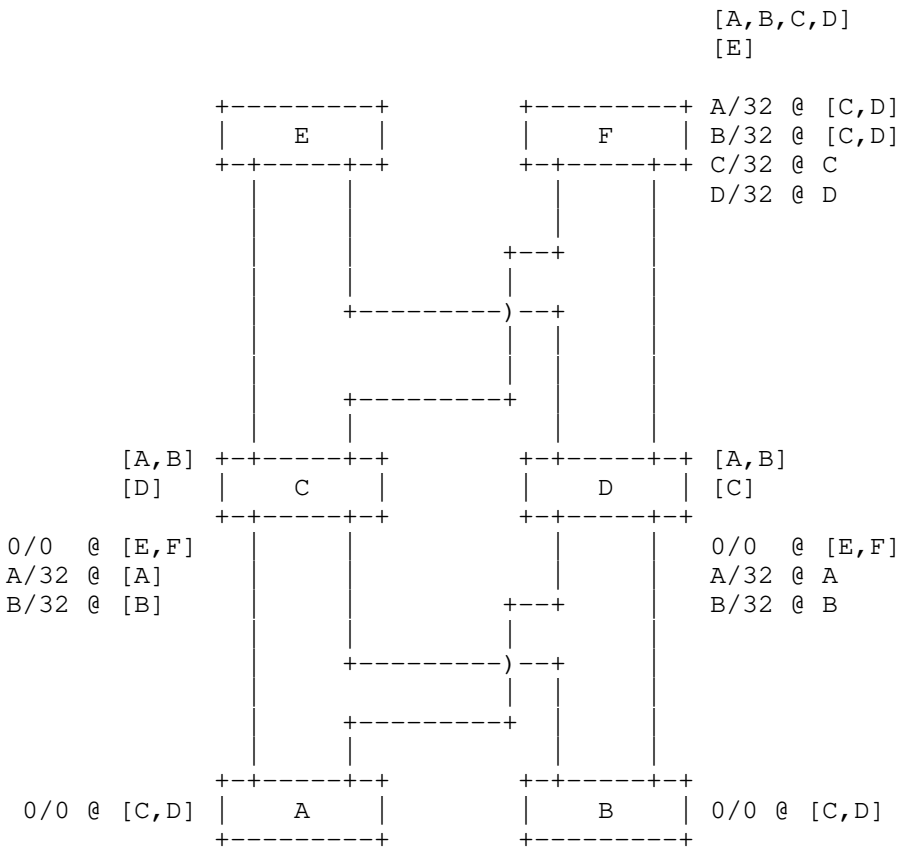


Figure 1: RIFT Information Distribution

1.1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

2. A Reader's Digest

This section is an initial guided tour through the document in order to convey the necessary information for different readers, depending on their level of interest. The authors recommend reading the HTML or PDF versions of this document due to the inherent limitation of text version to represent complex figures.

The Terminology (Section 3.1) section should be used as a supporting reference as the document is read.

The indications of direction (i.e. "top", "bottom", etc.) referenced in Section 1 are of paramount importance. RIFT requires a topology with a sense of top and bottom in order to properly achieve a sorted topology. Clos, Fat Tree, and other similarly structured networks are conducive to such requirements. Where RIFT does allow for further relaxation of these constraints, this will be mentioned later in this section.

Several of the images in this document are annotated with "northern view" or "southern view" to indicate perspective to the reader. A "northern view" should be interpreted as "from the top of the fabric looking down", whereas "southern view" should be interpreted as "from the bottom looking up".

Operators and implementors alike must decide whether multi-plane IP fabrics are of interest for them. Section 3.2 illustrates an example of both single-plane in Figure 2 and multi-plane fabric in Figure 3. Multi-plane fabrics require understanding of additional RIFT concepts (e.g. negative disaggregation in Section 6.5.2) that are unnecessary in the context of fabrics consisting of a single-plane only. The Overview (Section 5) and Section 5.2 aim to provide enough context to determine if multi-plane fabrics are of interest to the reader. The Fallen Leaf part (Section 5.3), and additionally Section 5.4 and Section 5.5 describe further considerations that are specific to multi-plane fabrics.

The fundamental protocol concepts are described starting in the specification part (Section 6), but some sub-sections are less relevant unless the protocol is being implemented. The protocol transport (Section 6.1) is of particular importance for two reasons. First, it introduces RIFT's packet format content in the form of a normative Thrift model given in Appendix B.3 carried in according security envelope as described in Section 6.9.3. Second, the Thrift model component is a prelude to understanding the RIFT's inherent security features as defined in both security models part (Section 6.9) and the security segment (Section 9). The normative schema defining the Thrift model can be found in Appendix B.2 and Appendix B.3. Furthermore, while a detailed understanding of Thrift [thrift] and the models is not required unless implementing RIFT, they may provide additional useful information for other readers.

If implementing RIFT to support multi-plane topologies Section 6 should be reviewed in its entirety in conjunction with the previously mentioned Thrift schemas. Sections not relevant to single-plane implementations will be noted later in this section.

All readers dealing with implementation of the protocol should pay special attention to the Link Information Element (LIE) definitions part (Section 6.2) as it not only outlines basic neighbor discovery and adjacency formation, but also provides necessary context for RIFT's Zero Touch Provisioning (ZTP) (Section 6.7) and mis-cabling detection capabilities that allow it to automatically detect and build the underlay topology with basically no configuration. These specific capabilities are detailed in Section 6.7.

For other readers, the following sections provide a more detailed understanding of the fundamental properties and highlight some additional benefits of RIFT such as link state packet formats, efficient flooding, synchronization, loop-free path computation and link-state database maintenance - Section 6.3, Section 6.3.2, Section 6.3.3, Section 6.3.4, Section 6.3.6, Section 6.3.7, Section 6.3.8, Section 6.4, Section 6.4.1, Section 6.4.2, Section 6.4.3, Section 6.4.4. RIFT's ability to perform weighted unequal-cost load balancing of traffic across all available links is outlined in Section 6.8.7 with an accompanying example.

Section 6.5 is the place where the single-plane vs. multi-plane requirement is explained in more detail. For those interested in single-plane fabrics, only Section 6.5.1 is required. For the multi-plane interested reader Section 6.5.2, Section 6.5.2.1, Section 6.5.2.2, and Section 6.5.2.3 are also mandatory. Section 6.6 is especially important for any multi-plane interested reader as it outlines how the RIB (Routing Information Base) and FIB (Forwarding Information Base) are built via the disaggregation mechanisms, but also illustrates how they prevent defective routing decisions that cause traffic loss in both single or multi-plane topologies.

Section 7 contains a set of comprehensive examples that show how RIFT contains the impact of failures to only the required set of nodes. It should also help cement some of RIFT's core concepts in the reader's mind.

Last, but not least, RIFT has other optional capabilities. One example is the key-value data-store, which enables RIFT to advertise data post-convergence in order to bootstrap higher levels of functionality (e.g. operational telemetry). Those are covered in Section 6.8.

More information related to RIFT can be found in the "RIFT Applicability" [APPLICABILITY] document, which discusses alternate topologies upon which RIFT may be deployed, use cases where it is applicable, and presents operational considerations that complement this document. The RIFT DayOne [DayOne] book covers some practical details of existing RIFT implementations.

3. Reference Frame

3.1. Terminology

This section presents the terminology used in this document.

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

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

Bow-tying:
: Traffic patterns in fully converged IP fabrics traverse normally the shortest route based on hop count toward their destination (e.g., leaf, spine, leaf). Some failure scenarios with partial routing information cause nodes to lose the required downstream reachability to a destination and forcing traffic to utilize routes that traverse higher levels in the fabric in order to turn south again using a different to resolve reachability (e.g., leaf, spine-1, super-spine, spine-2, leaf).

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

Cost:
: The sum of metrics between two nodes.

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

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

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

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

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

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

Folded Spine-and-Leaf:
: In case the Clos fabric input and output stages are analogous, the fabric can be "folded" to build a "superspine" or top which is called the ToF in this document.

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

Key Value (KV) TIE:
: A TIE that is carrying a set of key value pairs [DYNAMO]. It can be used to distribute non topology related information within the protocol.

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

Leaf:
: A node without southbound adjacencies. Level 0 implies a leaf in RIFT but a leaf does not have to be level 0.

Level:
: Clos and Fat Tree networks are topologically partially ordered graphs and 'level' denotes the set of nodes at the same height in such a network. Nodes at the top level (i.e., ToF) are at the level with the highest value and count down to the nodes at the bottom level (i.e., leaf) with the lowest value. A node will have links to nodes one level down and/or one level up. In some

circumstances, a node may have links to other nodes at the same level. A leaf node may also have links to nodes multiple levels higher. In RIFT, Level 0 always indicates that a node is a leaf, but does not have to be level 0. Level values can be configured manually or automatically derived via Section 6.7. As a final footnote: Clos terminology often uses the concept of "stage", but due to the folded nature of the Fat Tree it is not used from this point on to prevent misunderstandings.

LIE:
: This is an acronym for a "Link Information Element" exchanged on all the system's links running RIFT to form _ThreeWay_ adjacencies and carry information used to perform Zero Touch Provisioning (ZTP) of levels.

Metric:
: The cost between two neighbors exactly one layer 3 hop away from each other.

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

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

North Radix:
: The number of ports cabled northbound to higher level nodes.

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

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

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

Overloaded:
: Applies to a node advertising the _overload_ attribute as set. Overload attribute is carried in the _NodeFlags_ object of the encoding schema.

Point of Delivery (PoD):
: A self-contained vertical slice or subset of a Clos or Fat Tree network containing normally only level 0 and level 1 nodes. A node in a PoD communicates with nodes in other PoDs via the ToF nodes. PoDs are numbered to distinguish them and PoD value 0 (defined later in the encoding schema as _common.default_pod_) is used to denote "undefined" or "any" PoD.

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

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

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

Security Envelope:
: RIFT packets are flooded within an authenticated security envelope that allows to protect the integrity of information a node accepts. This is described in Section 6.9.3.

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

South Radix:
: The number of ports cabled southbound to lower-level nodes.

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

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

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

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

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

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

Superspine, Aggregation/Spine and Edge/Leaf Switches:"
: Traditional level names in 5-stages folded Clos for Level 2, 1 and 0 respectively (counting up from the bottom). We normalize this language to talk about ToF, Top-of-Pod (ToP) and leaves.

System ID:
: RIFT nodes identify themselves with a unique network-wide number when trying to build adjacencies or describe their topology. RIFT System IDs can be auto-derived or configured.

ThreeWay Adjacency:
: RIFT tries to form a unique adjacency between two nodes over a point-to-point interface and exchange local configuration and necessary ZTP information. An adjacency is only advertised in Node TIEs and used for computations after it achieved _ThreeWay_ state, i.e. both routers reflected each other in LIEs including relevant security information. Nevertheless, LIEs before _ThreeWay_ state is reached may carry ZTP related information already.

TIDE:
: Topology Information Description Element carrying descriptors of the TIEs stored in the node.

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. A TIE has always a direction and a type. North TIEs (sometimes abbreviated as N-TIEs) are used when dealing with TIEs in the northbound representation and South-TIEs (sometimes abbreviated as S-TIEs) for the southbound equivalent. TIEs have different types such as node and prefix TIEs.

TIEDB:
: The database holding the newest versions of all TIE headers (and the corresponding TIE content if it is available).

TIRE:
: Topology Information Request Element carrying set of TIDE descriptors. It can both confirm received and request missing TIEs.

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

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

ToF Plane or Partition:
: In large fabrics ToF switches may not have enough ports to aggregate all switches south of them and with that, the ToF is 'split' into multiple independent planes. Section 5.2 explains the concept in more detail. A plane is a subset of ToF nodes that are aware of each other through south reflection or E-W links.

Valid LIE:
: LIEs undergo different checks to determine their validity. The term "valid LIE" is used to describe a LIE that can be used to form or maintain an adjacency. The amount of checking itself depends on the FSM (Finite State Machine) involved and its state. A "minimally valid LIE" is a LIE that passes checks necessary on any FSM in any state. A "ThreeWay valid LIE" is a LIE that

successfully underwent further checks with a LIE FSM in _ThreeWay_ state. Minimally valid LIE is a subcategory of _ThreeWay_ valid LIE.

Zero Touch Provisioning (ZTP):
Optional RIFT mechanism which allows the automatic derivation of node levels based on minimum configuration. Such a mininum configuration consists solely of ToFs being configured as such.

Additionally, when the specification refers to elements of packet encoding or constants provided in the Appendix B a special emphasis is used, e.g. _invalid_distance_. The same convention is used when referring to finite state machine states or events outside the context of the machine itself, e.g., _OneWay_.

3.2. Topology

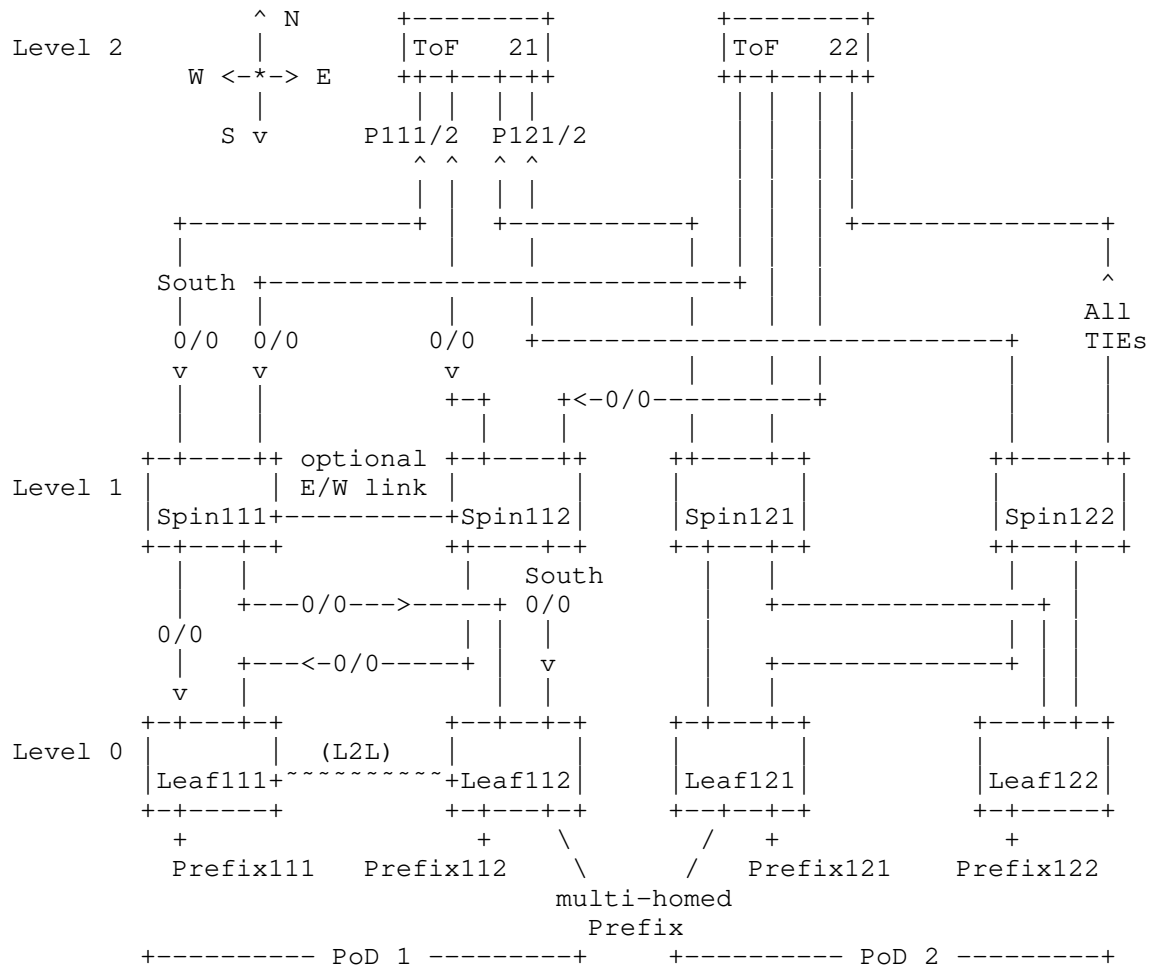
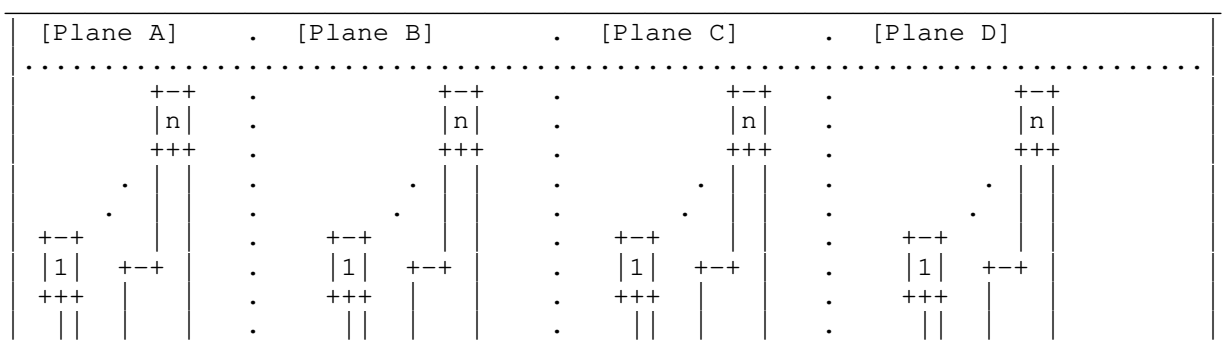


Figure 2: A Three Level Spine-and-Leaf Topology



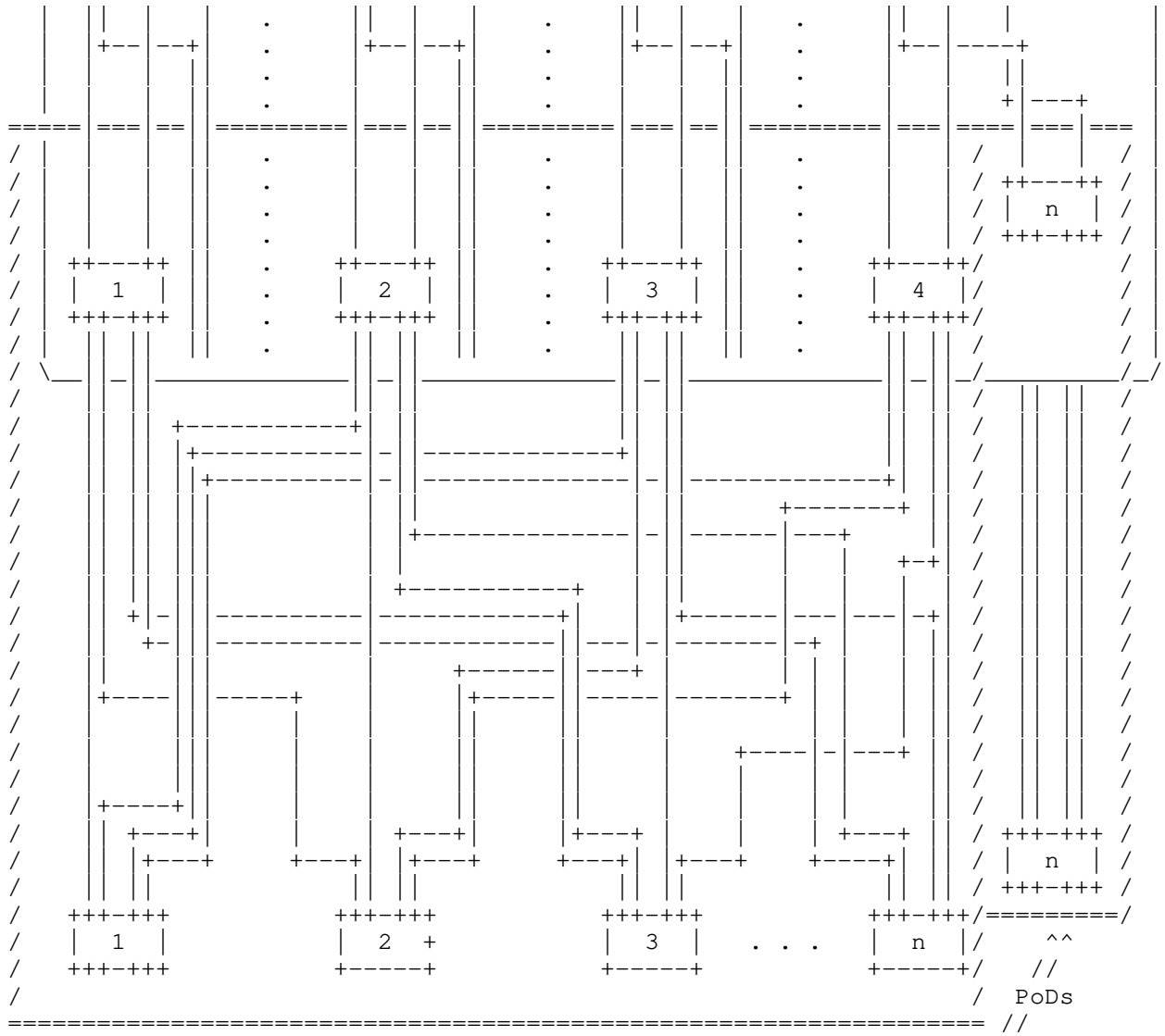


Figure 3: Topology with Multiple Planes

The topology in Figure 2 is referred to in all further considerations. This figure depicts a generic "single plane fat tree" and the concepts explained using three levels apply by induction to further levels and higher degrees of connectivity. Further, this document will deal also with designs that provide only sparser connectivity and "partitioned spines" as shown in Figure 3 and explained further in Section 5.2.

4. RIFT: Routing in Fat Trees

The remainder of this document presents the detailed specification of the RIFT protocol, which in the most abstract terms has many properties of a modified link-state protocol when distributing information northbound and a distance vector protocol when distributing information southbound. While this is an unusual combination, it does quite naturally exhibit desired properties.

5. Overview

5.1. Properties

The most singular property of RIFT is that it floods link-state information northbound only so that each level obtains the full topology of levels south of it. Link-State information is, with some exceptions, not flooded East-West or back South again. Exceptions like south reflection is explained in detail in Section 6.5.1 and east-west flooding at ToF level in multi-plane fabrics is outlined in Section 5.2. In the southbound direction, the necessary routing information required (normally just a default route as per Section 6.3.8) only propagates one hop south. Those nodes then generate their own routing information and flood it south to avoid the overhead of building an update per adjacency. For the moment describing the East-West direction is left out.

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

To account for the "northern" and the "southern" information split the link state database is partitioned accordingly into "north representation" and "south representation" Topology Information Elements (TIEs). In simplest terms the North TIEs contain a link state topology description of lower levels and South TIEs carry

simply node description of the level above and default routes pointing north. This oversimplified view will be refined gradually in the following sections while introducing protocol procedures and state machines at the same time.

5.2. Generalized Topology View

This section and resulting Section 6.5.2 are dedicated to multi-plane fabrics, in contrast with the single plane designs where all ToF nodes are topologically equal and initially connected to all the switches at the level below them.

Multi-plane design is effectively a multi-dimensional switching matrix. To make that easier to visualize, this document introduces a methodology depicting the connectivity in two-dimensional pictures. Further, it can be leveraged that what is under consideration here are basically stacked crossbar fabrics where ports align "on top of each other" in a regular fashion.

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

5.2.1. Terminology and Glossary

This section describes the terminology and abbreviations used in the rest of the text. Though the glossary may not be clear on a first read, the following sections will introduce the terms in their proper context.

P:
: Denotes the number of PoDs in a topology.

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

K:
To simplify the visual aids, notations and further considerations, the assumption is made that the switches are symmetrical, i.e., they have an equal number of ports pointing northbound and southbound. With that simplification, K denotes half of the radix of a symmetrical switch, meaning that the switch has K ports pointing north and K ports pointing south. K_LEAF (K of a leaf) thus represents both the number of access ports in a leaf Node and the maximum number of planes in the fabric, whereas K_TOP (K of a ToP) represents the number of leaves in the PoD and the number of ports pointing north in a ToP Node towards a higher spine level and thus the number of ToF nodes in a plane.

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

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

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

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

5.2.2. Clos as Crossed, Stacked Crossbars

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

Figure 4 provides a view of a leaf node as seen from the north, i.e. showing ports that connect northbound. For lack of a better symbol, the document chooses to use the "o" as ASCII visualisation of a

single port. In this example, K_LEAF has 6 ports. Observe that the number of PoDs is not related to Radix unless the ToF Nodes are constrained to be the same as the PoD nodes in a particular deployment.

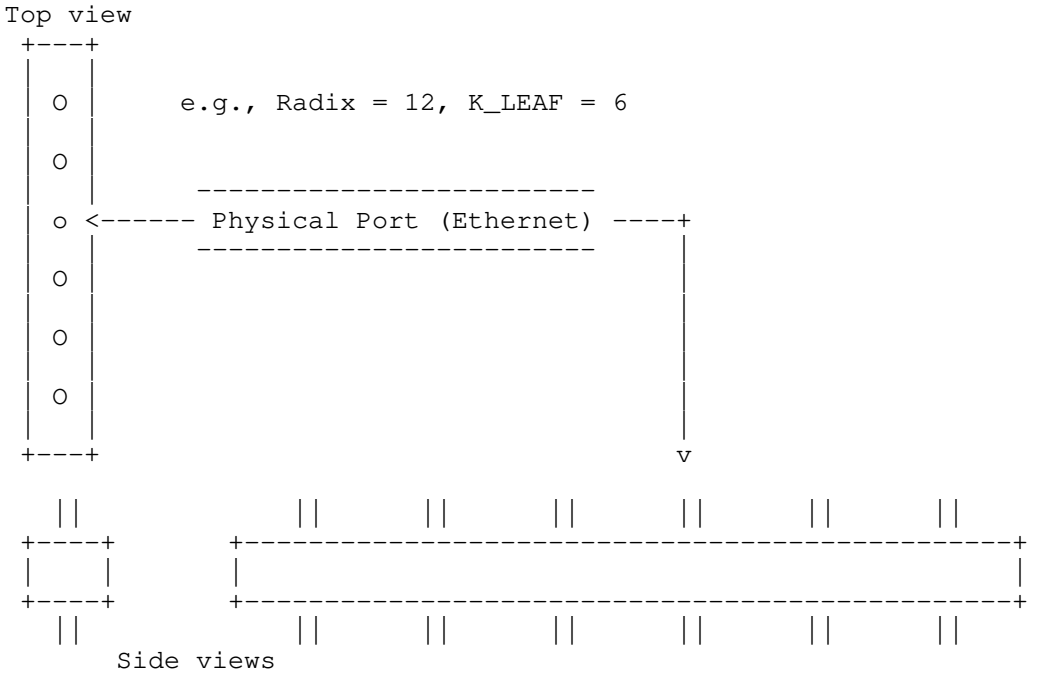


Figure 4: A Leaf Node, K_LEAF=6

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

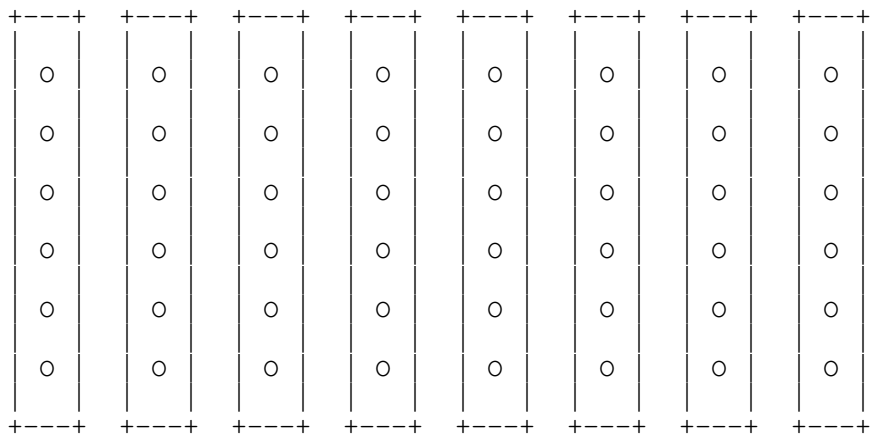


Figure 5: Southern View of Leaf Nodes of a PoD, K_TOP=8

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

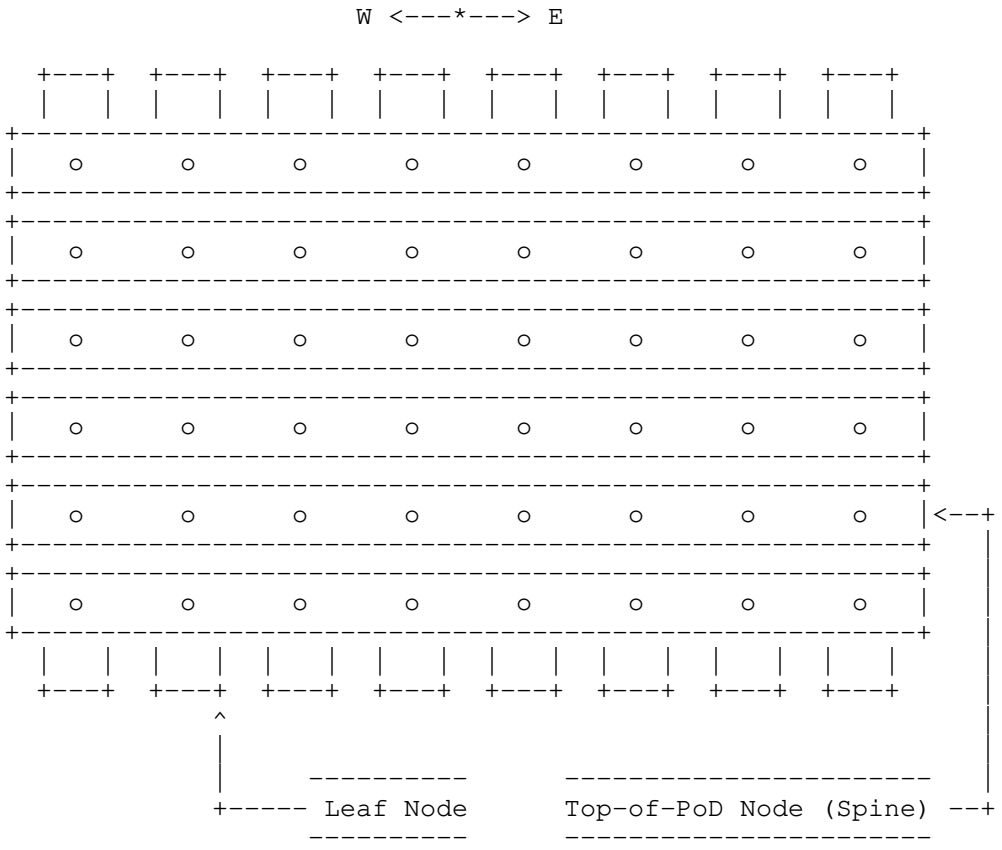


Figure 6: Northern View of a PoD's Spines, K_TOP=8

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

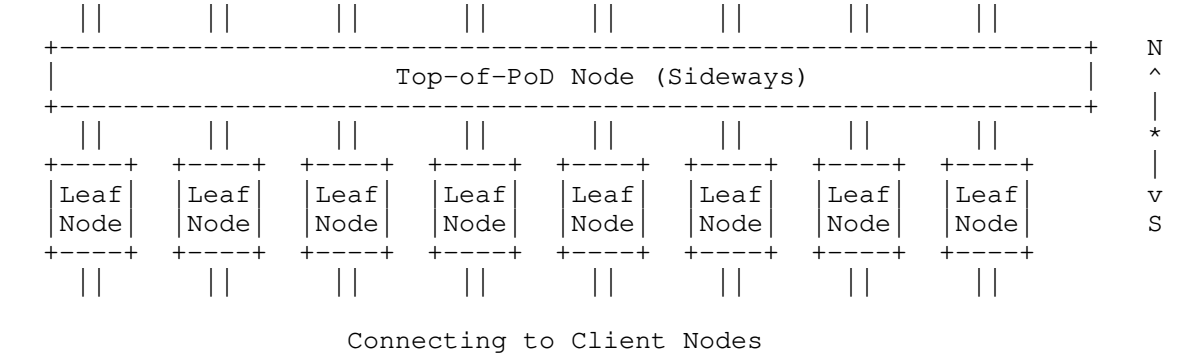


Figure 7: Side View of a PoD, K_TOP=8, K_LEAF=6

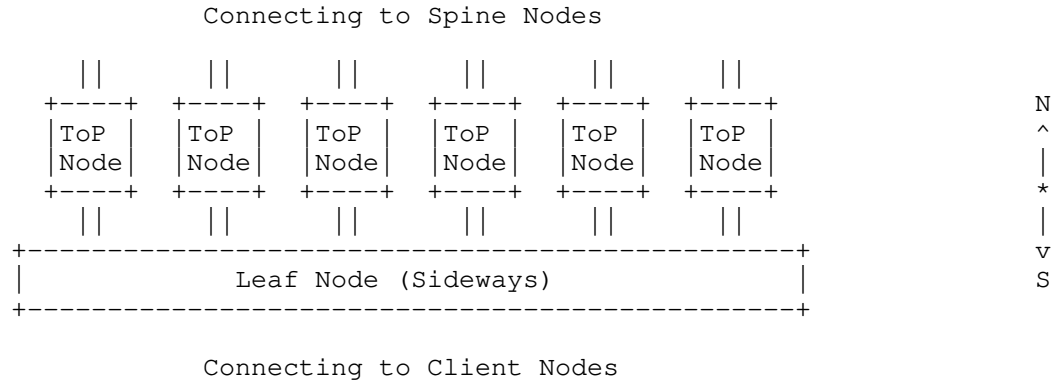


Figure 8: Other Side View of a PoD, K_TOP=8, K_LEAF=6, 90 Degree Turn in E-W Plane from the Previous Figure

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

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

To continue, the PoDs are interconnected with each other through a ToF node at the very top or the north edge of the fabric. The resulting ToF is *not* partitioned if, and only if (IIF), every PoD top level node (spine) is connected to every ToF Node. This topology is also referred to as a single plane configuration and is quite popular due to its simplicity. In order to reach a 1:1 connectivity ratio between the ToF and the leaves, it results that there are K_TOP ToF nodes, because each port of a ToP node connects to a different ToF node, and K_LEAF ToP nodes for the same reason. Consequently, it will take at least (P * K_LEAF) ports on a ToF node to connect to each of the K_LEAF ToP nodes of the P PoDs. Figure 9 illustrates this, looking at P=3 PoDs from above and 2 sides. The large view is the one from above, with the 8 ToF of 3*6 ports each interconnecting the PoDs, every ToP Node being connected to every ToF node.

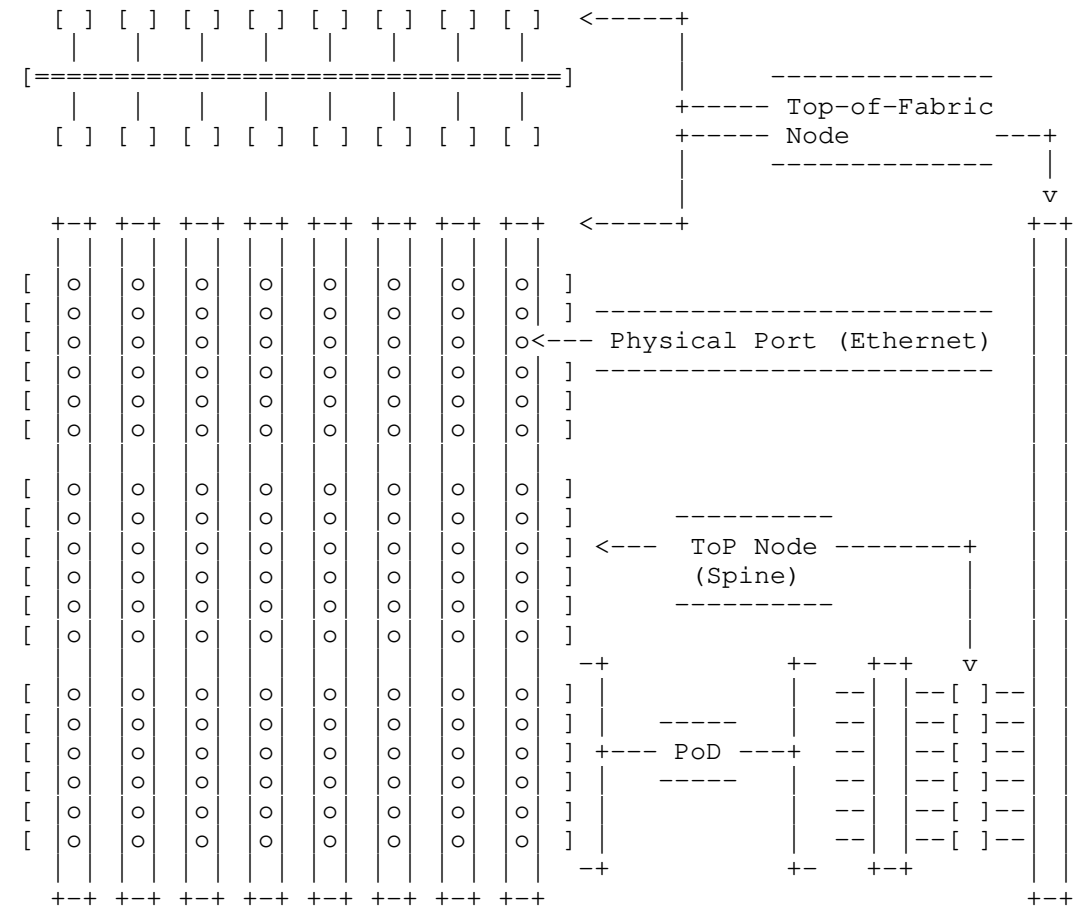


Figure 9: Fabric Spines and TOFs in Single Plane Design, 3 PoDs

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

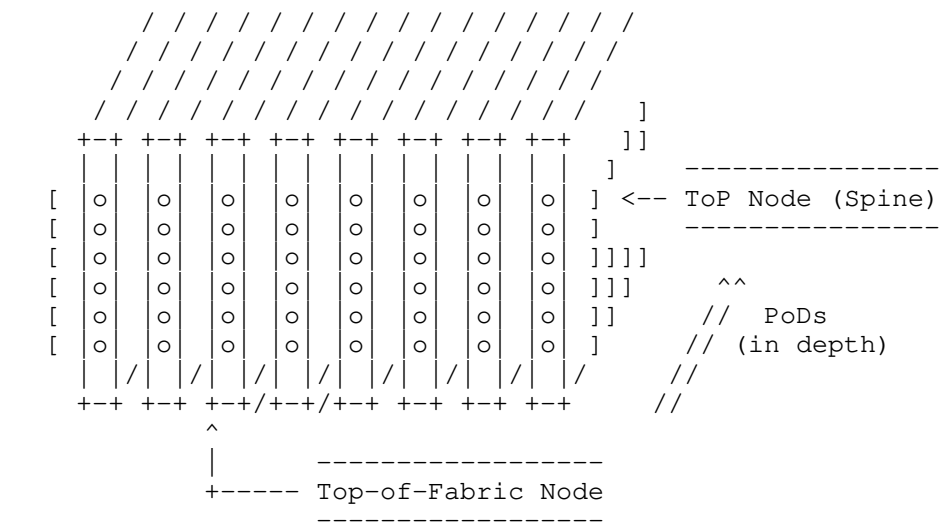


Figure 10: Collapsed Northern View of a Fabric for Any Number of PoDs

As simple as a single plane deployment is, it introduces a limit due to the bound on the available radix of the ToF nodes that has to be at least P * K_LEAF. Nevertheless, it will become clear that a distinct advantage of a connected or non-partitioned ToF is that all failures can be resolved by simple, non-transitive, positive disaggregation (i.e., nodes advertising more specific prefixes with the default to the level below them that is, however, not propagated further down the fabric) as described in Section 6.5.1 . In other words, non-partitioned ToF nodes can always reach nodes below or withdraw the routes from PoDs they cannot reach unambiguously. And with this, positive disaggregation can heal all failures and still allow all the ToF nodes to be aware of each other via south reflection. Disaggregation will be explained in further detail in Section 6.5.

In order to scale beyond the "single plane limit", the ToF can be partitioned into N number of identically wired planes where N is an integer divider of K_LEAF. The 1:1 ratio and the desired symmetry are still served, this time with (K_TOP * N) ToF nodes, each of (P * K_LEAF / N) ports. N=1 represents a non-partitioned Spine and N=K_LEAF is a maximally partitioned Spine. Further, if R is any integer divisor of K_LEAF, then N=K_LEAF/R is a feasible number of planes and R a redundancy factor that denotes the number of independent paths between 2 leaves within a plane. It proves convenient for deployments to use a radix for the leaf nodes that is a power of 2 so they can pick a number of planes that is a lower power of 2. The example in Figure 11 splits the Spine in 2 planes

with a redundancy factor R=3, meaning that there are 3 non-intersecting paths between any leaf node and any ToF node. A ToF node must have, in this case, at least 3*P ports, and be directly connected to 3 of the 6 ToP nodes (spines) in each PoD. The ToP nodes are represented horizontally with K_TOP=8 ports northwards each.

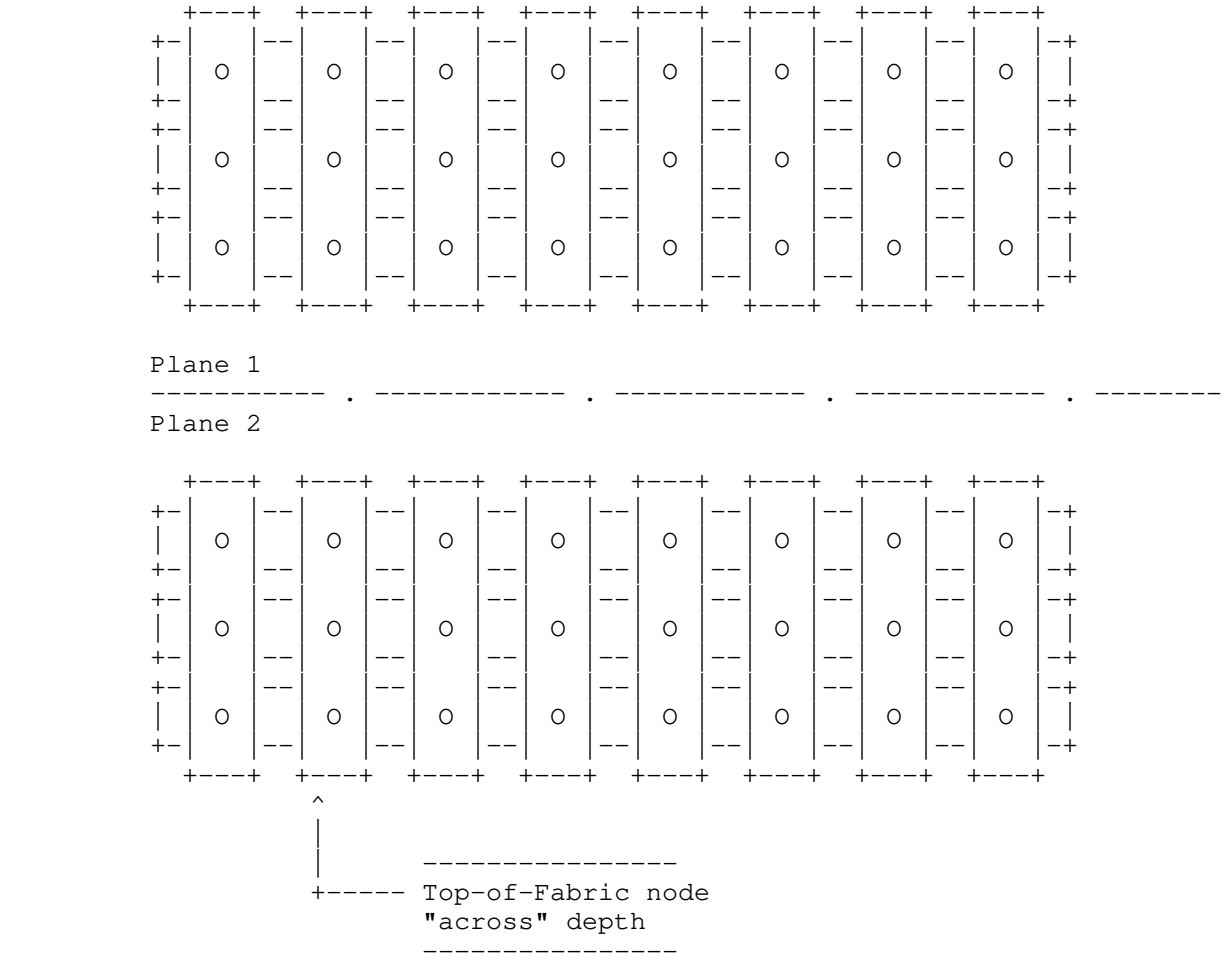


Figure 11: Northern View of a Multi-Plane ToF Level, K_LEAF=6, N=2

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

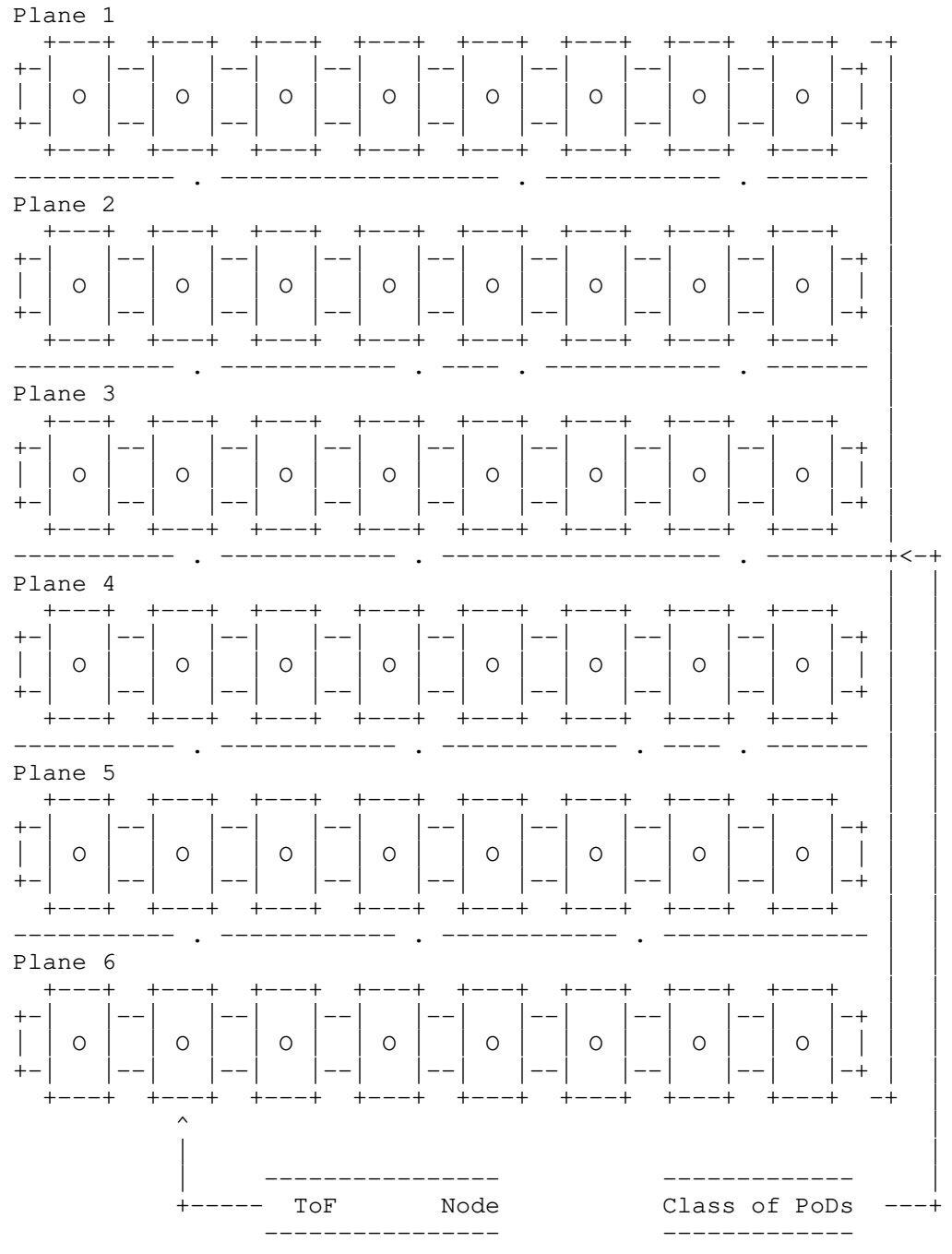


Figure 12: Northern View of a Maximally Partitioned ToF Level, R=1

5.3. Fallen Leaf Problem

As mentioned earlier, RIFT exhibits an anisotropic behavior tailored for fabrics with a North / South orientation and a high level of interleaving paths. A non-partitioned fabric makes a total loss of connectivity between a ToF node at the north and a leaf node at the south a very rare but yet possible occasion that is fully healed by positive disaggregation as described in Section 6.5.1. In large fabrics or fabrics built from switches with low radix, the ToF may often become partitioned in planes which makes the occurrence of having a given leaf being only reachable from a subset of the ToF nodes more likely to happen. This makes some further considerations necessary.

A "Fallen Leaf" is a leaf that can be reached by only a subset of ToF nodes due to missing connectivity. If R is the redundancy factor, then it takes at least R breakages to reach a "Fallen Leaf" situation.

In a maximally partitioned fabric, the redundancy factor is R=1, so any breakage in the fabric will cause one or more fallen leaves in the affected plane. R=2 guarantees that a single breakage will not cause a fallen leaf. However, not all cases require disaggregation. The following cases do not require particular action:

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

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

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

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

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

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

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

These abstractions are rolled back into a simplified example that shows that in Figure 3 the loss of link between spine node 3 and leaf node 3 will make leaf node 3 a fallen leaf for ToF nodes in plane C. Worse, if the cabling was never present in the first place, plane C will not even be able to know that such a fallen leaf exists. Hence partitioning without further treatment results in two grave problems:

* Leaf node 1 trying to route to leaf node 3 must not choose spine node 3 in plane C as its next hop since it will inevitably drop the packet when forwarding using default routes or do excessive bow-tying. This information must be in its routing table.

* A path computation trying to deal with the problem by distributing host routes may only form paths through leaves. The flooding of information about leaf node 3 would have to go up to ToF nodes in planes A, B, and D and then "loopback" over other leaves to ToF C leading in extreme cases to traffic for leaf node 3 when presented to plane C taking an "inverted fabric" path where leaves start to serve as ToFs, at least for the duration of a protocol's convergence.

5.4. Discovering Fallen Leaves

When aggregation is used, RIFT deals with fallen leaves by ensuring that all the ToF nodes share the same north topology database. This happens naturally in single plane design by the means of northbound flooding and south reflection but needs additional considerations in multi-plane fabrics. To enable routing to fallen leaves in multi-plane designs, RIFT requires additional interconnection across planes between the ToF nodes, e.g., using rings as illustrated in Figure 13. Other solutions are possible but they either need more cabling or end up having much longer flooding paths and/or single points of failure.

In detail, by reserving at least two ports on each ToF node it is possible to connect them together by interplane bi-directional rings as illustrated in Figure 13. The rings will be used to exchange full north topology information between planes. All ToFs having the same north topology allows by the means of transitive, negative disaggregation described in Section 6.5.2 to efficiently fix any possible fallen leaf scenario. Somewhat as a side-effect, the exchange of information fulfills the requirement for a full view of the fabric topology at the ToF level, without the need to collate it from multiple points.

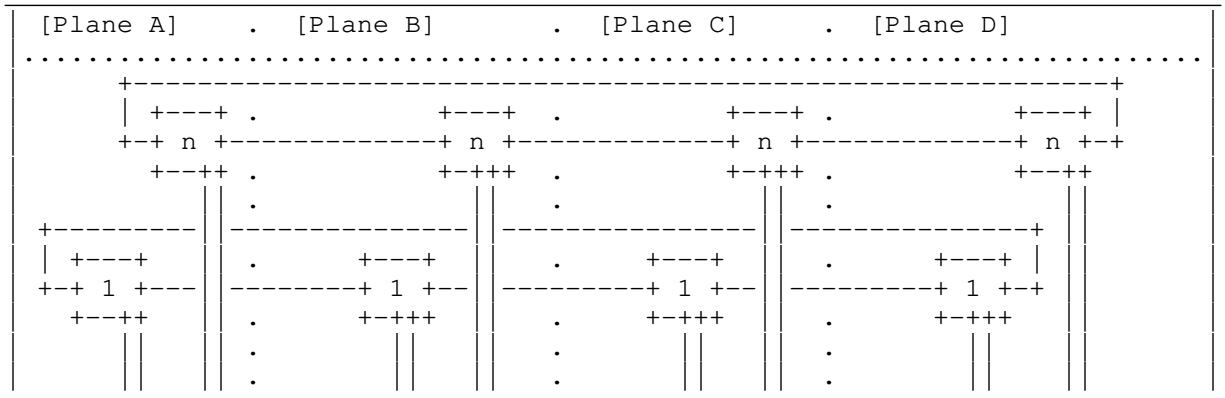


Figure 13: Using rings to bring all planes and at the ToF bind them

5.5. Addressing the Fallen Leaves Problem

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

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

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

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

originator with the special consideration of the negative prefix removing certain next hop neighbors. The specific procedures will be explained in detail in Section 6.5.2.3.

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

On the other hand, when the fabric is partitioned in planes, the positive disaggregation from ToF nodes in different planes do not reach the ToP switches in the affected plane and cannot solve the fallen leaves problem. In other words, a breakage in a plane can only be solved in that plane. Also, the selection of the plane for a packet typically occurs at the leaf level and the disaggregation must be transitive and reach all the leaves. In that case, the negative disaggregation is necessary. The details on the RIFT approach to deal with fallen leaves in an optimal way are specified in Section 6.5.2.

6. Specification

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

The FSMs, as usual, are presented as states a neighbor can assume, events that can occur, and the corresponding actions performed when transitioning between states on event processing.

Actions are performed before the end state is assumed.

The FSMs can queue events against itself to chain actions or against other FSMs in the specification. Events are always processed in the sequence they have been queued.

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

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

Any attempt to transition from a state towards another on reception of an event where no action is specified MUST be considered an unrecoverable error and the protocol MUST reset all adjacencies and discard all the state (i.e., force the FSM back to _OneWay_ and flush all of the queues holding flooding information).

The data structures and FSMs described in this document are conceptual and do not have to be implemented precisely as described here, i.e., an implementation is considered conforming as long as it supports the described functionality and exhibits externally observable behavior equivalent to the behavior of the standardized FSMs.

The FSMs can use "timers" for different situations. Those timers are started through actions and their expiration leads to queuing of corresponding events to be processed.

The term "holdtime" is used often as short-hand for "holddown timer" and signifies either the length of the holding down period or the timer used to expire after such period. Such timers are used to "hold down" state within an FSM that is cleaned if the machine triggers a _HoldtimeExpired_ event.

6.1. Transport

All normative RIFT packet structures and their contents are defined in the Thrift [thrift] models in Appendix B. The packet structure itself is defined in _ProtocolPacket_ which contains the packet header in _PacketHeader_ and the packet contents in _PacketContent_. _PacketContent_ is a union of the LIE, TIE, TIDE, and TIRE packets which are subsequently defined in _LIEPacket_, _TIEPacket_, _TIDEPacket_, and _TIREPacket_ respectively.

Further, in terms of bits on the wire, it is the _ProtocolPacket_ that is serialized and carried in an envelope defined in Section 6.9.3 within a UDP frame that provides security and allows validation/modification of several important fields without Thrift de-serialization for performance and security reasons. Security model and procedures are further explained in Section 9.

6.2. Link (Neighbor) Discovery (LIE Exchange)

RIFT LIE exchange auto-discovers neighbors, negotiates ZTP parameters and discovers miscablings. The formation progresses under normal conditions from _OneWay_ to _TwoWay_ and then _ThreeWay_ state at which point it is ready to exchange TIEs per Section 6.3. The adjacency exchanges ZTP information (Section 6.7) in any of the states, i.e. it is not necessary to reach _ThreeWay_ for zero-touch

provisioning to operate.

RIFT supports any combination of IPv4 and IPv6 addressing on the fabric with the additional capability for forwarding paths that are capable of forwarding IPv4 packets in presence of IPv6 addressing only.

IPv4 LIE exchange happens over well-known administratively locally scoped and configured or otherwise well-known IPv4 multicast address [RFC2365]. For IPv6 [RFC8200] exchange is performed over link-local multicast scope [RFC4291] address which is configured or otherwise well-known. In both cases a destination UDP port defined in the schema Appendix B.2 is used unless configured otherwise. LIEs MUST be sent with an IPv4 Time to Live (TTL) or an IPv6 Hop Limit (HL) of either 1 or 255 to prevent RIFT information reaching beyond a single L3 next-hop in the topology. LIEs SHOULD be sent with network control precedence unless an implementation is prevented from doing so [RFC2474].

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

An implementation may listen and send LIEs on IPv4 and/or IPv6 multicast addresses. A node MUST NOT originate LIEs on an address family if it does not process received LIEs on that family. LIEs on the same link are considered part of the same LIE FSM independent of the address family they arrive on. The LIE source address may not identify the peer uniquely in unnumbered or link-local address cases so the response transmission MUST occur over the same interface the LIEs have been received on. A node may use any of the adjacency's source addresses it saw in LIEs on the specific interface during adjacency formation to send TIEs (Section 6.3.3). That implies that an implementation MUST be ready to accept TIEs on all addresses it used as source of LIE frames.

A simplified version MAY be implemented on platforms with limited or no multicast support (e.g. IoT devices) by sending and receiving LIE frames on IPv4 subnet broadcast addresses or IPv6 all routers multicast address. However, this technique is less optimal and presents a wider attack surface from a security perspective.

A _ThreeWay_ adjacency (as defined in the glossary) over any address family implies support for IPv4 forwarding if the _ipv4_forwarding_capable_ flag in _LinkCapabilities_ is set to true. In the absence of IPv4 LIEs with _ipv4_forwarding_capable_ set to true, a node MUST forward IPv4 packets using gateways discovered on IPv6-only links advertising this capability. The mechanism to

discover the corresponding IPv6 gateway is out of scope for this specification and may be implementation specific. It is expected that the whole fabric supports the same type of forwarding of address families on all the links, any other combination is outside the scope of this specification. If IPv4 forwarding is supported on an interface, _ipv4_forwarding_capable_ MUST be set to true for all LIEs advertised from that interface. If IPv4 and IPv6 LIEs indicate contradicting information, protocol behavior is unspecified.

Operation of a fabric where only some of the links are supporting forwarding on an address family or have an address in a family and others do not is outside the scope of this specification.

Any attempt to construct IPv6 forwarding over IPv4 only adjacencies is outside this specification.

Table 1 outlines protocol behavior pertaining to LIE exchange over different address family combinations. Table 2 outlines the way in which neighbors forward traffic as it pertains to the _ipv4_forwarding_capable_ flag setting across the same address family combinations.

The specific forwarding implementation to support the described behavior is out of scope for this document.

Local Neighbor AF	Remote Neighbor AF	LIE Exchange Behavior
IPv4	IPv4	LIEs and TIEs are exchanged over IPv4 only. The local neighbor receives TIEs from remote neighbors on any of the LIE source addresses.
IPv6	IPv6	LIEs and TIEs are exchanged over IPv6 only. The local neighbor receives TIEs from remote neighbors on any of the LIE source addresses.
IPv4, IPv6	IPv6	The local neighbor sends LIEs for both IPv4 and IPv6 while the remote neighbor only sends LIEs for IPv6. The resulting adjacency will exchange TIEs over IPv6 on any of the IPv6 LIE source addresses.
IPv4, IPv6	IPv4, IPv6	LIEs and TIEs are exchanged over IPv6 and IPv4. TIEs are received on any of the IPv4 or IPv6 LIE source addresses. The local neighbor receives TIEs from the remote neighbors on any of the IPv4 or IPv6 LIE source addresses.

Table 1: Control Plane Behavior for Neighbor AF Combinations

Local Neighbor AF	Remote Neighbor AF	Forwarding Behavior
IPv4	IPv4	Both nodes are required to set the _ipv4_forwarding_capable_ flag to true. Only IPv4 traffic can be forwarded.
IPv6	IPv6	If either neighbor sets _ipv4_forwarding_capable_ to false, only IPv6 traffic can be forwarded. If both neighbors set _ipv4_forwarding_capable_ to true, IPv4 traffic is also forwarded via IPv6 gateways.
IPv4, IPv6	IPv6	If the remote neighbor sets _ipv4_forwarding_capable_ to false, only IPv6 traffic can be forwarded. If both neighbors set _ipv4_forwarding_capable_ to true, IPv4 traffic is also forwarded via IPv6 gateways.
IPv4, IPv6	IPv4, IPv6	IPv4 and IPv6 traffic can be forwarded. If IPv4 and IPv6 LIEs advertise conflicting _ipv4_forwarding_capable_ flags, the behavior is unspecified.

Table 2: Forwarding Behavior for Neighbor AF Combinations

The protocol does *not* support selective disabling of address families after adjacency formation, disabling IPv4 forwarding capability or any local address changes in _ThreeWay_ state, i.e. if a link has entered ThreeWay IPv4 and/or IPv6 with a neighbor on an adjacency and it wants to stop supporting one of the families or change any of its local addresses or stop IPv4 forwarding, it MUST tear down and rebuild the adjacency. It MUST also remove any state it stored about the remote side of the adjacency such as associated LIE source addresses.

Unless ZTP as described in Section 6.7 is used, each node is provisioned with the level at which it is operating and advertises it in the _level_ of the _PacketHeader_ schema element. It MAY be also provisioned with its PoD. If level is not provisioned, it is not present in the optional _PacketHeader_ schema element and established by ZTP procedures if feasible. If PoD is not provisioned, it is governed by the _LIEPacket_ schema element assuming the

_common.default_pod_ value. This means that switches except ToF do not need to be configured at all. Necessary information to configure all values is exchanged in the _LIEPacket_ and _PacketHeader_ or derived by the node automatically.

Further definitions of leaf flags are found in Section 6.7 given they have implications in terms of level and adjacency forming here. Leaf flags are carried in _HierarchyIndications_.

A node MUST form a _ThreeWay_ adjacency if at a minimum the following first order logic conditions are satisfied on a LIE packet as specified by the _LIEPacket_ schema element and received on a link (such a LIE is considered a "minimally valid" LIE). Observe that depending on the FSM involved and its state further conditions may be checked and even a minimally valid LIE can be considered ultimately invalid if any of the additional conditions fail.

1. the neighboring node is running the same major schema version as indicated in the _major_version_ element in _PacketHeader_ *and*

2. the neighboring node uses a valid System ID (i.e. value different from _IllegalSystemID_) in the _sender_ element in _PacketHeader_ *and*

3. the neighboring node uses a different System ID than the node itself *and*

4. (the advertised MTU values in the _LiePacket_ element match on both sides while a missing MTU in the _LiePacket_ element is interpreted as _default_mtu_size_) *and*

5. both nodes advertise defined level values in _level_ element in _PacketHeader_ *and*

6. [

   i) the node is at _leaf_level_ value and has no _ThreeWay_ adjacencies already to nodes at Highest Adjacency _ThreeWay_ (HAT as defined later in Section 6.7.1) with level different than the adjacent node *or*

   ii) the node is not at _leaf_level_ value and the neighboring node is at _leaf_level_ value *or*

   iii) both nodes are at _leaf_level_ values *and* both indicate support for Section 6.8.9 *or*

iv) neither node is at _leaf_level_ value and the neighboring node is at most one level difference away

].

LIEs arriving with IPv4 Time to Live (TTL) or an IPv6 Hop Limit (HL) different than 1 or 255 MUST be ignored.

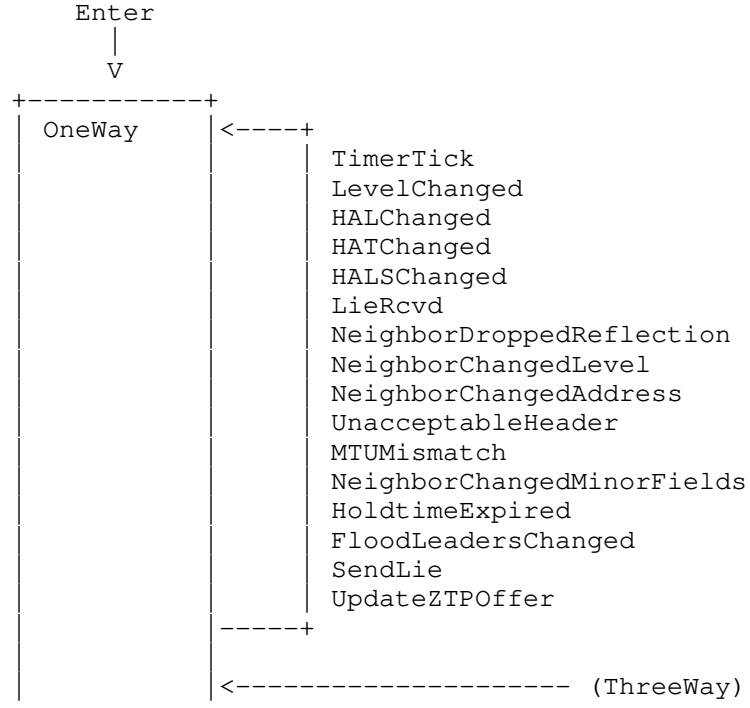
6.2.1. LIE Finite State Machine

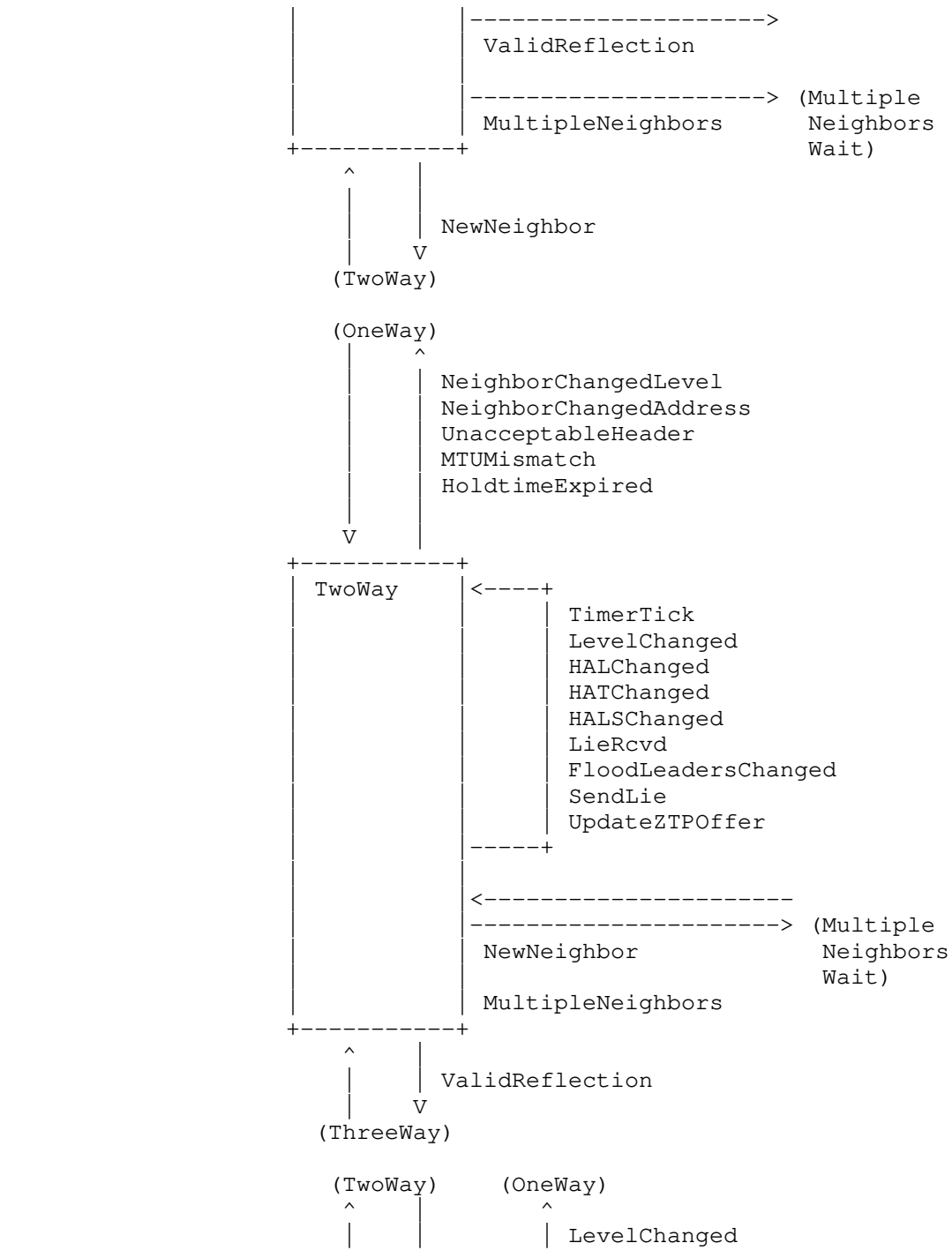
This section specifies the precise, normative LIE FSM which is given as well in Figure 14. Additionally, some sets of actions repeat often and are hence summarized into well-known procedures.

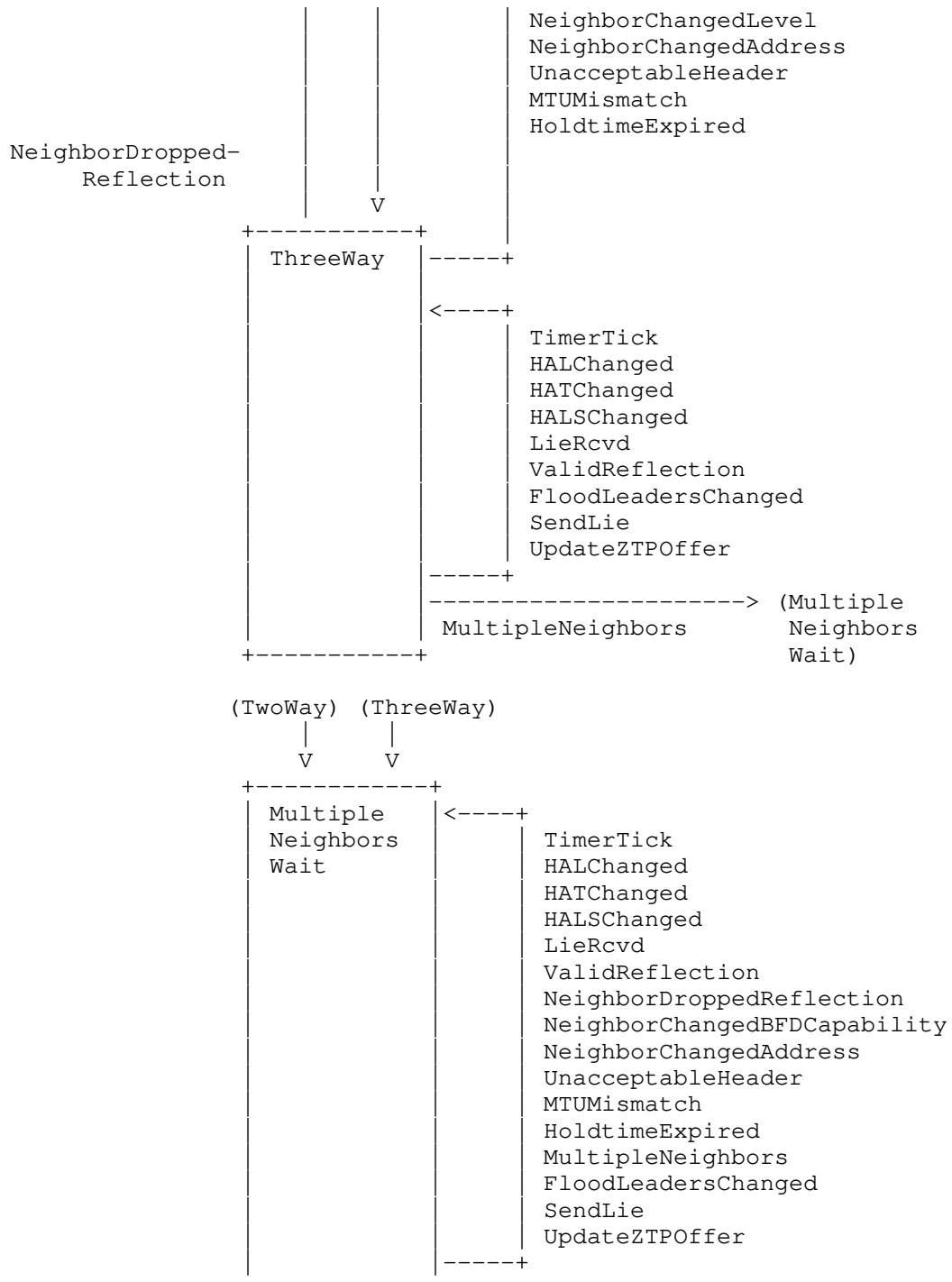
Events generated are fairly fine grained, especially when indicating problems in adjacency forming conditions to simplify tracking of problems in deployment.

Initial state is _OneWay_.

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




ValidReflection
MultipleNeighbors
(Multiple Neighbors Wait)
NewNeighbor
(TwoWay)
(OneWay)
NeighborChangedLevel
NeighborChangedAddress
UnacceptableHeader
MTUMismatch
HoldtimeExpired
TwoWay
TimerTick
LevelChanged
HALChanged
HATChanged
HALSChanged
LieRcvd
FloodLeadersChanged
SendLie
UpdateZTPOffer
NewNeighbor
(Multiple Neighbors Wait)
MultipleNeighbors
ValidReflection
(ThreeWay)
(TwoWay)
(OneWay)
LevelChanged


NeighborChangedLevel
NeighborChangedAddress
UnacceptableHeader
MTUMismatch
HoldtimeExpired
NeighborDropped-
Reflection
ThreeWay
TimerTick
HALChanged
HATChanged
HALSChanged
LieRcvd
ValidReflection
FloodLeadersChanged
SendLie
UpdateZTPOffer
MultipleNeighbors
(Multiple
Neighbors
Wait)
(TwoWay)
(ThreeWay)
Multiple
Neighbors
Wait
TimerTick
HALChanged
HATChanged
HALSChanged
LieRcvd
ValidReflection
NeighborDroppedReflection
NeighborChangedBFDCapability
NeighborChangedAddress
UnacceptableHeader
MTUMismatch
HoldtimeExpired
MultipleNeighbors
FloodLeadersChanged
SendLie
UpdateZTPOffer

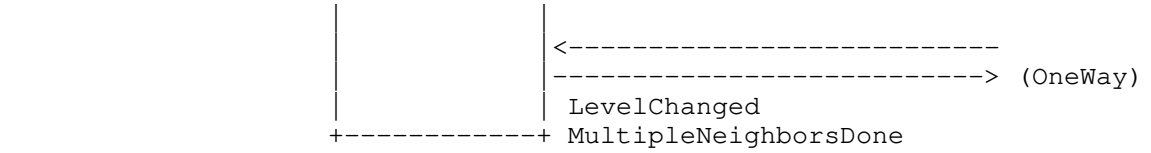


Figure 14: LIE FSM

The following words are used for well-known procedures:

* PUSH Event: queues an event to be executed by the FSM upon exit of this action

* CLEANUP: The FSM *conceptually* holds a 'current neighbor' variable that contains information received in the remote node's LIE that is processed against LIE validation rules. In the event that the LIE is considered to be invalid, the existing state held by 'current neighbor' MUST be deleted.

* SEND_LIE: create and send a new LIE packet

  1. reflecting the _neighbor_ element as described in ValidReflection and

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

  3. setting _you_are_flood_repeater_ variable to the computed value

* PROCESS_LIE:

  1. if LIE has a major version not equal to this node's major version *or* System ID equal to (this node's System ID or _IllegalSystemID_) then CLEANUP else

  2. if both sides advertise MTU values and the MTU in the received LIE does not match the MTU advertised by the local system *or* at least one of the nodes does not advertise an MTU value and the advertising node's LIE does not match the _default_mtu_size_ of the system not advertising an MTU then CLEANUP, PUSH UpdateZTPOffer, PUSH MTUMismatch else

  3. if the LIE has an undefined level *or* this node's level is undefined *or* this node is a leaf and remote level is lower than HAT *or* (the LIE's level is not leaf *and* its difference is more than one from this node's level) then CLEANUP, PUSH UpdateZTPOffer, PUSH UnacceptableHeader else

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

    1. if current neighbor System ID differs from LIE's System ID then PUSH MultipleNeighbors else

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

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

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

    5. CHECK_THREE_WAY

* CHECK_THREE_WAY: if current state is _OneWay_ do nothing else

    1. if LIE packet does not contain neighbor then if current state is _ThreeWay_ then PUSH NeighborDroppedReflection else

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

States:

* OneWay: initial state the FSM is starting from. In this state the router did not receive any valid LIEs from a neighbor.

* TwoWay: that state is entered when a node has received a minimally valid LIE from a neighbor but not a ThreeWay valid LIE.

* ThreeWay: this state signifies that _ThreeWay_ valid LIEs from a neighbor have been received. On achieving this state the link can be advertised in _neighbors_ element in _NodeTIEElement_.

* MultipleNeighborsWait: occurs normally when more than two nodes become aware of each other on the same link or a remote node is quickly reconfigured or rebooted without regressing to _OneWay_ first. Each occurrence of the event SHOULD generate notification to help operational deployments.

Events:

* TimerTick: one second timer tick, i.e., the event is provided to the FSM once a second by an implementation-specific mechanism that is outside the scope of this specification. This event is quietly ignored if the relevant transition does not exist.

* LevelChanged: node's level has been changed by ZTP or configuration. This is provided by the ZTP FSM.

* HALChanged: best HAL computed by ZTP has changed. This is provided by the ZTP FSM.

* HATChanged: HAT computed by ZTP has changed. This is provided by the ZTP FSM.

* HALSChanged: set of HAL offering systems computed by ZTP has changed. This is provided by the ZTP FSM.

* LieRcvd: received LIE on the interface.

* NewNeighbor: new neighbor is present in the received LIE.

* ValidReflection: received valid reflection of this node from neighbor, i.e. all elements in _neighbor_ element in _LiePacket_ have values corresponding to this link.

* NeighborDroppedReflection: lost previously held reflection from neighbor, i.e. _neighbor_ element in _LiePacket_ does not correspond to this node or is not present.

* NeighborChangedLevel: neighbor changed advertised level from the previously held one.

* NeighborChangedAddress: neighbor changed IP address, i.e. LIE has been received from an address different from previous LIEs. Those changes will influence the sockets used to listen to TIEs, TIREs, TIDEs.

* UnacceptableHeader: Unacceptable header received.

* MTUMismatch: MTU mismatched.

* NeighborChangedMinorFields: minor fields changed in neighbor's LIE.

* HoldtimeExpired: adjacency holddown timer expired.

* MultipleNeighbors: more than one neighbor is present on interface

* MultipleNeighborsDone: multiple neighbors timer expired.

* FloodLeadersChanged: node's election algorithm determined new set of flood leaders.

* SendLie: send a LIE out.

* UpdateZTPOffer: update this node's ZTP offer. This is sent to the ZTP FSM.

Actions:

* on HATChanged in _OneWay_ finishes in OneWay: store HAT

* on FloodLeadersChanged in _OneWay_ finishes in OneWay: update _you_are_flood_repeater_ LIE elements based on flood leader election results

* on UnacceptableHeader in _OneWay_ finishes in OneWay: no action

* on NeighborChangedMinorFields in _OneWay_ finishes in OneWay: no action

* on SendLie in _OneWay_ finishes in OneWay: SEND_LIE

* on HALSChanged in _OneWay_ finishes in OneWay: store HALS

* on MultipleNeighbors in _OneWay_ finishes in MultipleNeighborsWait: start multiple neighbors timer with interval _multiple_neighbors_lie_holdtime_multipler_ * _default_lie_holdtime_

* on NeighborChangedLevel in _OneWay_ finishes in OneWay: no action

* on LieRcvd in _OneWay_ finishes in OneWay: PROCESS_LIE

* on MTUMismatch in _OneWay_ finishes in OneWay: no action

* on ValidReflection in _OneWay_ finishes in ThreeWay: no action

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

* on HALChanged in _OneWay_ finishes in OneWay: store new HAL

* on HoldtimeExpired in _OneWay_ finishes in OneWay: no action

* on NeighborChangedAddress in _OneWay_ finishes in OneWay: no action
* on NewNeighbor in _OneWay_ finishes in TwoWay: PUSH SendLie event
* on UpdateZTPOffer in _OneWay_ finishes in OneWay: send offer to ZTP FSM
* on NeighborDroppedReflection in _OneWay_ finishes in OneWay: no action
* on TimerTick in _OneWay_ finishes in OneWay: PUSH SendLie event
* on FloodLeadersChanged in _TwoWay_ finishes in TwoWay: update _you_are_flood_repeater_ LIE elements based on flood leader election results
* on UpdateZTPOffer in _TwoWay_ finishes in TwoWay: send offer to ZTP FSM
* on NewNeighbor in _TwoWay_ finishes in MultipleNeighborsWait: PUSH SendLie event
* on ValidReflection in _TwoWay_ finishes in ThreeWay: no action
* on LieRcvd in _TwoWay_ finishes in TwoWay: PROCESS_LIE
* on UnacceptableHeader in _TwoWay_ finishes in OneWay: no action
* on HALChanged in _TwoWay_ finishes in TwoWay: store new HAL
* on HoldtimeExpired in _TwoWay_ finishes in OneWay: no action
* on LevelChanged in _TwoWay_ finishes in TwoWay: update level with event value
* on TimerTick in _TwoWay_ finishes in TwoWay: PUSH SendLie event, if last valid LIE was received more than _holdtime_ ago as advertised by neighbor then PUSH HoldtimeExpired event
* on HATChanged in _TwoWay_ finishes in TwoWay: store HAT
* on NeighborChangedLevel in _TwoWay_ finishes in OneWay: no action
* on HALSChanged in _TwoWay_ finishes in TwoWay: store HALS
* on MTUMismatch in _TwoWay_ finishes in OneWay: no action

* on NeighborChangedAddress in _TwoWay_ finishes in OneWay: no action

* on SendLie in _TwoWay_ finishes in TwoWay: SEND_LIE

* on MultipleNeighbors in _TwoWay_ finishes in MultipleNeighborsWait: start multiple neighbors timer with interval _multiple_neighbors_lie_holdtime_multipler_ * _default_lie_holdtime_

* on TimerTick in _ThreeWay_ finishes in ThreeWay: PUSH SendLie event, if last valid LIE was received more than _holdtime_ ago as advertised by neighbor then PUSH HoldtimeExpired event

* on LevelChanged in _ThreeWay_ finishes in OneWay: update level with event value

* on HATChanged in _ThreeWay_ finishes in ThreeWay: store HAT

* on MTUMismatch in _ThreeWay_ finishes in OneWay: no action

* on UnacceptableHeader in _ThreeWay_ finishes in OneWay: no action

* on MultipleNeighbors in _ThreeWay_ finishes in MultipleNeighborsWait: start multiple neighbors timer with interval _multiple_neighbors_lie_holdtime_multipler_ * _default_lie_holdtime_

* on NeighborChangedLevel in _ThreeWay_ finishes in OneWay: no action

* on HALSChanged in _ThreeWay_ finishes in ThreeWay: store HALS

* on LieRcvd in _ThreeWay_ finishes in ThreeWay: PROCESS_LIE

* on FloodLeadersChanged in _ThreeWay_ finishes in ThreeWay: update _you_are_flood_repeater_ LIE elements based on flood leader election results, PUSH SendLie

* on NeighborDroppedReflection in _ThreeWay_ finishes in TwoWay: no action

* on HoldtimeExpired in _ThreeWay_ finishes in OneWay: no action

* on ValidReflection in _ThreeWay_ finishes in ThreeWay: no action

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

* on NeighborChangedAddress in _ThreeWay_ finishes in OneWay: no action

* on HALChanged in _ThreeWay_ finishes in ThreeWay: store new HAL

* on SendLie in _ThreeWay_ finishes in ThreeWay: SEND_LIE

* on MultipleNeighbors in MultipleNeighborsWait finishes in MultipleNeighborsWait: start multiple neighbors timer with interval _multiple_neighbors_lie_holdtime_multipler_ * _default_lie_holdtime_

* on FloodLeadersChanged in MultipleNeighborsWait finishes in MultipleNeighborsWait: update _you_are_flood_repeater_ LIE elements based on flood leader election results

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

* on ValidReflection in MultipleNeighborsWait finishes in MultipleNeighborsWait: no action

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

* on NeighborDroppedReflection in MultipleNeighborsWait finishes in MultipleNeighborsWait: no action

* on LieRcvd in MultipleNeighborsWait finishes in MultipleNeighborsWait: no action

* on UnacceptableHeader in MultipleNeighborsWait finishes in MultipleNeighborsWait: no action

* on NeighborChangedAddress in MultipleNeighborsWait finishes in MultipleNeighborsWait: no action

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

* on HATChanged in MultipleNeighborsWait finishes in MultipleNeighborsWait: store HAT

* on MTUMismatch in MultipleNeighborsWait finishes in MultipleNeighborsWait: no action

* on HALSChanged in MultipleNeighborsWait finishes in MultipleNeighborsWait: store HALS

* on HALChanged in MultipleNeighborsWait finishes in MultipleNeighborsWait: store new HAL

* on HoldtimeExpired in MultipleNeighborsWait finishes in MultipleNeighborsWait: no action

* on SendLie in MultipleNeighborsWait finishes in MultipleNeighborsWait: no action

* on MultipleNeighborsDone in MultipleNeighborsWait finishes in OneWay: no action

* on Entry into OneWay: CLEANUP

6.3. Topology Exchange (TIE Exchange)

6.3.1. Topology Information Elements

Topology and reachability information in RIFT is conveyed by TIEs.

The TIE exchange mechanism uses the port indicated by each node in the LIE exchange as _flood_port_ in _LIEPacket_ and the interface on which the adjacency has been formed as destination. TIEs MUST be sent with an IPv4 Time to Live (TTL) or an IPv6 Hop Limit (HL) of either 1 or 255 and also MUST be ignored if received with values different than 1 or 255. This prevents RIFT information from reaching beyond a single L3 next-hop in the topology. TIEs SHOULD be sent with network control precedence unless an implementation is prevented from doing so [RFC2474].

TIEs contain sequence numbers, lifetimes, and a type. Each type has ample identifying number space and information is spread across multiple TIEs with the same TIEElement type (this is true for all TIE types).

More information about the TIE structure can be found in the schema in Appendix B starting with _TIEPacket_ root.

6.3.2. Southbound and Northbound TIE Representation

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

The North TIEs hold all of the node's adjacencies and local prefixes while the South TIEs hold only all of the node's adjacencies, the default prefix with necessary disaggregated prefixes and local prefixes. Section 6.5 explains further details.

All TIE types are mostly symmetrical in both directions. The (Appendix B.3) defines the TIE types (i.e., the TIETypeType element) and their directionality (i.e., _direction_ within the _TIEID_ element).

As an example illustrating a databases holding both representations, the topology in Figure 2 with the optional link between spine 111 and spine 112 (so that the flooding on an East-West link can be shown) is shown below. Unnumbered interfaces are implicitly assumed and for simplicity, the key value elements which may be included in their South TIEs or North TIEs are not shown. First, in Figure 15 are the TIEs generated by some nodes.

```
ToF 21 South TIEs:
Node South TIE:
  NodeTIEElement(level=2,
    neighbors(
      (Spine 111, level 1, cost 1, links(...)),
      (Spine 112, level 1, cost 1, links(...)),
      (Spine 121, level 1, cost 1, links(...)),
      (Spine 122, level 1, cost 1, links(...))
    )
  )
Prefix South TIE:
  PrefixTIEElement(prefixes(0/0, metric 1), (::/0, metric 1))

Spine 111 South TIEs:
Node South TIE:
  NodeTIEElement(level=1,
    neighbors(
      (ToF 21, level 2, cost 1, links(...)),
      (ToF 22, level 2, cost 1, links(...)),
      (Spine 112, level 1, cost 1, links(...)),
```

```
            (Leaf111, level 0, cost 1, links(...)),
            (Leaf112, level 0, cost 1, links(...))
          )
        )
      Prefix South TIE:
        PrefixTIEElement(prefixes(0/0, metric 1), (::/0, metric 1))

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

      Spine 121 South TIEs:
      Node South TIE:
        NodeTIEElement(level=1,
          neighbors(
            (ToF 21, level 2, cost 1, links(...)),
            (ToF 22, level 2, cost 1, links(...)),
            (Leaf121, level 0, cost 1, links(...)),
            (Leaf122, level 0, cost 1, links(...))
          )
        )
      Prefix South TIE:
        PrefixTIEElement(prefixes(0/0, metric 1), (::/0, metric 1))

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

      Leaf112 North TIEs:
```

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

Figure 15: Example TIEs Generated in a 2 Level Spine-and-Leaf Topology

It may not be obvious here as to why the Node South TIEs contain all the adjacencies of the corresponding node. This will be necessary for algorithms further elaborated on in Section 6.3.9 and Section 6.8.7.

For Node TIEs to carry more adjacencies than fit into an MTU-sized packet, the element _neighbors_ may contain a different set of neighbors in each TIE. Those disjointed sets of neighbors MUST be joined during corresponding computation. However, if the following occurs across multiple Node TIEs

1. _capabilities_ do not match *or*

2. _flags_ values do not match *or*

3. same neighbor repeats in multiple TIEs with different values

The implementation is expected to use the value of any of the valid TIEs it received as it cannot control the arrival order of those TIEs.

The _miscabled_links_ element SHOULD be included in every Node TIE, otherwise the behavior is undefined.

A ToF node MUST include information on all other ToFs it is aware of through reflection. The _same_plane_tofs_ element is used to carry this information. To prevent MTU overrun problems, multiple Node TIEs can carry disjointed sets of ToFs which MUST be joined to form a single set.

Different TIE types are carried in _TIEElement_. Schema enum ‘common.TIETypeType‘ in _TIEID_ indicates which elements MUST be present in the _TIEElement_. In case of a mismatch between the _TIETypeType_ in the _TIEID_ and the present element, the unexpected elements MUST be ignored. In case of lack of expected element in the

TIE an error MUST be reported and the TIE MUST be ignored. The element _positive_disaggregation_prefixes_ and _positive_external_disaggregation_prefixes_ MUST be advertised southbound only and ignored in North TIEs. The element _negative_disaggregation_prefixes_ MUST be propagated according to Section 6.5.2 southwards towards lower levels to heal pathological upper-level partitioning, otherwise traffic loss may occur in multiplane fabrics. It MUST NOT be advertised within a North TIE and MUST be ignored otherwise.

6.3.3. Flooding

As described before, TIEs themselves are transported over UDP with the ports indicated in the LIE exchanges and using the destination address on which the LIE adjacency has been formed.

TIEs are uniquely identified by the _TIEID_ schema element. The _TIEID_ induces a total order achieved by comparing the elements in sequence defined in the element and comparing each value as an unsigned integer of corresponding length. The _TIEHeader_ element contains a _seq_nr_ element to distinguish newer versions of same TIE.

The TIEHEader can also carry an _origination_time_ schema element (for fabrics that utilize precision timing) which contains the absolute timestamp of when the TIE was generated and an _origination_lifetime_ to indicate the original lifetime when the TIE was generated. When carried, they can be used for debugging or security purposes (e.g. to prevent lifetime modification attacks).

_remaining_lifetime_ counts down to 0 from _origination_lifetime_. TIEs with lifetimes differing by less than _lifetime_diff2ignore_ MUST be considered EQUAL (if all other fields are equal). This constant MUST be larger than _purge_lifetime_ to avoid retransmissions.

This normative ordering methodology is described in Figure 16 and MUST be used by all implementations.

```
for each TIEPacket:
    TIEHeader = TIEPacket.TIEHeader
    TIEElement = TIEPacket.TIEElement

    seq_nr = TIEHeader.seq_nr

    TIEID = TIEHeader.TIEID
    direction = TIEID.direction

    # System ID
    originator = TIEID.originator

    # TIETypeType
    tietype = TIEID.tietype
    tie_nr = TIEID.tie_nr

    if X.direction > Y.direction:
        return X.direction
    else if X.direction < Y.direction:
        return Y.direction
    else if X.originator > Y.originator:
        return X.originator
    else if X.originator < Y.originator:
        return Y.originator
    else:
        if X.tietype == Y.tietype:
            if X.tie_nr == Y.tie_nr:
                if X.seq_nr == Y.seq_nr:
                    X.lifetime_left = X.remaining_lifetime - time since TIE was r
eceived
                    Y.lifetime_left = Y.remaining_lifetime - time since TIE was r
eceived

                    if absolute_value_of(X.lifetime_left - Y.lifetime_left) <= co
mmon.lifetime_diff2ignore:
                        return equal

                    else:
                        return TIE with largest lifetime_left
                else:
                    return X.seq_nr compared to Y.seq_nr
            else:
                return X.tie_nr compared to Y.tie_nr
        else:
            return X.TIEType compared to Y.TIEType
```

Figure 16: TIE Ordering

All valid TIE types are defined in _TIETypeType_. This enum indicates what TIE type the TIE is carrying. In case the value is not known to the receiver, the TIE MUST be re-flooded with scope

identical to the scope of a prefix TIE. This allows for future extensions of the protocol within the same major schema with types opaque to some nodes with some restrictions defined in Appendix B.

6.3.3.1. Normative Flooding Procedures

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

This section specifies the precise, normative flooding mechanism and can be omitted unless the reader is pursuing an implementation of the protocol or looks for a deep understanding of underlying information distribution mechanism.

Flooding Procedures are described in terms of the flooding state of an adjacency and resulting operations on it driven by packet arrivals. Implementations MUST implement a behavior that is externally indistinguishable from the FSMs and normative procedures given here.

RIFT does not specify any kind of flood rate limiting. To help with adjustment of flooding speeds the encoded packets provide hints to react accordingly to losses or overruns via _you_are_sending_too_quickly_ in the _LIEPacket_ and 'Packet Number' in the security envelope described in Section 6.9.3. Flooding of all corresponding topology exchange elements SHOULD be performed at the highest feasible rate but the rate of transmission MUST be throttled by reacting to packet elements and features of the system such as e.g. queue lengths or congestion indications in the protocol packets.

A node SHOULD NOT send out any topology information elements if the adjacency is not in a "ThreeWay" state. No further tightening of this rule is possible. For example, link buffering may cause both LIEs and TIEs/TIDEs/TIREs to be re-ordered.

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

TIEs generated by other nodes MUST be re-flooded. TIDEs and TIREs MUST NOT be re-flooded.

6.3.3.1.1. FloodState Structure per Adjacency

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

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

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

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

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

FILTERED_TIEDB:
A filtered view of TIEDB, which retains for consideration only those headers permitted by is_tide_entry_filtered and which either have a lifetime left > 0 or have no content.

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

TIE:
Describes either a full RIFT TIE or just the _TIEHeader_ or _TIEID_ equivalent as defined in Appendix B.3. The corresponding meaning is unambiguously contained in the context of each algorithm.

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

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

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

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

try_to_transmit_tie(TIE):
A. if not is_flood_filtered(TIE) then

1. remove TIE from TIES_RTX if present

2. if TIE with same key is found on TIES_ACK then

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

    b. remove TIE from TIES_ACK and add TIE to TIES_TX

3. else insert TIE into TIES_TX

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

tie_been_acked(TIE):
remove TIE from all collections.

remove_from_all_queues(TIE):
same as _tie_been_acked_.

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

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

clear_requests(TIEs):
remove all TIEs from TIES_REQ.

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

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

1. Elements on TIES_ACK should be processed with highest priority

2. TIES_TX

3. TIES_REQ and TIES_RTX should be processed with lowest priority

6.3.3.1.2. TIDEs

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

6.3.3.1.2.1. TIDE Generation

As given by timer constant, periodically generate TIDEs by:

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

a. NEXT_TIDE_ID = MIN_TIEID

b. while NEXT_TIDE_ID not equal to MAX_TIEID do

    1. HEADERS = Exactly TIRDEs_PER_PKT headers from FILTERED_TIEDB starting at NEXT_TIDE_ID, unless fewer than TIRDEs_PER_PKT remain, in which case all remaining headers.

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

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

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

    5. NEXT_TIDE_ID = END

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

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

The algorithm will intentionally enter the loop once and send a single TIDE even when the database is empty, otherwise no TIDEs would be sent for in case of empty database and break intended synchronization.

6.3.3.1.2.2. TIDE Processing

On reception of TIDEs the following processing is performed:

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

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

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

LASTPROCESSED: Last processed TIEID in TIDE

DBTIE: TIE in the Link State Database (LSDB) if found

a. LASTPROCESSED = TIDE.start_range

b. for every HEADER in TIDE do

1. DBTIE = find HEADER in current LSDB

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

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

4. LASTPROCESSED = HEADER

5. if DBTIE not found then

I) if originator is this node, then bump_own_tie

II) else put HEADER into REQKEYS

6. if DBTIE.HEADER < HEADER then

I) if originator is this node then bump_own_tie else

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

ii. else put HEADER into REQKEYS

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

8. if DBTIE.HEADER = HEADER then

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

II) else put HEADER into REQKEYS

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

d. for all TIEs in TXKEYS try_to_transmit_tie(TIE)

e. for all TIEs in REQKEYS request_tie(TIE)

f. for all TIEs in CLEARKEYS remove_from_all_queues(TIE)

6.3.3.1.3. TIREs

6.3.3.1.3.1. TIRE Generation

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

6.3.3.1.3.2. TIRE Processing

On reception of TIREs the following processing is performed:

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

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

ACKKEYS: Collection of TIEIDs that have been acked

DBTIE: TIE in the LSDB if found

a. for every HEADER in TIRE do

1. DBTIE = find HEADER in current LSDB

2. if DBTIE not found then do nothing

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

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

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

b. for all TIEs in TXKEYS try_to_transmit_tie(TIE)

c. for all TIEs in REQKEYS request_tie(TIE)

d. for all TIEs in ACKKEYS tie_been_acked(TIE)

6.3.3.1.4. TIEs Processing on Flood State Adjacency

On reception of TIEs the following processing is performed:

ACKTIE: TIE to acknowledge

TXTIE: TIE to transmit

DBTIE: TIE in the LSDB if found

a. DBTIE = find TIE in current LSDB

b. if DBTIE not found then

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

2. else insert TIE into LSDB and ACKTIE = TIE

else

1. if DBTIE.HEADER = TIE.HEADER then

i. if DBTIE has content already then ACKTIE = TIE

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

2. if DBTIE.HEADER < TIE.HEADER then

i. if originator is this node then bump_own_tie

ii. else insert TIE into LSDB and ACKTIE = TIE

3. if DBTIE.HEADER > TIE.HEADER then

i. if DBTIE has content already then TXTIE = DBTIE

ii. else ACKTIE = DBTIE

c. if TXTIE is set then try_to_transmit_tie(TXTIE)

d. if ACKTIE is set then ack_tie(TIE)

6.3.3.1.5. Sending TIEs

On a periodic basis all TIEs with lifetime left > 0 MUST be sent out on the adjacency, removed from TIES_TX list and requeued onto TIES_RTX list. The specific period is out of scope for this document.

6.3.3.1.6. TIEs Processing In LSDB

The Link State Database (LSDB) holds the most recent copy of TIEs received via flooding from according peers. Consecutively, after version tie-breaking by LSDB, a peer receives from the LSDB the newest versions of TIEs received by other peers and processes them (without any filtering) just like receiving TIEs from its remote peer. Such a publisher model can be implemented in several ways, either in a single thread of execution or in multiple parallel threads.

LSDB can be logically considered as the entity aging out TIEs, i.e. being responsible to discard TIEs that are stored longer than _remaining_lifetime_ on their reception.

LSDB is also expected to periodically re-originate the node's own TIEs. Originating at an interval significantly shorter than _default_lifetime_ is RECOMMENDED to prevent TIE expiration by other nodes in the network which can lead to instabilities.

6.3.4. TIE Flooding Scopes

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

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

A node's Node South TIEs, consisting of all node's adjacencies and prefix South TIEs limited to those related to default IP prefix and disaggregated prefixes, are flooded southbound in order to inform

nodes one level down of connectivity of the higher level as well as reachability to the rest of the fabric. In order to allow an E-W disconnected node in a given level to receive the South TIEs of other nodes at its level, every *NODE* South TIE is "reflected" northbound to the level from which it was received. It should be noted that East-West links are included in South TIE flooding (except at the ToF level); those TIEs need to be flooded to satisfy algorithms in Section 6.4. In that way nodes at same level can learn about each other using without a lower level except in case of leaf level. The precise, normative flooding scopes are given in Table 3. Those rules also govern what SHOULD be included in TIDEs on the adjacency. Again, East-West flooding scopes are identical to South flooding scopes except in case of ToF East-West links (rings) which are basically performing northbound flooding.

Node South TIE "south reflection" enables support of positive disaggregation on failures as described in in Section 6.5 and flooding reduction in Section 6.3.9.

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

Table 3: Normative Flooding Scopes

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

To illustrate these rules, consider using the topology in Figure 2, with the optional link between spine 111 and spine 112, and the associated TIEs given in Figure 15. The flooding from particular nodes of the TIEs is given in Table 4.

Local Node	Neighbor Node	TIEs Flooded from Local to Neighbor Node
Leaf111	Spine 112	Leaf111 North TIEs, Spine 111 Node South TIE
Leaf111	Spine 111	Leaf111 North TIEs, Spine 112 Node South TIE
...	...	...
Spine 111	Leaf111	Spine 111 South TIEs
Spine 111	Leaf112	Spine 111 South TIEs
Spine 111	Spine 112	Spine 111 South TIEs
Spine 111	ToF 21	Spine 111 North TIEs, Leaf111 North TIEs, Leaf112 North TIEs, ToF 22 Node South TIE
Spine 111	ToF 22	Spine 111 North TIEs, Leaf111 North TIEs, Leaf112 North TIEs, ToF 21 Node South TIE
...	...	...
ToF 21	Spine 111	ToF 21 South TIEs
ToF 21	Spine 112	ToF 21 South TIEs
ToF 21	Spine 121	ToF 21 South TIEs
ToF 21	Spine 122	ToF 21 South TIEs
...	...	...

Table 4: Flooding some TIEs from example topology

6.3.5. RAIN: RIFT Adjacency Inrush Notification

The optional RIFT Adjacency Inrush Notification (RAIN) mechanism helps to prevent adjacencies from being overwhelmed by flooding on restart or bring-up with many southbound neighbors. A node MAY set in its LIEs the corresponding _you_are_sending_too_quickly_ flag to indicate to the neighbor that it SHOULD flood Node TIEs with normal speed and significantly slow down the flooding of any other TIEs. The flag SHOULD be set only in the southbound direction. The receiving node SHOULD accommodate the request to lessen the flooding load on the affected node if south of the sender and should ignore the indication if north of the sender.

The distribution of Node TIEs at normal speed even at high load guarantees correct behavior of algorithms like disaggregation or default route origination. Furthermore though, the use of this bit presents an inherent trade-off between processing load and convergence speed since significantly slowing down flooding of northbound prefixes from neighbors for an extended time will lead to traffic losses.

6.3.6. Initial and Periodic Database Synchronization

The initial exchange of RIFT includes periodic TIDE exchanges that contain description of the link state database and TIREs which perform the function of requesting unknown TIEs as well as confirming reception of flooded TIEs. The content of TIDEs and TIREs is governed by Table 3.

6.3.7. Purging and Roll-Overs

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

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

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

TIE sequence numbers are rolled over using the method described in Appendix A. First sequence number of any spontaneously originated TIE (i.e. not originated to override a detected older copy in the network) MUST be a reasonably unpredictable random number (for example [RFC4086]) in the interval [0, 2^30-1] which will prevent otherwise identical TIE headers to remain "stuck" in the network with content different from TIE originated after reboot. In traditional link-state protocols this is delegated to a 16-bit checksum on packet content. RIFT avoids this design due to the CPU burden presented by computation of such checksums and additional complications tied to the fact that the checksum must be "patched" into the packet after the generation of the content, a difficult proposition in binary hand-crafted formats already and highly incompatible with model-based, serialized formats. The sequence number space is hence consciously chosen to be 64-bits wide to make the occurrence of a TIE with same sequence number but different content as much or even more unlikely than the checksum method. To emulate the "checksum behavior" an implementation could choose to compute a 64-bit checksum or hash function over the TIE content and use that as part of the first sequence number after reboot.

6.3.8. Southbound Default Route Origination

Under certain conditions nodes issue a default route in their South Prefix TIEs with costs as computed in Section 6.8.7.1.

A node X that

1. is *not* overloaded *and*

2. has southbound or East-West adjacencies

SHOULD originate in its south prefix TIE such a default route if and only if

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

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

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

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

A node originating a southbound default route SHOULD install a default discard route if it did not compute a default route during N-SPF. This basically means that the top of the fabric will drop traffic for unreachable addresses.

6.3.9. Northbound TIE Flooding Reduction

RIFT chooses only a subset of northbound nodes to propagate flooding and with that both balances it (to prevent 'hot' flooding links) across the fabric as well as reduces its volume. The solution is based on several principles:

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

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

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

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

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

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

7. it is expected that there will be from time to time a broken link between a parent and a grandparent, and in that case the parent is probably a poor FR due to its lower reliability. An algorithm may attempt to eliminate parents with broken northbound adjacencies first in order to reduce the number of FRs. Albeit it could be argued that relying on higher fanout FRs will slow flooding due to higher replication, load reliability of FR's links is likely a more pressing concern.

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

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

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

* let |NA(Node) be the set of Northbound adjacencies of node Node and CN(Node) be the cardinality of |NA(Node);

* let |SA(Node) be the set of Southbound adjacencies of node Node and CS(Node) be the cardinality of |SA(Node);

* let |P(Node) be the set of node Node's parents;

* let |G(Node) be the set of node Node's grandparents. Observe that |G(Node) = |P(|P(Node));

* let N be the child node at level L computing a set of FR;

* let P be a node at level L+1 and a parent node of N, i.e. bi-directionally reachable over adjacency ADJ(N, P);

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

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

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

* let RND be a 64-bit random number (for example [RFC4086]) generated by the system once on startup.

The algorithm consists of the following steps:

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

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

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

      where << is the circular shift operator.

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

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

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

   B. set i=0;

   C. while i < CN(N) do

      i) set j=i;

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

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

         b. set i=i+1;

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

         set k=k+1;

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

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

   A. while k > 0 do

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

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

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

      ii) set k=k-1;

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

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

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

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

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

         a. place P in FR set;

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

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

Additional rules for flooding reduction:

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

2. Upon a change of the flood repeater set, a node SHOULD send out LIEs that grant flood repeater status to newly promoted nodes before it sends LIEs that revoke the status to the nodes that have been newly demoted. This is done to prevent transient behavior where the full coverage of grandparents is not guaranteed. Such a condition is sometimes unavoidable in case of lost LIEs but it will correct itself though at possible transient reduction in flooding propagation speeds. The election can use the LIE FSM _FloodLeadersChanged_ event to notify LIE FSMs of necessity to update the sent LIEs.

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

4. A node receiving a TIE originated by a node for which it is not a flood repeater SHOULD NOT reflood such TIEs to its neighbors except for rules in Section 6.3.9, Paragraph 10, Item 6.

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

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

6.3.10. Special Considerations

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

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

6.4. Reachability Computation

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

To compute prefix reachability, a node runs conceptually a northbound and a southbound SPF. N-SPF and S-SPF notation denotes here the direction in which the computation front is progressing.

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

For reachability computation purposes, RIFT considers all parallel links between two nodes to be of the same cost advertised in the _cost_ element of _NodeNeighborsTIEElement_. In case the neighbor has multiple parallel links at different cost, the largest distance (highest numerical value) MUST be advertised. Given the range of thrift encodings, _infinite_distance_ is defined as the largest non-negative _MetricType_. Any link with metric larger than that (i.e. negative MetricType) MUST be ignored in computations. Any link with metric set to _invalid_distance_ MUST also be ignored in computation. In case of a negatively distributed prefix the metric attribute MUST be set to _infinite_distance_ by the originator and it MUST be ignored by all nodes during computation except for the purpose of determining transitive propagation and building the corresponding routing table.

A prefix can carry the _directly_attached_ attribute to indicate that the prefix is directly attached, i.e., should be routed to even if the node is in overload. In case of a negatively distributed prefix this attribute MUST NOT be included by the originator and it MUST be ignored by all nodes during SPF computation. If a prefix is locally originated the attribute _from_link_ can indicate the interface to which the address belongs to. In case of a negatively distributed prefix this attribute MUST NOT be included by the originator and it MUST be ignored by all nodes during computation. A prefix can also carry the _loopback_ attribute to indicate the said property.

Prefixes are carried in different types of TIEs indicating their type. For same prefix being included in different TIE types tie-breaking is performed according to Section 6.8.1. If the same prefix is included multiple times in multiple TIEs of the same type originating at the same node the resulting behavior is unspecified.

6.4.1. Northbound Reachability SPF

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

Once progressing, the computation uses the next higher level's Node South TIEs to find corresponding adjacencies to verify backlink connectivity. Two unidirectional links MUST be associated together to confirm bidirectional connectivity, a process often known as 'backlink check'. As part of the check, both Node TIEs MUST contain the correct System IDs *and* expected levels.

The default route found when crossing an E-W link SHOULD be used if and only if

1. the node itself does *not* have any northbound adjacencies *and*

2. the adjacent node has one or more northbound adjacencies

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

Other south prefixes found when crossing E-W link MAY be used if and only if

1. no north neighbors are advertising same or a supersuming non-default prefix *and*

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

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

A detailed example can be found in Section 7.4.

6.4.2. Southbound Reachability SPF

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

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

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

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

6.4.4. East-West Links Within ToF Level

E-W ToF links behave in terms of flooding scopes defined in Section 6.3.4 like northbound links and MUST be used exclusively for control plane information flooding. Even though a ToF node could be tempted to use those links during southbound SPF and carry traffic over them this MUST NOT be attempted since it may, in anycast cases, lead to routing loops. An implementation MAY try to resolve the looping problem by following on the ring strictly tie-broken shortest-paths only but the details are outside this specification. And even then, the problem of proper capacity provisioning of such links when they become traffic-bearing in case of failures is vexing and when used for forwarding purposes, they defeat statistical non-blocking guarantees that Clos is providing normally.

6.5. Automatic Disaggregation on Link & Node Failures

6.5.1. Positive, Non-transitive Disaggregation

Under normal circumstances, a node's South TIEs contain just the adjacencies and a default route. However, if a node detects that its default IP prefix covers one or more prefixes that are reachable through it but not through one or more other nodes at the same level, then it MUST explicitly advertise those prefixes in a South TIE. Otherwise, some percentage of the northbound traffic for those prefixes would be sent to nodes without corresponding reachability,

causing it to be dropped. Even when traffic is not being dropped, the resulting forwarding could 'backhaul' packets through the higher level spines, clearly an undesirable condition affecting the blocking probabilities of the fabric.

This specification refers to the process of advertising additional prefixes southbound as 'positive disaggregation'. Such disaggregation is non-transitive, i.e., its' effects are always constrained to a single level of the fabric. Naturally, multiple node or link failures can lead to several independent instances of positive disaggregation necessary to prevent looping or bow-tying the fabric.

A node determines the set of prefixes needing disaggregation using the following steps:

1. A DAG computation in the southern direction is performed first. The North TIEs are used to find all of prefixes it can reach and the set of next-hops in the lower level for each of them. Such a computation can be easily performed on a Fat Tree by 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, its set of next-hops is defined to be |H(r).

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

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

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

To summarize the above in simplest terms: if a node detects that its default route encompasses prefixes for which one of the other nodes in its level has no possible next-hops in the level below, it has to disaggregate it to prevent traffic loss or suboptimal routing through such nodes. Hence a node X needs to determine if it can reach a different set of south neighbors than other nodes at the same level,

which are connected to it via at least one common south neighbor. If it can, then prefix disaggregation may be required. If it can't, then no prefix disaggregation is needed. An example of disaggregation is provided in Section 7.3.

Finally, a possible algorithm is described here:

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

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

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

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

A node X computes reachability to all nodes below it based upon the received North TIEs first. This results in a set of routes, each categorized by (prefix, path_distance, next-hop set). Alternately, for clarity in the following procedure, these can be organized by next-hop set as ((next-hops), {(prefix, path_distance)}). If partial_neighbors isn't empty, then the procedure in Figure 17 describes how to identify prefixes to disaggregate.

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

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

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

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

Figure 17: Computation of Disaggregated Prefixes

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

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

1. all neighbor relationships MUST perform backlink checks.

2. overload flag as introduced in Section 6.8.2 and carried in the _overload_ schema element have to be respected during the computation. Nodes advertising themselves as overloaded MUST NOT be transited in reachability computation but MUST be used as terminal nodes with prefixes they advertise being reachable.

3. all the lower-level nodes are flooded the same disaggregated prefixes since RIFT does not build a South TIE per node which would complicate things unnecessarily. The lower-level node that can compute a southbound route to the prefix will prefer it to the disaggregated route anyway based on route preference rules.

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

5. disaggregated Prefix South TIEs are not "reflected" by the lower level. Nodes within same level do *not* need to be aware which node computed the need for disaggregation.

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

In case positive disaggregation is triggered and due to the very stable but un-synchronized nature of the algorithm the nodes may issue the necessary disaggregated prefixes at different points in time. This can lead for a short time to an "incast" behavior where the first advertising router based on the nature of longest prefix match will attract all the traffic. Different implementation strategies can be used to lessen that effect, but those are outside the scope of this specification.

It is worth observing that, in a single plane ToF, this disaggregation prevents traffic loss up to (K_LEAF * P) link failures in terms of Section 5.2 or, in other terms, it takes at minimum that many link failures to partition the ToF into multiple planes.

6.5.2. Negative, Transitive Disaggregation for Fallen Leaves

As explained in Section 5.3 failures in multi-plane ToF or more than (K_LEAF * P) links failing in single plane design can generate fallen leaves. Such scenario cannot be addressed by positive disaggregation only and needs a further mechanism.

6.5.2.1. Cabling of Multiple ToF Planes

Returning in this section to designs with multiple planes as shown originally in Figure 3, Figure 18 highlights how the ToF is cabled in case of two planes by the means of dual-rings to distribute all the North TIEs within both planes.

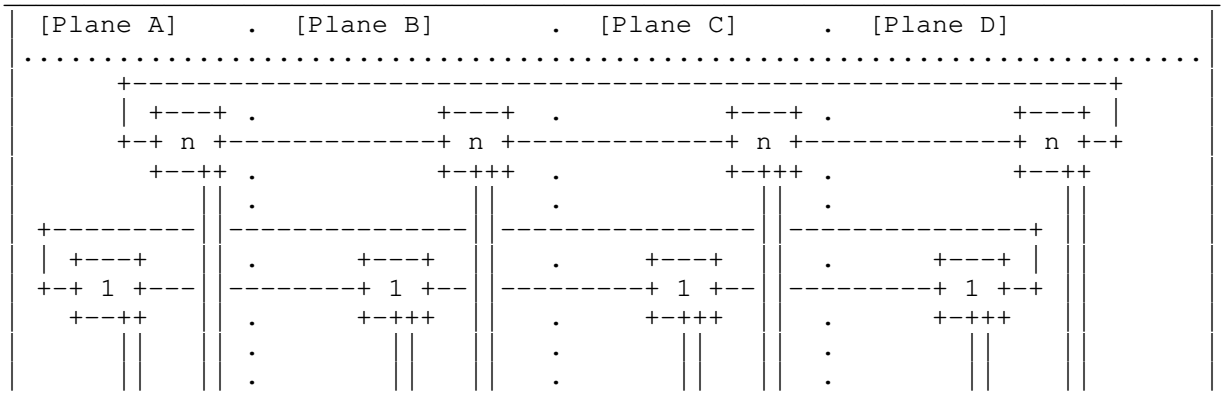


Figure 18: Topologically Connected Planes

Section 5.3 already describes how failures in multi-plane fabrics can lead to traffic loss that normal positive disaggregation cannot fix. The mechanism of negative, transitive disaggregation incorporated in RIFT provides the corresponding solution and next section explains the involved mechanisms in more detail.

6.5.2.2. Transitive Advertisement of Negative Disaggregates

A ToF node discovering that it cannot reach a fallen leaf SHOULD disaggregate all the prefixes of that leaf. It uses for that purpose negative prefix South TIEs that are, as usual, flooded southwards with the scope defined in Section 6.3.4.

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

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

6.5.2.3. Computation of Negative Disaggregates

Negative prefixes can in fact be advertised due to two different triggers. This will be described consecutively.

The first origination reason is a computation that uses all the node North TIEs to build the set of all reachable nodes by reachability computation over the complete graph and including horizontal ToF links. The computation uses the node itself as root. This is compared with the result of the normal southbound SPF as described in Section 6.4.2. The difference are the fallen leaves and all their attached prefixes are advertised as negative prefixes southbound if the node does not consider the prefix to be reachable within the southbound SPF.

The second origination reason hinges on the understanding how the negative prefixes are used within the computation as described in Figure 19. When attaching the negative prefixes at a certain point in time the negative prefix may find itself with all the viable nodes from the shorter match nexthop being pruned. In other words, all its northbound neighbors provided a negative prefix advertisement. This is the trigger to advertise this negative prefix transitively south and is normally caused by the node being in a plane where the prefix belongs to a fabric leaf that has "fallen" in this plane. Obviously, when one of the northbound switches withdraws its negative advertisement, the node has to withdraw its transitively provided negative prefix as well.

6.6. Attaching Prefixes

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

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

Each prefix has its cost adjusted before being added into the RIFT route database. The cost of the prefix is set to the cost received plus the cost of the minimum distance next-hop to that neighbor while considering its attributes such as mobility per Section 6.8.4. Then each prefix can be added into the RIFT route database with the next-hop set; ties are broken based upon type first and then distance and further on _PrefixAttributes_. Only the best combination is used for forwarding. RIFT route preferences are normalized by the enum _RouteType_ in Thrift [thrift] model given in Appendix B.

An example implementation for node X follows:

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

Figure 19: Adding Routes from South TIE Positive and Negative Prefixes

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

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

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

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

To make the negative disaggregation less abstract and provide an example ToP node T1 with 4 ToF parents S1..S4 as represented in Figure 20 are considered further:

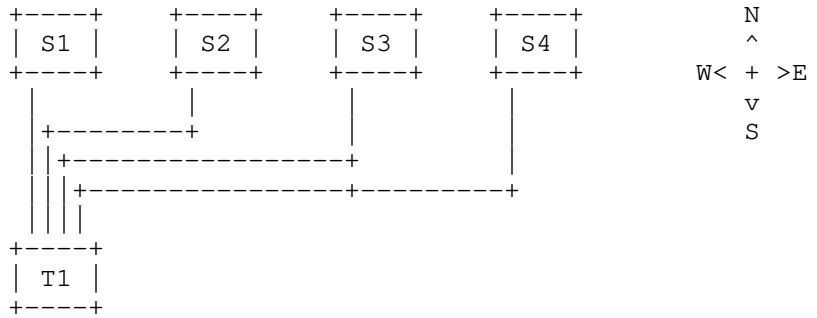


Figure 20: A ToP Node with 4 Parents

If all ToF nodes can reach all the prefixes in the network; with RIFT, they will normally advertise a default route south. An abstract Routing Information Base (RIB), more commonly known as a routing table, stores all types of maintained routes including the negative ones and "tie-breaks" for the best one, whereas an abstract Forwarding table (FIB) retains only the ultimately computed "positive" routing instructions. In T1, those tables would look as illustrated in Figure 21:

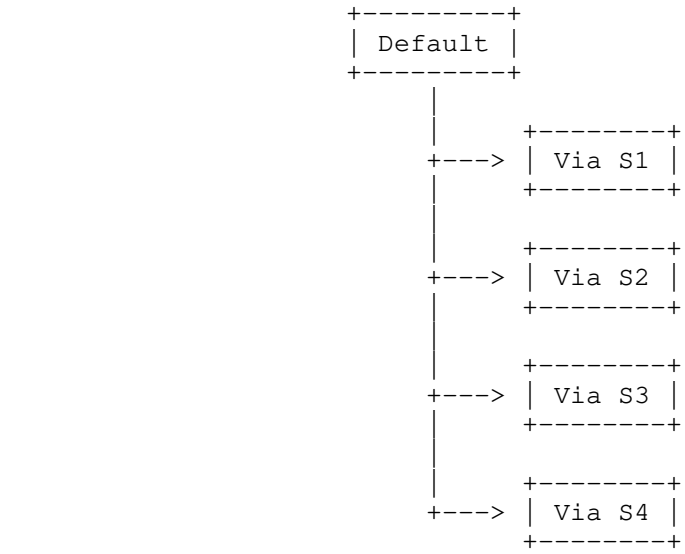


Figure 21: Abstract RIB

In case T1 receives a negative advertisement for prefix 2001:db8::/32 from S1 a negative route is stored in the RIB (indicated by a ~ sign), while the more specific routes to the complementing ToF nodes are installed in FIB. RIB and FIB in T1 now look as illustrated in Figure 22 and Figure 23, respectively:

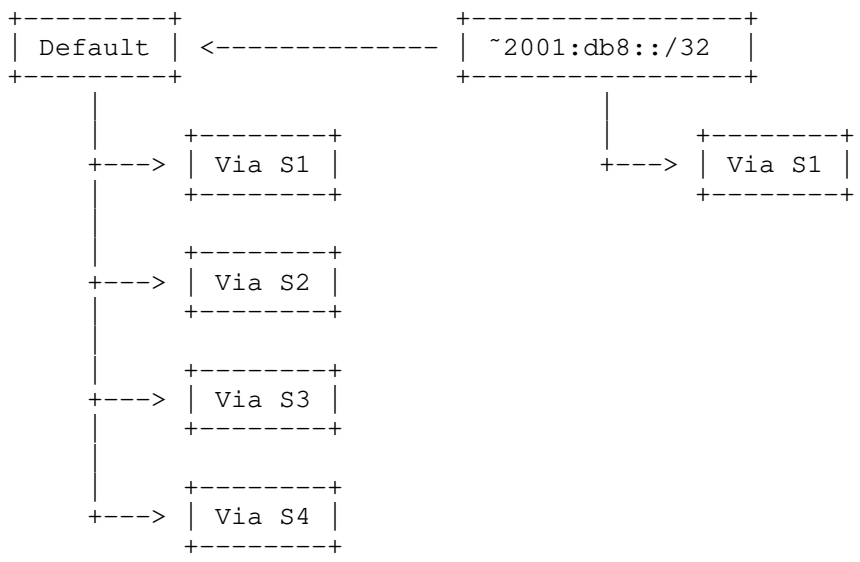


Figure 22: Abstract RIB after Negative 2001:db8::/32 from S1

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

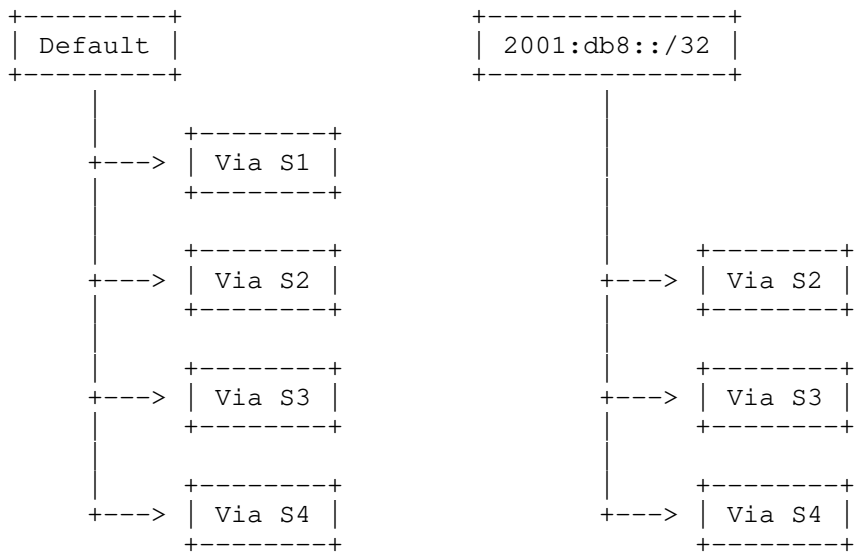


Figure 23: Abstract FIB after Negative 2001:db8::/32 from S1

To illustrate matters further consider T1 receiving a negative advertisement for prefix 2001:db8:1::/48 from S2, which is stored in RIB again. After the update, the RIB in T1 is illustrated in Figure 24:

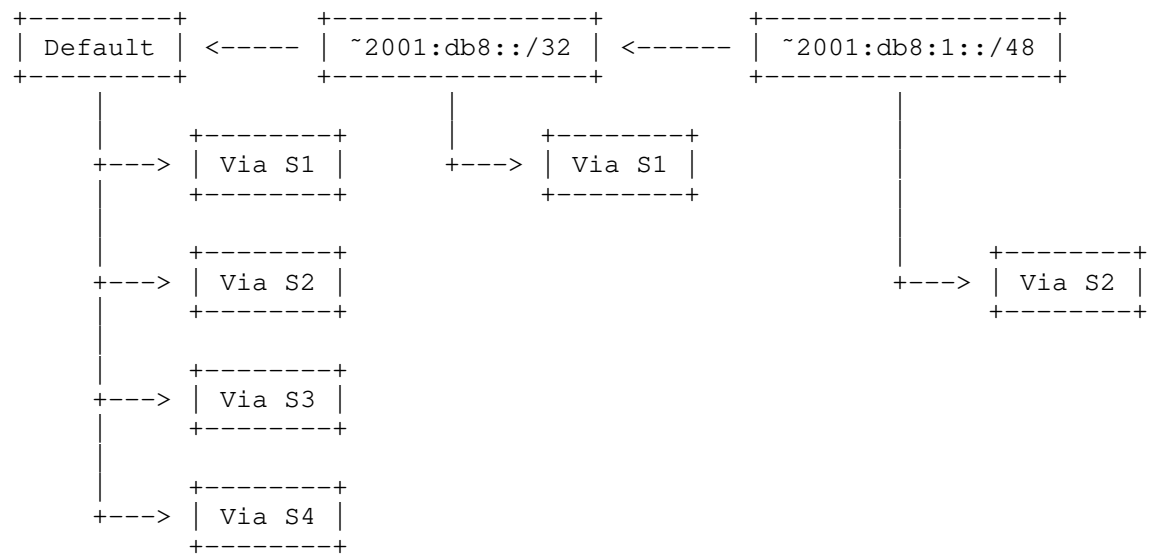


Figure 24: Abstract RIB after Negative 2001:db8:1::/48 from S2

Negative 2001:db8:1::/48 inherits from 2001:db8::/32 now, so the positive more specific routes are the complements to S2 in the set of next hops for 2001:db8::/32, which are S3 and S4, or, in other words, all entries of the parent with the negative holes "punched in" again. After the update, the FIB in T1 shows as illustrated in Figure 25:

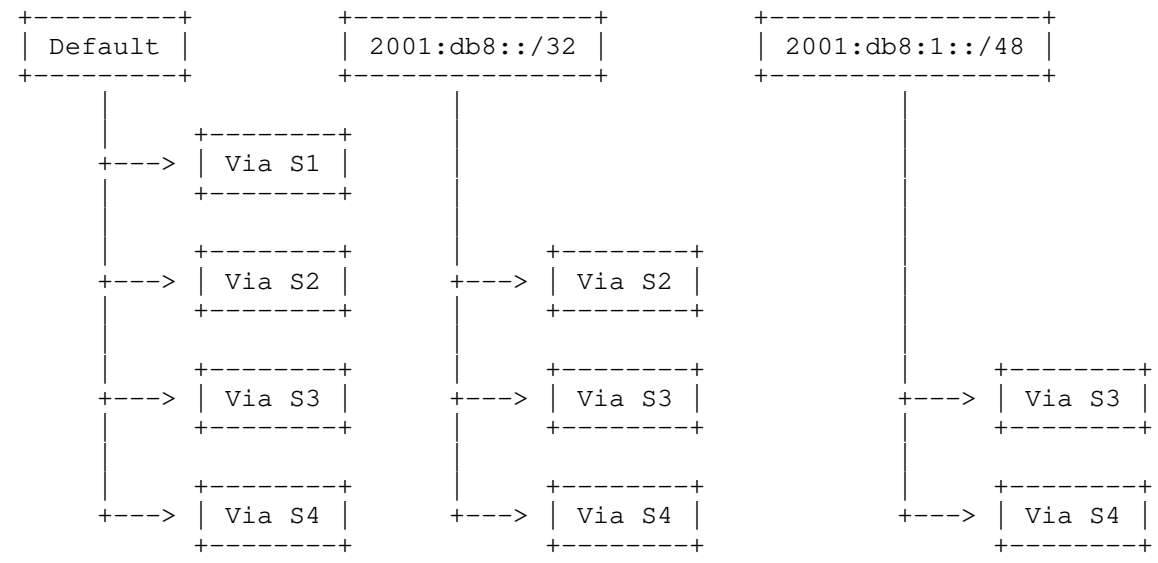


Figure 25: Abstract FIB after Negative 2001:db8:1::/48 from S2

Further, assume that S3 stops advertising its service as default gateway. The entry is removed from RIB as usual. In order to update the FIB, it is necessary to eliminate the FIB entry for the default route, as well as all the FIB entries that were created for negative routes pointing to the RIB entry being removed (::/0). This is done recursively for 2001:db8::/32 and then for, 2001:db8:1::/48. The related FIB entries via S3 are removed, as illustrated in Figure 26.

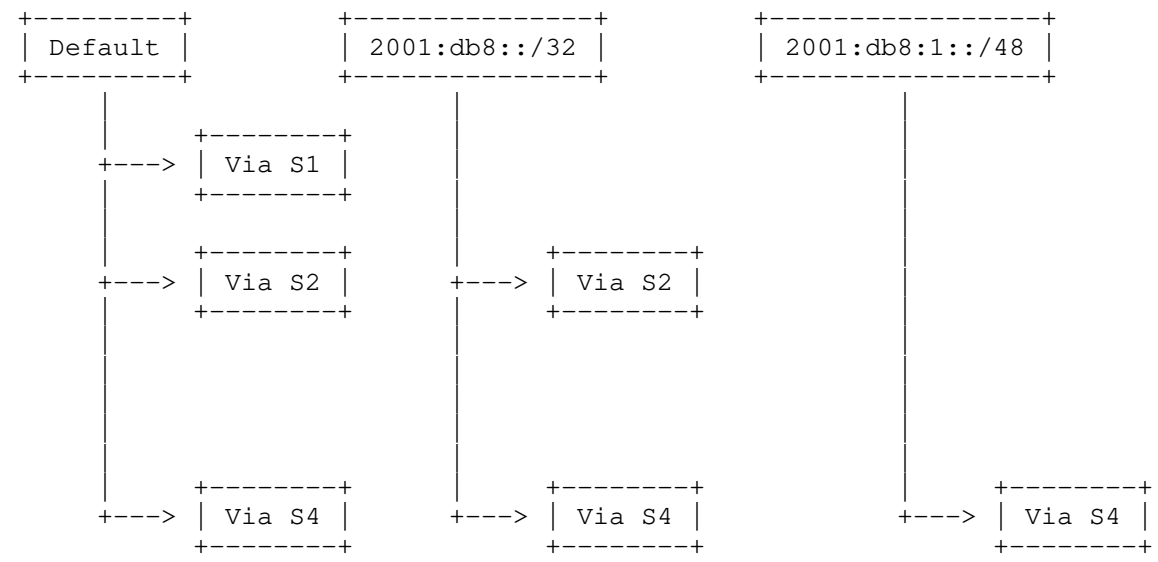


Figure 26: Abstract FIB after Loss of S3

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

Finally, the case occurs where S3 becomes available again as a default gateway, and a negative advertisement is received from S4 about prefix 2001:db8:2::/48 as opposed to 2001:db8:1::/48. Again, a negative route is stored in the RIB, and the more specific route to the complementing ToF nodes are installed in FIB. Since 2001:db8:2::/48 inherits from 2001:db8::/32, the positive FIB routes are chosen by removing S4 from S2, S3, S4. The abstract FIB in T1 now shows as illustrated in Figure 27:

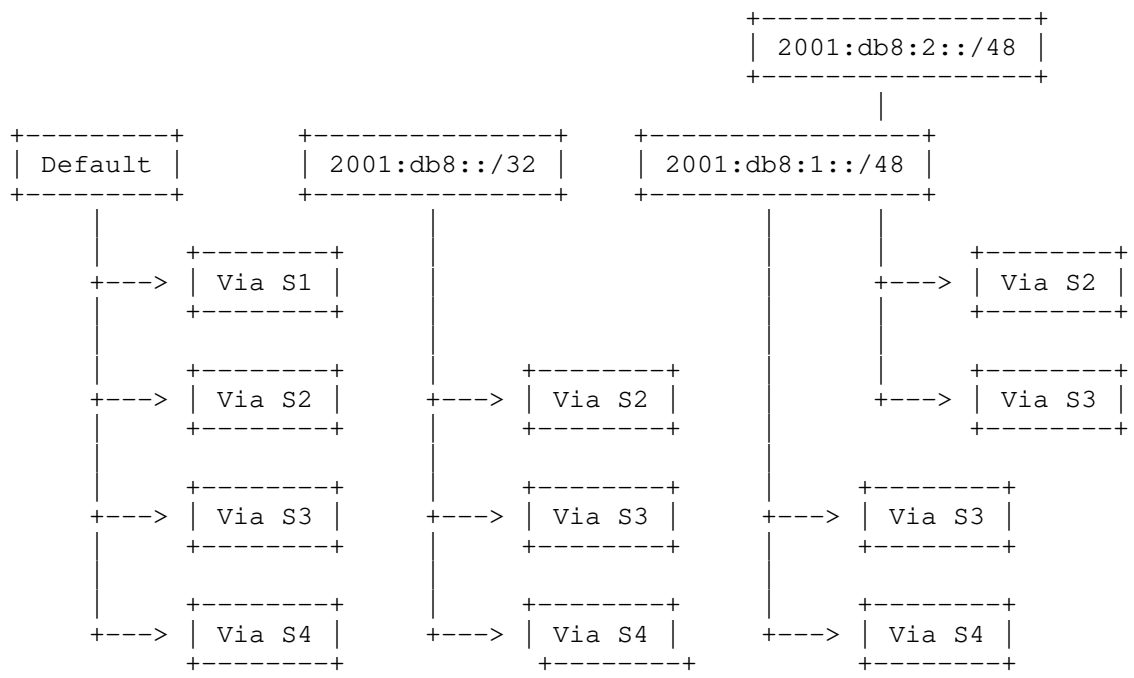


Figure 27: Abstract FIB after Negative 2001:db8:2::/48 from S4

6.7. Optional Zero Touch Provisioning (ZTP)

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

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

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

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

6.7.1. Terminology

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

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

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

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

TOP_OF_FABRIC:
: A configuration flag that MUST be provided on all ToF nodes. LEAF_FLAG and CONFIGURED_LEVEL cannot be defined at the same time as this flag. It implies a CONFIGURED_LEVEL value. In fact, it is basically a shortcut for configuring same level at all ToF nodes which is unavoidable since an initial 'seed' is needed for other ZTP nodes to derive their level in the topology. The flag plays an important role in fabrics with multiple planes to enable successful negative disaggregation (Section 6.5.2). It is carried in the _top_of_fabric_ schema element. A standards conform RIFT implementation implies a CONFIGURED_LEVEL value of _top_of_fabric_level_ in case of TOP_OF_FABRIC. This value is kept reasonably low to allow for fast ZTP re-convergence on failures.

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 in the sense of deriving its own level based on other nodes' information. TOP_OF_FABRIC flag is ignored when this value is defined. LEAF_ONLY can be set only if this value is undefined or set to _leaf_level_.

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 6.8.9. It is a capability that implies LEAF_ONLY and the corresponding restrictions. TOP_OF_FABRIC flag is ignored when set at the same time as this flag. It is carried in the _leaf_only_and_leaf_2_leaf_procedures_ schema flag.

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

Valid Offered Level (VOL):
A neighbor's level received in 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 _leaf_level_ do not constitute VOLs (since no valid DERIVED_LEVEL can be obtained from those and consequently _not_a_ztp_offer_ flag MUST be ignored). Offers from LIEs with _not_a_ztp_offer_ being true are not VOLs either. If a node maintains parallel adjacencies to the neighbor, VOL on each adjacency is considered as equivalent, i.e. the newest VOL from any such adjacency updates the VOL received from the same node.

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

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

Highest Adjacency ThreeWay (HAT):
Highest neighbor level of all the formed _ThreeWay_ adjacencies for the node.

6.7.2. Automatic System ID Selection

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

As matter of operational concern, the router MUST ensure that such identifier is not changing very frequently (or at least not without sending all its TIEs with fairly short lifetimes, i.e. purging them) since otherwise the network may be left with large amounts of stale TIEs in other nodes (though this is not necessarily a serious problem if the procedures described in Section 9 are implemented).

6.7.3. Generic Fabric Example

ZTP forces considerations of an incorrectly or unusually cabled fabric and how such a topology can be forced into a "lattice" structure which a fabric represents (with further restrictions). A necessary and sufficient physical cabling is shown in Figure 28. The assumption here is that all nodes are in the same PoD.

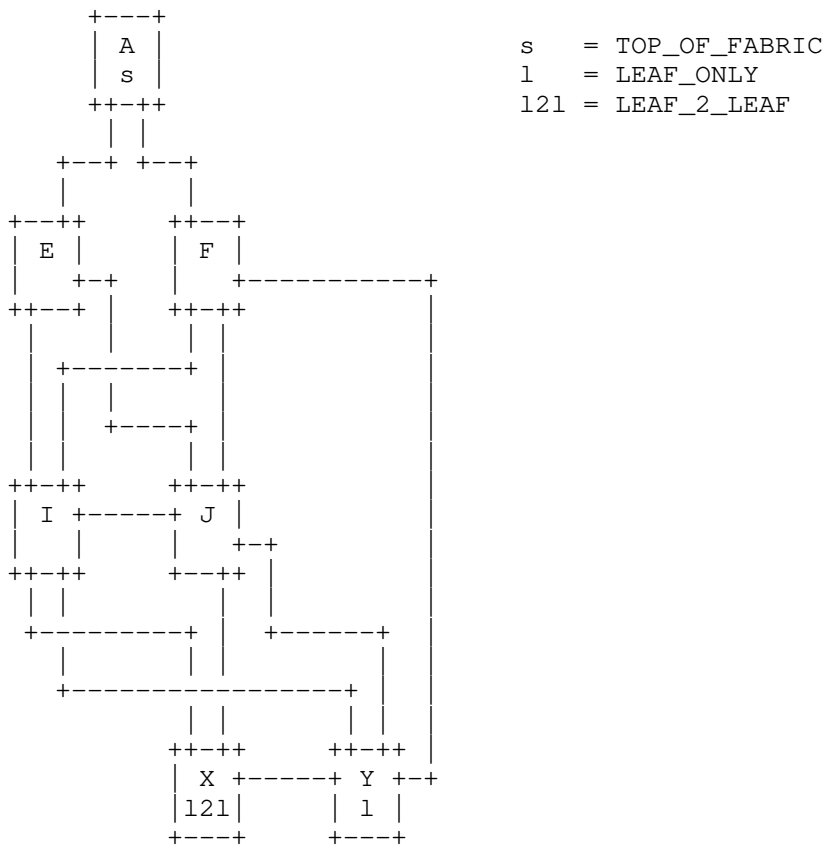


Figure 28: Generic ZTP Cabling Considerations

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

6.7.4. Level Determination Procedure

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

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

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

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

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

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

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

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

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

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

1. It computes HAT per procedures above but does *not* use it to compute DERIVED_LEVEL. HAT is used to limit adjacency formation per Section 6.2.

It MAY also follow modified procedures:

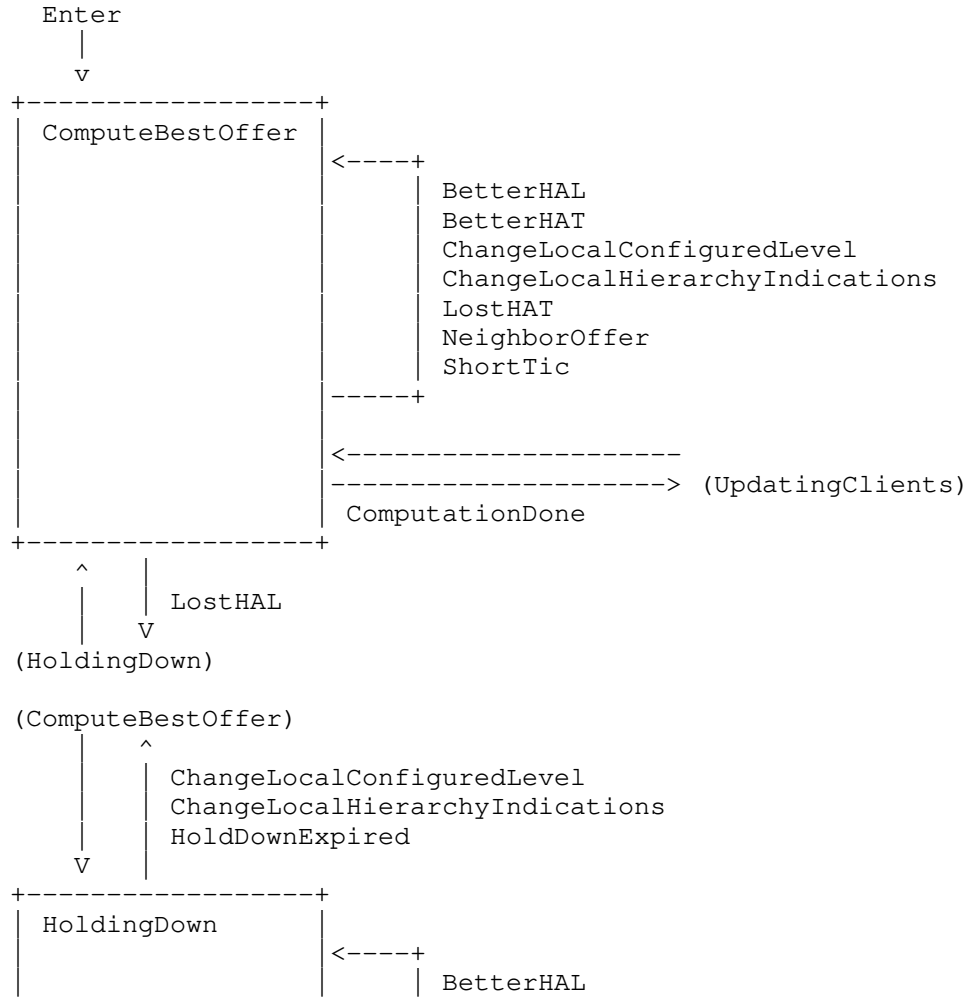
1. It may pick a different strategy to choose VOL, e.g. use the VOL value with highest number of VOLs. Such strategies are only possible since the node always remains "at the bottom of the

fabric" while another layer could "invert" the fabric by picking its preferred VOL in a different fashion than always trying to achieve the highest viable level.

6.7.5. ZTP FSM

This section specifies the precise, normative ZTP FSM and can be omitted unless the reader is pursuing an implementation of the protocol. For additional clarity a graphical representation of the ZTP FSM is depicted in Figure 29. It may also be helpful to refer to the normative schema in Appendix B.

Initial state is ComputeBestOffer.



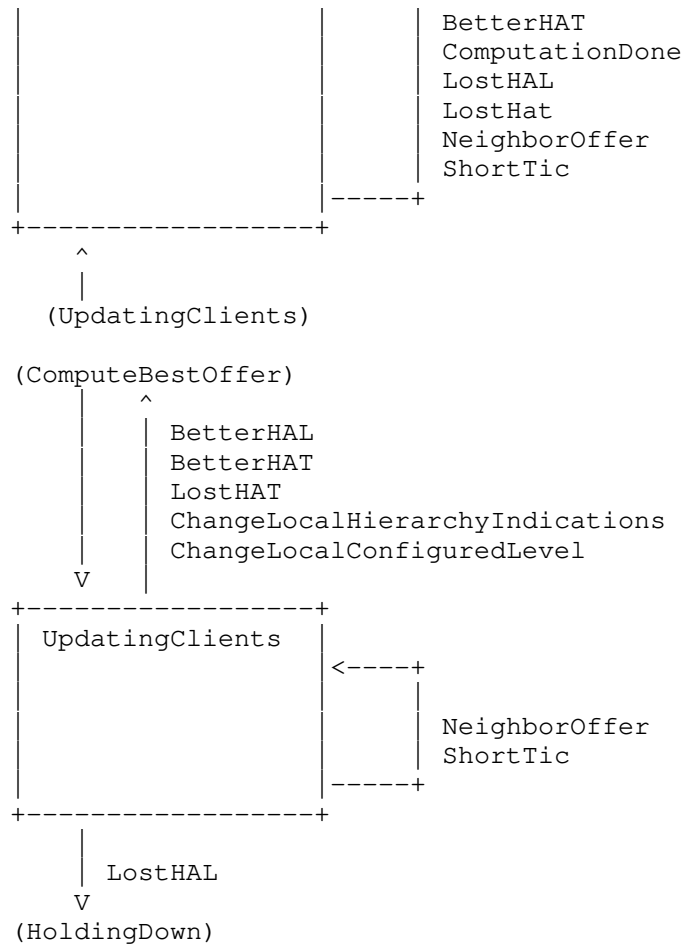


Figure 29: ZTP FSM

The following words are used for well-known procedures:

* PUSH Event: queues an event to be executed by the FSM upon exit of this action

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

* UPDATE_OFFER: store current offer with adjacency holdtime as lifetime and COMPARE_OFFERS, then PUSH corresponding events

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

* REMOVE_OFFER: remove the corresponding offer and COMPARE_OFFERS, PUSH corresponding events

* PURGE_OFFERS: REMOVE_OFFER for all held offers, COMPARE OFFERS, PUSH corresponding events

* PROCESS_OFFER:

    1. if no level offered then REMOVE_OFFER

    2. else

        1. if offered level > leaf then UPDATE_OFFER

        2. else REMOVE_OFFER

States:

* ComputeBestOffer: processes received offers to derive ZTP variables

* HoldingDown: holding down while receiving updates

* UpdatingClients: updates other FSMs on the same node with computation results

Events:

* ChangeLocalHierarchyIndications: node locally configured with new leaf flags.

* ChangeLocalConfiguredLevel: node locally configured with a defined level

* NeighborOffer: a new neighbor offer with optional level and neighbor state.

* BetterHAL: better HAL computed internally.

* BetterHAT: better HAT computed internally.

* LostHAL: lost last HAL in computation.

* LostHAT: lost HAT in computation.

* ComputationDone: computation performed.

* HoldDownExpired: holddown timer expired.

* ShortTic: one second timer tick. This event is provided to the FSM once a second by an implementation-specific mechanism that is outside the scope of this specification. This event is quietly ignored if the relevant transition does not exist.

Actions:

* on ChangeLocalConfiguredLevel in HoldingDown finishes in ComputeBestOffer: store configured level

* on BetterHAT in HoldingDown finishes in HoldingDown: no action

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

* on NeighborOffer in HoldingDown finishes in HoldingDown: PROCESS_OFFER

* on ComputationDone in HoldingDown finishes in HoldingDown: no action

* on BetterHAL in HoldingDown finishes in HoldingDown: no action

* on LostHAT in HoldingDown finishes in HoldingDown: no action

* on LostHAL in HoldingDown finishes in HoldingDown: no action

* on HoldDownExpired in HoldingDown finishes in ComputeBestOffer: PURGE_OFFERS

* on ChangeLocalHierarchyIndications in HoldingDown finishes in ComputeBestOffer: store leaf flags

* on LostHAT in ComputeBestOffer finishes in ComputeBestOffer: LEVEL_COMPUTE

* on NeighborOffer in ComputeBestOffer finishes in ComputeBestOffer: PROCESS_OFFER

* on BetterHAT in ComputeBestOffer finishes in ComputeBestOffer: LEVEL_COMPUTE

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

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

* on ShortTic in ComputeBestOffer finishes in ComputeBestOffer: remove expired offers

* on ComputationDone in ComputeBestOffer finishes in UpdatingClients: no action

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

* on BetterHAL in ComputeBestOffer finishes in ComputeBestOffer: LEVEL_COMPUTE

* on ShortTic in UpdatingClients finishes in UpdatingClients: remove expired offers

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

* on BetterHAT in UpdatingClients finishes in ComputeBestOffer: no action

* on BetterHAL in UpdatingClients finishes in ComputeBestOffer: no action

* on ChangeLocalConfiguredLevel in UpdatingClients finishes in ComputeBestOffer: store configured level

* on ChangeLocalHierarchyIndications in UpdatingClients finishes in ComputeBestOffer: store leaf flags

* on NeighborOffer in UpdatingClients finishes in UpdatingClients: PROCESS_OFFER

* on LostHAT in UpdatingClients finishes in ComputeBestOffer: no action

* on Entry into ComputeBestOffer: LEVEL_COMPUTE

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

6.7.6. Resulting Topologies

The procedures defined in Section 6.7.4 will lead to the RIFT topology and levels depicted in Figure 30.

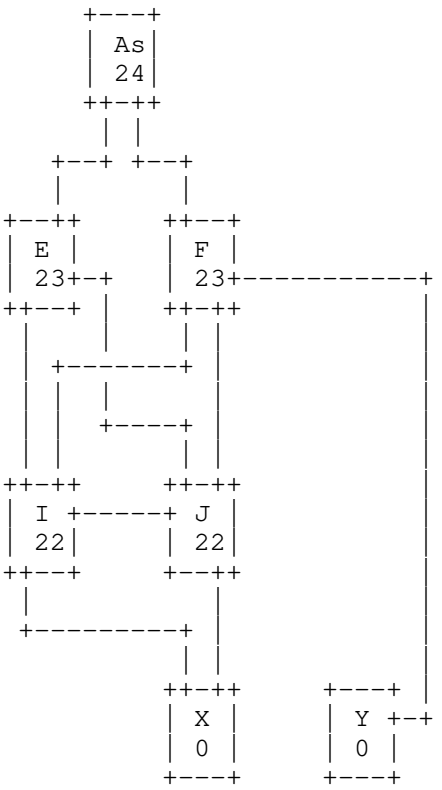


Figure 30: Generic ZTP Topology Autoconfigured

In case where the LEAF_ONLY restriction on Y is removed the outcome would be very different however and result in Figure 31. This demonstrates basically that auto configuration makes miscabling detection hard and with that can lead to undesirable effects in cases where leaves are not "nailed" by the appropriately configured flags and arbitrarily cabled.

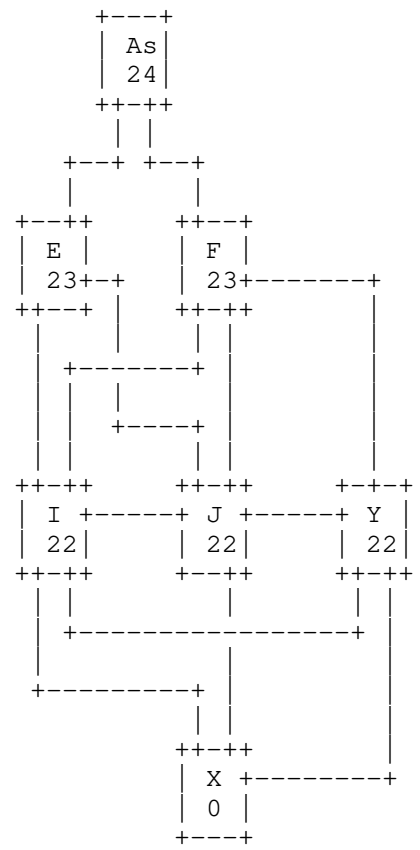


Figure 31: Generic ZTP Topology Autoconfigured

6.8. Further Mechanisms

6.8.1. Route Preferences

Since RIFT distinguishes between different route types such as e.g. external routes from other protocols and additionally advertises special types of routes on disaggregation, the protocol MUST tie-break internally different types on a clear preference scale to prevent traffic loss or loops. The preferences are given in the schema type _RouteType_.

Table Table 5 contains the route type as derived from the TIE type carrying it. Entries are sorted from the most preferred route type to the least preferred route type.

TIE Type	Resulting Route Type
None	Discard
Local Interface	LocalPrefix
S-PGP	South PGP
N-PGP	North PGP
North Prefix	NorthPrefix
North External Prefix	NorthExternalPrefix
South Prefix and South Positive Disaggregation	SouthPrefix
South External Prefix and South Positive External Disaggregation	SouthExternalPrefix
South Negative Prefix	NegativeSouthPrefix

Table 5: TIEs and Contained Route Types

6.8.2. Overload Bit

Overload attribute is specified in the packet encoding schema (Appendix B) in the _overload_ flag.

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

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

6.8.3. Optimized Route Computation on Leaves

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

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

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

Leaf nodes would then attach necessary prefixes as described in Section 6.6.

6.8.4. Mobility

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

There are two classic approaches to explicitly maintain this information, "timestamp" and "sequence counter" as follows:

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

sequence counter:
: With this method, a mobile device notifies its point of attachment on arrival with a sequence counter that is incremented upon each movement. On the positive side, this method does not have a dependency on a precise sense of time, since the sequence of

movements is kept in order by the mobile device. The disadvantage of this approach is the need for support for protocols that may be used by the mobile device to register its presence to the leaf node with the capability to provide a sequence counter. Well-known issues with sequence counters such as wrapping and comparison rules MUST be addressed properly. Sequence numbers MUST be compared by a single homogenous source to make operation feasible. Sequence number comparison from multiple heterogeneous sources would be extremely difficult to implement.

RIFT supports a hybrid approach by using an optional 'PrefixSequenceType' attribute (that is also called a _monotonic_clock_ in the schema) that consists of a timestamp and optional sequence number field. In case of a negatively distributed prefix this attribute MUST NOT be included by the originator and it MUST be ignored by all nodes during computation. When this attribute is present (observe that per data schema the attribute itself is optional but in case it is included the 'timestamp' field is required):

* The leaf node MAY advertise a timestamp of the latest sighting of a prefix, e.g., by snooping IP protocols or the node using the time at which it advertised the prefix. RIFT transports the timestamp within the desired prefix North TIEs as [IEEEstd1588] timestamp.

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

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

* Any prefix present on the fabric in multiple nodes that have the *same* clock is considered as anycast.

* RIFT specification assumes that all nodes are being synchronized within at least 200 milliseconds or less. This is achievable through the use of NTP [RFC5905]. An implementation MAY provide a way to reconfigure a domain to a different value, and provides for this purpose a variable called MAXIMUM_CLOCK_DELTA.

6.8.4.1. Clock Comparison

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

1. The ASNC is older than any other value except ASNC *and*
2. Clocks with timestamp differing by more than MAXIMUM_CLOCK_DELTA are comparable by using the timestamps only *and*
3. Clocks with timestamps differing by less than MAXIMUM_CLOCK_DELTA are comparable by using their TIDs only *and*
4. An undefined TID is always older than any other TID *and*
5. TIDs are compared using rules of [RFC8505].

6.8.4.2. Interaction between Time Stamps and Sequence Counters

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

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

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

6.8.4.3. Anycast vs. Unicast

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

If a monotonic clock attribute is provided on the prefix, then the prefix with the *newest* clock value is strictly preferred. An anycast prefix does not carry a clock or all clock attributes MUST be the same under the rules of Section 6.8.4.1.

It is important that in mobility events the leaf is re-flooding as quickly as possible to communicate the absence of the prefix that moved.

Without support for [RFC8505] movements on the fabric within intervals smaller than 100msec will be interpreted as anycast.

6.8.4.4. Overlays and Signaling

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

In the context of mobility, overlays provide another possible solution to avoid injecting mobile prefixes into the fabric as well as improving scalability of the deployment. It makes sense to consider overlays for mobility solutions in IP fabrics. As an example, a mobility protocol such as LISP [RFC9300] [RFC9301] may inform the ingress leaf of the location of the egress leaf in real time.

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

6.8.5. Key/Value (KV) Store

6.8.5.1. Southbound

RIFT supports the southbound distribution of key-value pairs that can be used to distribute information to facilitate higher levels of functionality (e.g. distribution of configuration information). KV South TIEs may arrive from multiple nodes and therefore MUST execute the following tie-breaking rules for each key:

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

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

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

6.8.5.2. Northbound

Certain use cases necessitate distribution of essential KV information that is generated by the leaves in the northbound direction. Such information is flooded in KV North TIEs. Since the originator of the KV North TIEs is preserved during flooding, the corresponding mechanism will define, if necessary, tie-breaking rules depending on the semantics of the information.

Only KV TIEs from nodes that are reachable via multiplane reachability computation mentioned in Section 6.5.2.3 SHOULD be considered.

6.8.6. Interactions with BFD

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

After RIFT _ThreeWay_ hello adjacency convergence a BFD session MAY be formed automatically between the RIFT endpoints without further configuration using the exchanged discriminators that are equal to the _local_id_ in the _LIEPacket_. The capability of the remote side to support BFD is carried in the LIEs in _LinkCapabilities_.

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

In case of parallel links between nodes each link MAY run its own independent BFD session or they MAY share a session. The specific manner in which this is implemented is outside the scope of this document.

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

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

The BFD TTL follows [RFC5082].

6.8.7. Fabric Bandwidth Balancing

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

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

6.8.7.1. Northbound Direction

Every RIFT node SHOULD compute the amount of northbound bandwidth available through neighbors at a higher level and modify the distance received on default route from these neighbors. The bandwidth is advertised in _NodeNeighborsTIEElement_ element which represents the sum of the bandwidths of all the parallel links to a neighbor. Default routes with differing distances SHOULD be used to support weighted ECMP forwarding. Such a distance is called Bandwidth Adjusted Distance (BAD). This is best illustrated by a simple example.

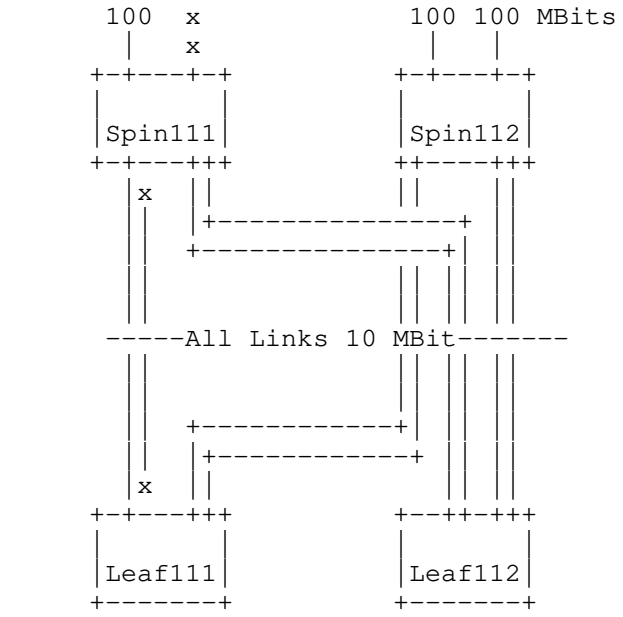


Figure 32: Balancing Bandwidth

Figure 32 depicts an example topology where links between leaf and spine nodes are 10 MBit/s and links from spine nodes northbound are 100 MBit/s. It includes parallel link failure between Leaf 111 and Spine 111 and as a result, Leaf 111 wants to forward more traffic toward Spine 112. Additionally, it includes as well an uplink failure on Spine 111.

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

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

L_N_u: sum of the bandwidth available from L to N (to account for parallel links)

N_u: sum of the uplink bandwidth available on N

T_N_u: L_N_u * OVERSUBSCRIPTION_CONSTANT + N_u

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

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

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

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

Table 6: BAD Computation

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

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

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

6.8.7.2. Southbound Direction

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

6.8.8. Label Binding

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

6.8.9. Leaf to Leaf Procedures

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

advertise the LEAF_2_LEAF flag in its node capabilities *and*

set the overload flag on all leaf's Node TIEs *and*

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

always use E-W leaf adjacency in all SPF computations *and*

install a discard route for any advertised aggregate routes in a leaf's TIE *and*

never form southbound adjacencies.

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

6.8.10. Address Family and Multi Topology Considerations

Multi-Topology (MT)[RFC5120] and Multi-Instance (MI)[RFC8202] concepts are used today in link-state routing protocols to support several domains on the same physical topology. RIFT supports this capability by carrying transport ports in the LIE protocol exchanges. Multiplexing of LIEs can be achieved by either choosing varying multicast addresses or ports on the same address.

BFD interactions in Section 6.8.6 are implementation dependent when multiple RIFT instances run on the same link.

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

Based on the rules defined in Section 6.4, Section 6.3.8 and given the presence of E-W links, RIFT can provide a one-hop protection for nodes that have lost all their northbound links. This can also be applied to multi-plane designs where complex link set failures occur at the ToF when links are exclusively used for flooding topology information. Section 7.4 outlines this behavior.

6.9. Security

6.9.1. Security Model

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

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

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

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

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

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

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

a. specification of the appropriate level in the fabric,

b. discovery and reporting of missing connections,

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

Operators may only choose to configure the level of each node, but not explicitly configure which connections are allowed. In this case, RIFT will only allow adjacencies to establish between nodes that are in adjacent levels. Operators with the lowest security requirements may not use any configuration to specify which connections are allowed. Nodes in such fabrics could rely fully on ZTP and only established adjacencies between nodes in adjacent levels. Figure 33 illustrates inherent tradeoffs between the different security models.

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

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

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

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

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

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

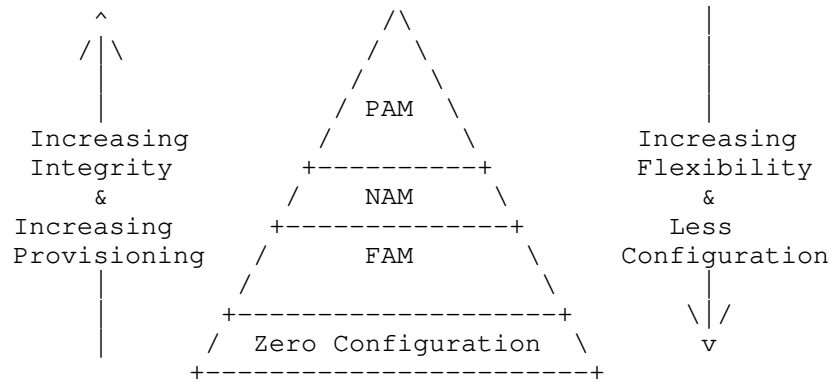


Figure 33: Security Model

6.9.2. Security Mechanisms

RIFT Security goals are to ensure:

1. authentication
2. message integrity
3. the prevention of replay attacks
4. low processing overhead
5. efficient messaging

Message confidentiality is a non-goal.

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

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

6.9.3. Security Envelope

A serialized schema _ProtocolPacket_ MUST be carried in a secure envelope illustrated in Figure 34. The _ProtocolPacket_ MUST be serialized using the default Thrift's Binary Protocol. Any value in the packet following a security fingerprint MUST be used by a receiver only after the appropriate fingerprint has been validated against the data covered by it and the advertised key. This means that for all packets, in case the node is configured to validate the outer fingerprint, an invalid fingerprint will lead to packet rejection. Further, in case of reception of a TIE, and the receiver being configured to validate the originator by checking the TIE Origin Security Envelope Header fingerprint, an invalid inner fingerprint will lead to the rejection of the packet.

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

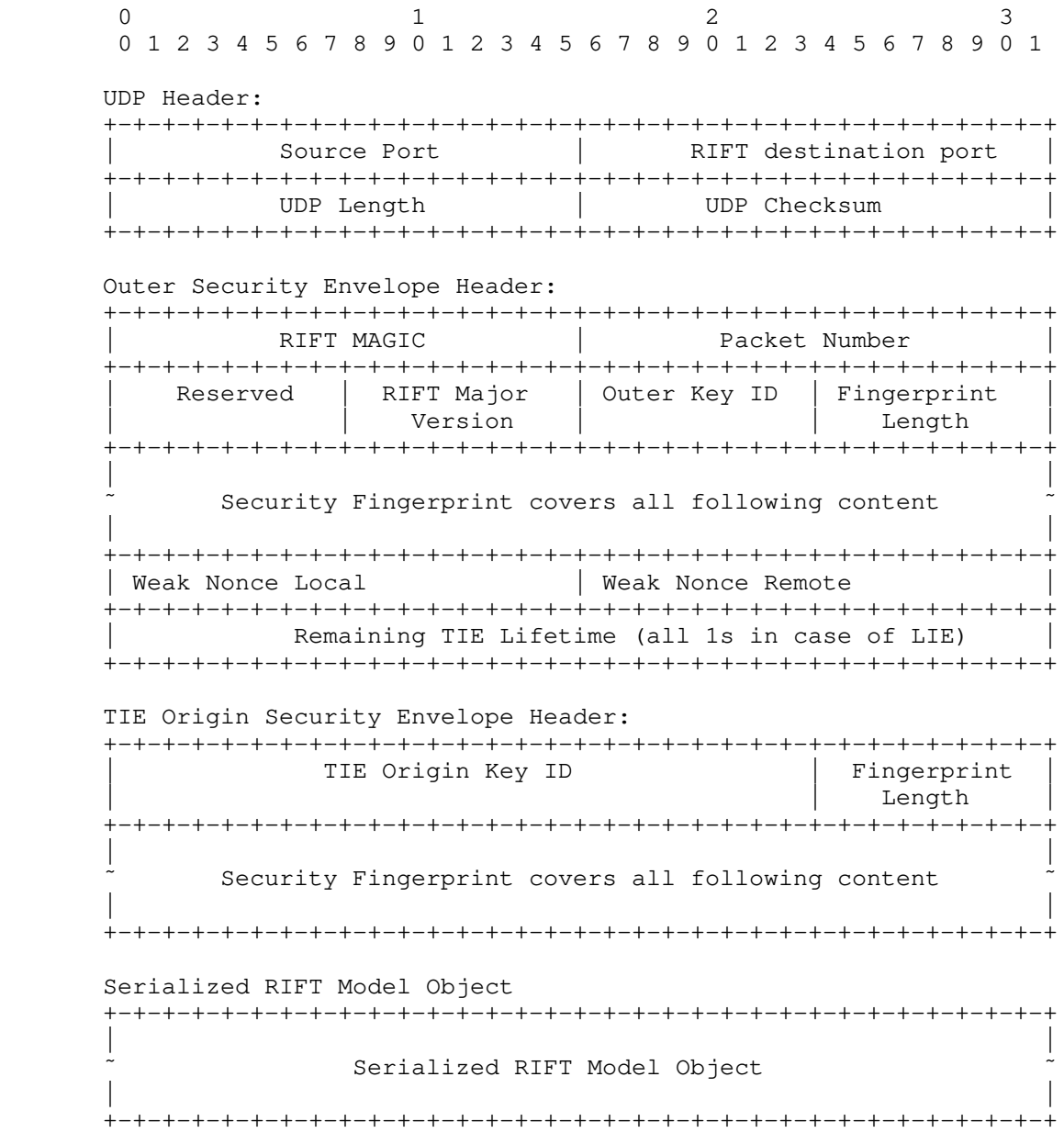


Figure 34: Security Envelope

RIFT MAGIC:
16 bits. Constant value of 0xA1F7 that allows easy classification of RIFT packets independent of the UDP port used.

Packet Number:
16 bits. An optional, per adjacency, per packet type number set using the sequence number arithmetic defined in Appendix A. If the arithmetic in Appendix A is not used the node MUST set the value to _undefined_packet_number_. This number can be used to detect losses and misordering in flooding for either operational purposes or in implementation to adjust flooding behavior to current link or buffer quality. This number MUST NOT be used to discard or validate the correctness of packets. Packet numbers are incremented on each interface and within that for each type of packet independently. This allows parallelizing packet generation and processing for different types within an implementation if so desired.

RIFT Major Version:
8 bits. It allows checking whether protocol versions are compatible, i.e., if the serialized object can be decoded at all. An implementation MUST drop packets with unexpected values and MAY report a problem. The specification of how an implementation negotiates the schema's major version is outside the scope of this document.

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

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

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

Security Fingerprint:
32 bits * Length of Fingerprint. This is a signature that is computed over all data following after it. If the significant bits of fingerprint are fewer than the 32 bits padded length then the significant bits MUST be left aligned and remaining bits on the right padded with 0s. When using PKI (Public Key

Infrastructure) the Security fingerprint originating node uses its private key to create the signature. The original packet can then be verified provided the public key is shared and current. Methodology to negotiate, distribute, or roll over keys are outside the scope of this document.

Remaining TIE Lifetime:
32 bits. In case of anything but TIEs this field MUST be set to all ones and Origin Security Envelope Header MUST NOT be present in the packet. For TIEs this field represents the remaining lifetime of the TIE and Origin Security Envelope Header MUST be present in the packet.

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

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

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

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

6.9.4. Weak Nonces

The protocol uses two 16-bit nonces to salt generated signatures. The term "nonce" is used a bit loosely since RIFT nonces are not being changed in every packet as often common in cryptography. For efficiency purposes they are changed at a high enough frequency to dwarf practical replay attack attempts. And hence, such nonces are called from this point on "weak" nonces.

Any implementation including RIFT security MUST generate and wrap around local nonces properly. When a nonce increment leads to _undefined_nonce_ value, the value MUST be incremented again immediately. All implementations MUST reflect the neighbor's nonces. An implementation SHOULD increment a chosen nonce on every LIE FSM transition that ends up in a different state from the previous one and MUST increment its nonce at least every

_nonce_regeneration_interval_ (such considerations allow for efficient implementations without opening a significant security risk). When flooding TIEs, the implementation MUST use recent (i.e. within allowed difference) nonces reflected in the LIE exchange. The schema specifies in _maximum_valid_nonce_delta_ the maximum allowable nonce value difference on a packet compared to reflected nonces in the LIEs. Any packet received with nonces deviating more than the allowed delta MUST be discarded without further computation of signatures to prevent computation load attacks. The delta is either a negative or positive difference that a mirrored nonce can deviate from local value to be considered valid. If nonces are not changed on every packet but at the maximum interval on both sides this opens statistically a _maximum_valid_nonce_delta_/2 window for identical LIEs, TIE and TI(x)E replays. The interval cannot be too small since LIE FSM may change states fairly quickly during ZTP without sending LIEs and additionally, UDP can both loose as well as misorder packets.

In cases where a secure implementation does not receive signatures or receives undefined nonces from a neighbor (indicating that it does not support or verify signatures), it is a matter of local policy as to how those packets are treated. A secure implementation MAY refuse forming an adjacency with an implementation that is not advertising signatures or valid nonces, or it MAY continue signing local packets while accepting a neighbor's packets without further security validation.

As a necessary exception, an implementation MUST advertise the remote nonce value as _undefined_nonce_ when the FSM is not in _TwoWay_ or _ThreeWay_ state and accept an _undefined_nonce_ for its local nonce value on packets in any other state than _ThreeWay_.

As an optional optimization, an implementation MAY send one LIE with previously negotiated neighbor's nonce to try to speed up a neighbor's transition from _ThreeWay_ to _OneWay_ and MUST revert to sending _undefined_nonce_ after that.

6.9.5. Lifetime

Reflooding same TIE version quickly with small variations in its lifetime may lead to an excessive number of security fingerprint computations. To avoid this, the application generating the fingerprints for flooded TIEs MAY round the value down to the next _rounddown_lifetime_interval_ on the packet header to reuse previous computation results. TIEs flooded with such rounded lifetimes only will limit the amount of computations necessary during transitions that lead to advertisement of same TIEs with same information within a short period of time.

6.9.6. Security Association Changes

There in no mechanism to convert a security envelope for the same Key ID from one algorithm to another once the envelope is operational. The recommended procedure to change to a new algorithm is to take the adjacency down, make the necessary changes, and bring the adjacency back up. Obviously, an implementation MAY choose to stop verifying security envelope for the duration of algorithm change to keep the adjacency up but since this introduces a security vulnerability window, such roll-over SHOULD NOT be recommended.

7. Examples

7.1. Normal Operation

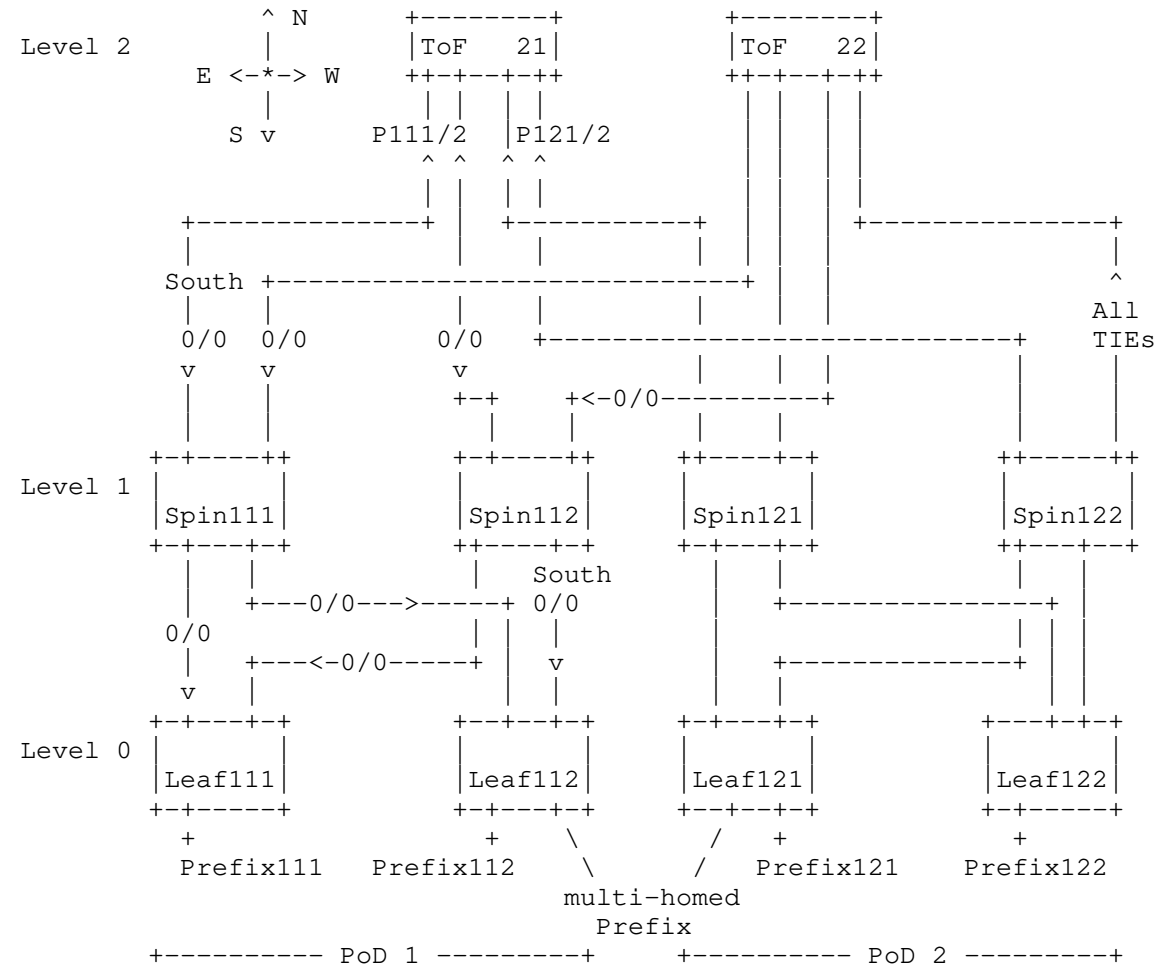


Figure 35: Normal Case Topology

This section describes RIFT deployment in the example topology given in Figure 35 without any node or link failures. The scenario disregards flooding reduction for simplicity's sake and compresses the node names in some cases to fit them into the picture better.

First, the following bi-directional adjacencies will be established:

1. ToF 21 (PoD 0) to Spine 111, Spine 112, Spine 121, and Spine 122

2. ToF 22 (PoD 0) to Spine 111, Spine 112, Spine 121, and Spine 122

3. Spine 111 to Leaf 111, Leaf 112

4. Spine 112 to Leaf 111, Leaf 112

5. Spine 121 to Leaf 121, Leaf 122

6. Spine 122 to Leaf 121, Leaf 122

Leaf 111 and Leaf 112 originate N-TIEs for Prefix 111 and Prefix 112 (respectively) to both Spine 111 and Spine 112 (Leaf 112 also originates an N-TIE for the multi-homed prefix). Spine 111 and Spine 112 will then originate their own N-TIEs, as well as flood the N-TIEs received from Leaf 111 and Leaf 112 to both ToF 21 and ToF 22.

Similarly, Leaf 121 and Leaf 122 originate North TIEs for Prefix 121 and Prefix 122 (respectively) to Spine 121 and Spine 122 (Leaf 121 also originates a North TIE for the multi-homed prefix). Spine 121 and Spine 122 will then originate their own North TIEs, as well as flood the North TIEs received from Leaf 121 and Leaf 122 to both ToF 21 and ToF 22.

Spines hold only North TIEs of level 0 for their PoD, while leaves only hold their own North TIEs while, at this point, both ToF 21 and ToF 22 (as well as any northbound connected controllers) would have the complete network topology.

ToF 21 and ToF 22 would then originate and flood South TIEs containing any established adjacencies and a default IP route to all spines. Spine 111, Spine 112, Spine 121, and Spine 122 will reflect all Node South TIEs received from ToF 21 to ToF 22, and all Node South TIEs from ToF 22 to ToF 21. South TIEs will not be re-propagated southbound.

South TIEs containing a default IP route are then originated by both Spine 111 and Spine 112 toward Leaf 111 and Leaf 112. Similarly, South TIEs containing a default IP route are originated by Spine 121 and Spine 122 toward Leaf 121 and Leaf 122.

At this point IP connectivity across maximum number of viable paths has been established for all leaves, with routing information constrained to only the minimum amount that allows for normal operation and redundancy.

7.2. Leaf Link Failure

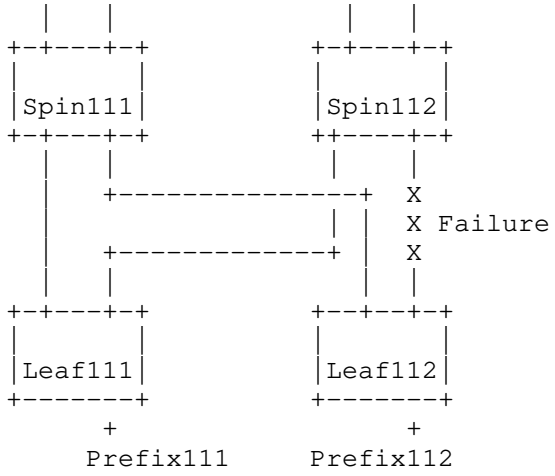


Figure 36: Single Leaf Link Failure

In the event of a link failure between Spine 112 and Leaf 112, both nodes will originate new Node TIEs that contain their connected adjacencies, except for the one that just failed. Leaf 112 will send a Node North TIE to Spine 111. Spine 112 will send a Node North TIE to ToF 21 and ToF 22 as well as a new Node South TIE to Leaf 111 that will be reflected to Spine 111. Necessary SPF recomputation will occur, resulting in Spine 112 no longer being in the forwarding path for Prefix 112.

Spine 111 will also disaggregate Prefix 112 by sending new Prefix South TIE to Leaf 111 and Leaf 112. Though disaggregation is covered in more detail in the following section, it is worth mentioning in this example as it further illustrates RIFT's mechanism to mitigate traffic loss. Consider that Leaf 111 has yet to receive the more specific (disaggregated) route from Spine 111. In such a scenario, traffic from Leaf 111 toward Prefix 112 may still use Spine 112's default route, causing it to traverse ToF 21 and ToF 22 back down via Spine 111. While this behavior is suboptimal, it is transient in nature and preferred to dropping traffic.

7.3. Partitioned Fabric

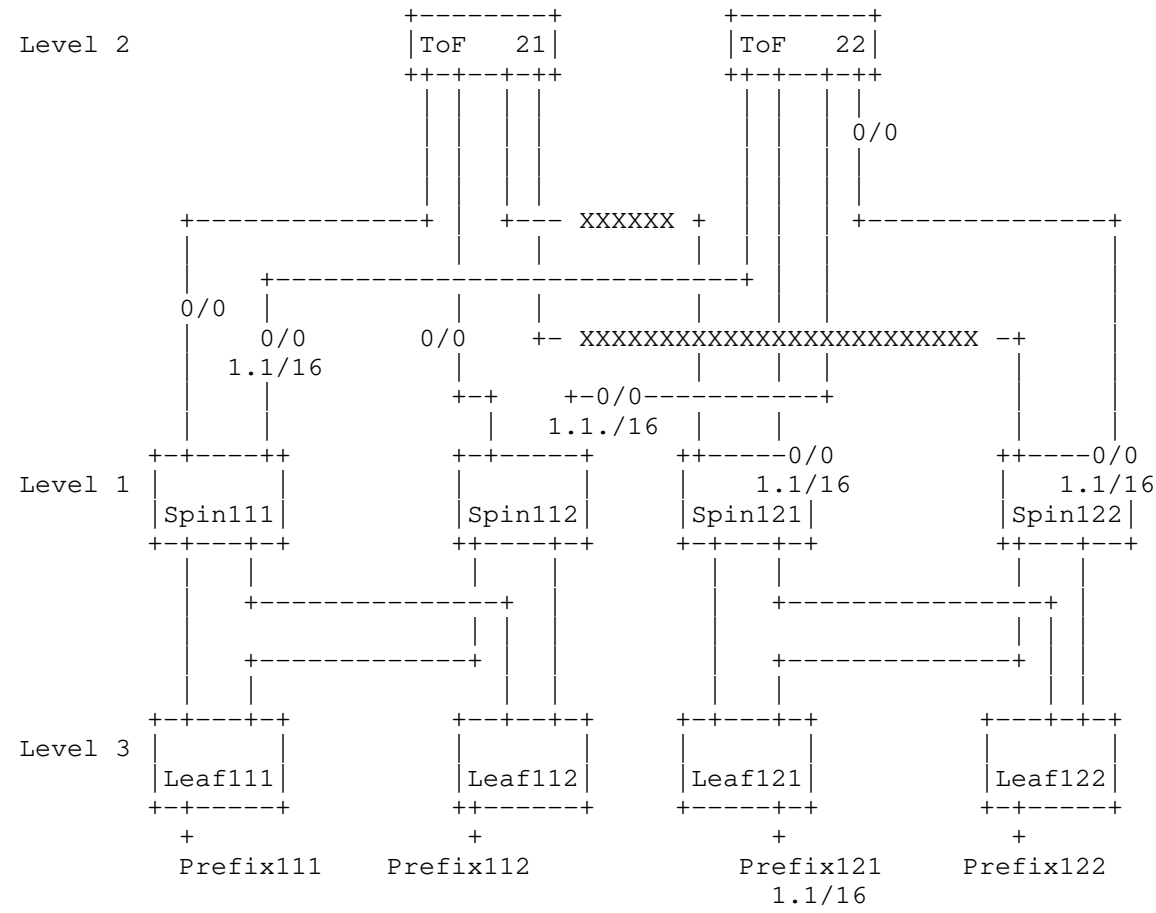


Figure 37: Fabric Partition

Figure 37 shows one of more catastrophic scenarios where ToF 21 is completely severed from access to Prefix 121 due to a double link failure. If only default routes existed, this would result in 50% of traffic from Leaf 111 and Leaf 112 toward Prefix 121 being dropped.

The mechanism to resolve this scenario hinges on ToF 21's South TIEs being reflected from Spine 111 and Spine 112 to ToF 22. Once ToF 22 is informed that Prefix 121 cannot be reached from ToF 21, it will begin to disaggregate Prefix 121 by advertising a more specific route (1.1/16) along with the default IP prefix route to all spines (ToF 21 still only sends a default route). The result is Spine 111 and Spine112 using the more specific route to Prefix 121 via ToF 22. All other prefixes continue to use the default IP prefix route toward both ToF 21 and ToF 22.

The more specific route for Prefix 121 being advertised by ToF 22 does not need to be propagated further south to the leaves, as they do not benefit from this information. Spine 111 and Spine 112 are only required to reflect the new South Node TIEs received from ToF 22 to ToF 21. In short, only the relevant nodes received the relevant updates, thereby restricting the failure to only the partitioned level rather than burdening the whole fabric with the flooding and recomputation of the new topology information.

To finish this example, the following table shows sets computed by ToF 22 using notation introduced in Section 6.5:

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

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

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

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

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

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

With that and |H (for r=Prefix 121) and |H (for r=Prefix 122) being disjoint from |A (for ToF 21), ToF 22 will originate a South TIE with Prefix 121 and Prefix 122, which will be flooded to all spines.

7.4. Northbound Partitioned Router and Optional East-West Links

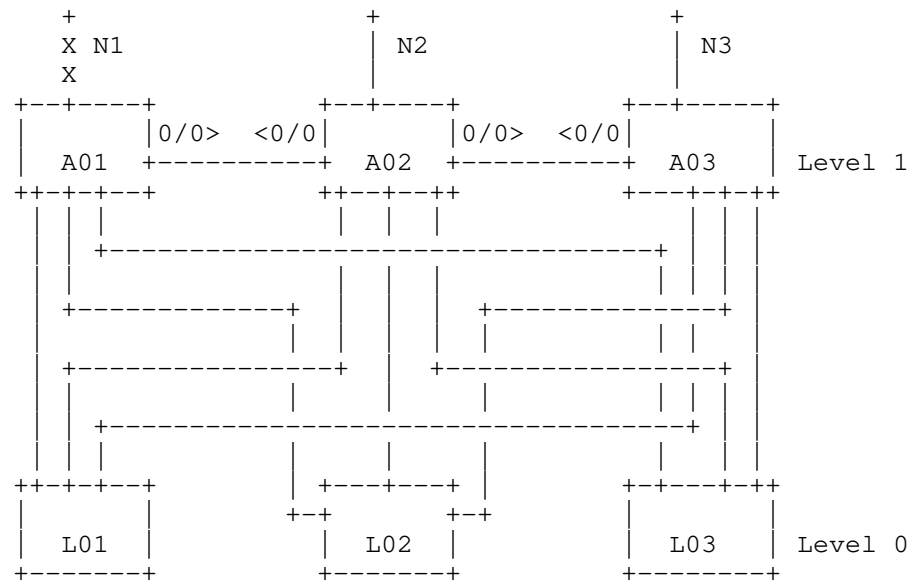


Figure 38: North Partitioned Router

Figure 38 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 6.4.1 A01 will compute northbound reachability by using the link A01 to A02. A02 however, will *not* use this link during N-SPF. The result is A01 utilizing the horizontal link for default route advertisement and unidirectional routing.

Furthermore, if A02 also loses its only northbound adjacency (N2), the situation evolves. A01 will no longer have northbound reachability while it receives A03's northbound adjacencies in South Node TIEs reflected by nodes south of it. As a result, A01 will no longer advertise its default route in accordance with Section 6.3.8.

8. Further Details on Implementation

8.1. Considerations for Leaf-Only Implementation

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

1. Leaf nodes only need to maintain a multipath default route under normal circumstances. However, in cases of catastrophic partitioning, leaf nodes SHOULD be capable of accommodating all the leaf routes in their own PoD to prevent traffic loss.

2. Leaf nodes hold only their own North TIEs and the South TIEs of Level 1 nodes they are connected to.

3. Leaf nodes do not have to support any type of disaggregation computation or propagation.

4. Leaf nodes are not required to support the overload flag.

5. Leaf nodes do not need to originate S-TIEs unless optional leaf-2-leaf features are desired.

8.2. Considerations for Spine Implementation

Nodes that do not act as ToF are not required to discover fallen leaves by comparing reachable destinations with peers and therefore do not need to run the computation of disaggregated routes based on that discovery. On the other hand, non-ToF nodes need to respect disaggregated routes advertised from the north. In the case of negative disaggregation, spines nodes need to generate southbound disaggregated routes when all parents are lost for a fallen leaf.

9. Security Considerations

9.1. General

One can consider attack vectors where a router may reboot many times while changing its System ID and pollute the network with many stale TIEs or TIEs that are sent with very long lifetimes and not cleaned up when the routes vanish. Those attack vectors are not unique to RIFT. Given large memory footprints available today those attacks should be relatively benign. Otherwise, a node SHOULD implement a strategy of discarding contents of all TIEs that were not present in the SPF tree over a certain, configurable period of time. Since the protocol 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 has an adjacency formed towards the System ID of the discarded TIEs.

9.2. Time to Live and Hop Limit Values

RIFT explicitly requires the use of a TTL/HL value of 1 *or* 255 when sending/receiving LIEs and TIEs so that implementors have a choice between the two.

Using a TTL/HL value of 255 does come with security concerns, but those risks are addressed in [RFC5082]. However, this approach may still have difficulties with some forwarding implementations (e.g. incorrectly processing TTL/HL, loops within forwarding plane itself, etc.).

It is for this reason that RIFT also allows implementations to use a TTL/HL of 1. Attacks that exploit this by spoofing it from several hops away are indeed possible, but are exceptionally difficult to engineer. Replay attacks are another potential attack vector, but as described in the subsequent security sections, RIFT is well protected against such attacks.

9.3. Malformed Packets

The protocol protects packets extensively through optional signatures and nonces so if the possibility of maliciously injected malformed or replayed packets exist in a deployment, this conclusively protects against such attacks.

Even with the security envelope, since RIFT relies on Thrift encoders and decoders generated automatically from IDL it is conceivable that errors in such encoders/decoders could be discovered and lead to delivery of corrupted packets or reception of packets that cannot be decoded. Misformatted packets lead normally to decoder returning an error condition to the caller and with that the packet is basically unparsable with no other choice but to discard it. Should the unlikely scenario occur of the decoder being forced to abort the protocol this is neither better nor worse than today’s behavior of other protocols.

9.4. ZTP

Section 6.7 presents many attack vectors in untrusted environments, starting with nodes that oscillate their level offers to the possibility of nodes offering a _ThreeWay_ adjacency with the highest possible level value and a very long holdtime trying to put itself "on top of the lattice" thereby allowing it to gain access to the whole southbound topology. Session authentication mechanisms are necessary in environments where this is possible and RIFT provides the security envelope to ensure this if so desired.

9.5. Lifetime

RIFT removes lifetime modification and replay attack vectors by protecting the lifetime behind a signature computed over it and additional nonce combination which results in the inability of an attacker to artificially shorten the _remaining_lifetime_.

9.6. Packet Number

An optional defined value number that is carried in the security envelope without any encryption protection and is hence vulnerable to replay and modification attacks. Contrary to nonces, this number must change on every packet and would present a very high cryptographic load if signed. The attack vector packet number present is relatively benign. Changing the packet number by a man-in-the-middle attack will only affect operational validation tools and possibly some performance optimizations on flooding. It is expected that an implementation detecting too many "fake losses" or "misorderings" due to the attack on the packet number would simply suppress its further processing.

9.7. Outer Fingerprint Attacks

A node can try to inject LIE packets observing a conversation on the wire by using the outer Key ID albeit it cannot generate valid hashes in case it changes the integrity of the message so the only possible attack is DoS due to excessive LIE validation.

A node can try to replay previous LIEs with changed state that it recorded but the attack is hard to replicate since the nonce combination must match the ongoing exchange and is then limited to a single flap only since both nodes will advance their nonces in case the adjacency state changed. Even in the most unlikely case the attack length is limited due to both sides periodically increasing their nonces.

Generally, since weak nonces are not changed on every packet for performance reasons a conceivable attack vector by a man-in-the-middle is to flood a receiving node with maximum bandwidth of recently observed packets, both LIEs as well as TIEs. In a scenario where such attacks are likely _maximum_valid_nonce_delta_ can be implemented as configurable, small value and _nonce_regeneration_interval_ configured to very small value as well. This will likely present a significant computational load on large fabrics under normal operation.

9.8. TIE Origin Fingerprint DoS Attacks

A compromised node can attempt to generate "fake TIEs" using other nodes' TIE origin key identifiers. Albeit the ultimate validation of the origin fingerprint will fail in such scenarios and not progress further than immediately peering nodes, the resulting denial of service attack seems unavoidable since the TIE origin Key ID is only protected by the, here assumed to be compromised, node.

9.9. Host Implementations

It can be reasonably expected that with the proliferation of RotH servers, rather than dedicated networking devices, will represent a significant amount of RIFT devices. Given their normally far wider software envelope and access granted to them, such servers are also far more likely to be compromised and present an attack vector on the protocol. Hijacking of prefixes to attract traffic is a trust problem and cannot be easily addressed within the protocol if the trust model is breached, i.e. the server presents valid credentials to form an adjacency and issue TIEs. In an even more devious way, the servers can present DoS (or even DDoS) vectors of issuing too many LIE packets, flooding large amounts of North TIEs, and attempting similar resource overrun attacks. A prudent implementation forming adjacencies to leaves should implement thresholds mechanisms and raise warnings when, e.g., a leaf is advertising an excess number of TIEs or prefixes. Additionally, such implementation could refuse any topology information except the node's own TIEs and authenticated, reflected South Node TIEs at own level.

To isolate possible attack vectors on the leaf to the largest possible extent a dedicated leaf-only implementation could run without any configuration by hard-coding a well-known adjacency key (which can be always rolled-over by the means of, e.g., well-known key-value distributed from top of the fabric), leaf level value and always setting overload flag. All other values can be derived by automatic means as described above.

9.9.1. IPv4 Broadcast and IPv6 All Routers Multicast Implementations

Section 6.2 describes an optional implementation that supports LIE exchange over IPv4 broadcast addresses and/or the IPv6 all routers multicast address. It is important to consider that if an implementation supports this, the attack surface widens as LIEs may be propagated to devices outside of the intended RIFT topology. This may leave RIFT nodes susceptible to the various attack vectors already described in this section.

10. IANA Considerations

This specification requests multicast address assignments and standard port numbers. Additionally registries for the schema are requested and suggested values provided that reflect the numbers allocated in the given schema.

10.1. Requested Multicast and Port Numbers

This document requests allocation in the 'IPv4 Multicast Address Space' registry the suggested value of 224.0.0.121 as 'ALL_V4_RIFT_ROUTERS' and in the 'IPv6 Multicast Address Space' registry the suggested value of FF02::A1F7 as 'ALL_V6_RIFT_ROUTERS'.

This document requests the following allocations from the "Service Name and Transport Protocol Port Number Registry":

RIFT LIE Port

Service Name: rift-lies
Transport Protocol(s): UDP
Assignee: Tony Przygienda (prz@juniper.net)
Contact: Jordan Head (jhead@juniper.net)
Description: Routing in Fat Trees Link Information Element
Reference: This Document
Port Number: 914

RIFT TIE Port

Service Name: rift-ties
Transport Protocol(s): UDP
Assignee: Tony Przygienda (prz@juniper.net)
Contact: Jordan Head (jhead@juniper.net)
Description: Routing in Fat Trees Topology Information Element
Reference: This Document
Port Number: 915

10.2. Requested Registries with Assigned Values

This section requests registries that help govern the schema via usual IANA registry procedures. A top-level group named 'RIFT' should hold the corresponding registries requested in the following sections with their pre-defined values. Registry values are stored with their minimum and maximum version in which they are available. All values not provided as to be considered 'Unassigned'. The range of every registry is a 16-bit integer. Allocation of new values is always performed via 'Expert Review' action.

10.2.1. Registry RIFT/Versions

This registry stores all RIFT protocol schema major and minor versions including the reference to the document introducing the version. This means as well that if multiple documents extend rift schema they have to serialize using this registry to increase the minor or major versions sequentially.

Schema Version	Reference
8.0	https://datatracker.ietf.org/doc/ draft-ietf-rift-rift/ Appendix B

Table 7

10.2.2. Registry RIFT/common/AddressFamilyType

The name of the registry should be CommonAddressFamilyType.

Address family type.

Schema Range	Registration Procedure
Major or Minor Change per Rules in section Appendix B	Expert Review
All Other Assignments	Specification Required

Table 8

Name	Value	Min. Schema Version	Max. Schema Version	Description
Illegal	0	8.0		
AddressFamilyMinValue	1	8.0		
IPv4	2	8.0		
IPv6	3	8.0		
AddressFamilyMaxValue	4	8.0		

Table 9

10.2.3. Registry RIFT/common/HierarchyIndications

The name of the registry should be CommonHierarchyIndications.

Flags indicating node configuration in case of ZTP.

Schema Range	Registration Procedure
Major or Minor Change per Rules in section Appendix B	Expert Review
All Other Assignments	Specification Required

Table 10

Name	Value	Min. Schema Version	Max. Schema Version	Description
leaf_only	0	8.0		
leaf_only_and_leaf_2_leaf_procedures	1	8.0		
top_of_fabric	2	8.0		

Table 11

10.2.4. Registry RIFT/common/IEEE802_1ASTimeStampType

The name of the registry should be CommonIEEE8021ASTimeStampType.

Timestamp per IEEE 802.1AS, all values MUST be interpreted in implementation as unsigned.

Schema Range	Registration Procedure
Major or Minor Change per Rules in section Appendix B	Expert Review
All Other Assignments	Specification Required

Table 12

Name	Value	Min. Schema Version	Max. Schema Version	Description
AS_sec	1	8.0		
AS_nsec	2	8.0		

Table 13

10.2.5. Registry RIFT/common/IPAddressType

The name of the registry should be CommonIPAddressType.

IP address type.

Schema Range	Registration Procedure
Major or Minor Change per Rules in section Appendix B	Expert Review
All Other Assignments	Specification Required

Table 14

Name	Value	Min. Schema Version	Max. Schema Version	Description
ipv4address	1	8.0		Content is IPv4
ipv6address	2	8.0		Content is IPv6

Table 15

10.2.6. Registry RIFT/common/IPPrefixType

The name of the registry should be CommonIPPrefixType.

Prefix advertisement.

@note: for interface addresses the protocol can propagate the address part beyond the subnet mask and on reachability computation that has to be normalized. The non-significant bits can be used for operational purposes.

Schema Range	Registration Procedure
Major or Minor Change per Rules in section Appendix B	Expert Review
All Other Assignments	Specification Required

Table 16

Name	Value	Min. Schema Version	Max. Schema Version	Description
ipv4prefix	1	8.0		
ipv6prefix	2	8.0		

Table 17

10.2.7. Registry RIFT/common/IPv4PrefixType

The name of the registry should be CommonIPv4PrefixType.

IPv4 prefix type.

Schema Range	Registration Procedure
Major or Minor Change per Rules in section Appendix B	Expert Review
All Other Assignments	Specification Required

Table 18

Name	Value	Min. Schema Version	Max. Schema Version	Description
address	1	8.0		
prefixlen	2	8.0		

Table 19

10.2.8. Registry RIFT/common/IPv6PrefixType

The name of the registry should be CommonIPv6PrefixType.

IPv6 prefix type.

Schema Range	Registration Procedure
Major or Minor Change per Rules in section Appendix B	Expert Review
All Other Assignments	Specification Required

Table 20

Name	Value	Min. Schema Version	Max. Schema Version	Description
address	1	8.0		
prefixlen	2	8.0		

Table 21

10.2.9. Registry RIFT/common/KVTypes

The name of the registry should be CommonKVTypes.

Schema Range	Registration Procedure
Major or Minor Change per Rules in section Appendix B	Expert Review
All Other Assignments	Specification Required

Table 22

Name	Value	Min. Schema Version	Max. Schema Version	Description
Experimental	1	8.0		
WellKnown	2	8.0		
OUI	3	8.0		

Table 23

10.2.10. Registry RIFT/common/PrefixSequenceType

The name of the registry should be CommonPrefixSequenceType.

Sequence of a prefix in case of move.

Schema Range	Registration Procedure
Major or Minor Change per Rules in section Appendix B	Expert Review
All Other Assignments	Specification Required

Table 24

Name	Value	Min. Schema Version	Max. Schema Version	Description
timestamp	1	8.0		
transactionid	2	8.0		Transaction ID set by client in e.g. in 6LoWPAN.

Table 25

10.2.11. Registry RIFT/common/RouteType

The name of the registry should be CommonRouteType.

RIFT route types. @note: The only purpose of those values is to introduce an ordering whereas an implementation can choose internally any other values as long the ordering is preserved

Schema Range	Registration Procedure
Major or Minor Change per Rules in section Appendix B	Expert Review
All Other Assignments	Specification Required

Table 26

Name	Value	Min. Schema Version	Max. Schema Version	Description
Illegal	0	8.0		
RouteTypeMinValue	1	8.0		
Discard	2	8.0		
LocalPrefix	3	8.0		
SouthPGPPrefix	4	8.0		
NorthPGPPrefix	5	8.0		
NorthPrefix	6	8.0		
NorthExternalPrefix	7	8.0		
SouthPrefix	8	8.0		
SouthExternalPrefix	9	8.0		
NegativeSouthPrefix	10	8.0		
RouteTypeMaxValue	11	8.0		

Table 27

10.2.12. Registry RIFT/common/TIETypeType

The name of the registry should be CommonTIETypeType.

Type of TIE.

Schema Range	Registration Procedure
Major or Minor Change per Rules in section Appendix B	Expert Review
All Other Assignments	Specification Required

Table 28

Name	Value	Min. Schema Version	Max. Schema Version	Description
Illegal	0	8.0		
TIETypeMinValue	1	8.0		
NodeTIEType	2	8.0		
PrefixTIEType	3	8.0		
PositiveDisaggregationPrefixTIEType	4	8.0		
NegativeDisaggregationPrefixTIEType	5	8.0		
PGPrefixTIEType	6	8.0		
KeyValueTIEType	7	8.0		
ExternalPrefixTIEType	8	8.0		
PositiveExternalDisaggregationPrefixTIEType	9	8.0		
TIETypeMaxValue	10	8.0		

Table 29

10.2.13. Registry RIFT/common/TieDirectionType

The name of the registry should be CommonTieDirectionType.

Direction of TIEs.

Schema Range	Registration Procedure
Major or Minor Change per Rules in section Appendix B	Expert Review
All Other Assignments	Specification Required

Table 30

Name	Value	Min. Schema Version	Max. Schema Version	Description
Illegal	0	8.0		
South	1	8.0		
North	2	8.0		
DirectionMaxValue	3	8.0		

Table 31

10.2.14. Registry RIFT/encoding/Community

The name of the registry should be EncodingCommunity.

Prefix community.

Schema Range	Registration Procedure
Major or Minor Change per Rules in section Appendix B	Expert Review
All Other Assignments	Specification Required

Table 32

Name	Value	Min. Schema Version	Max. Schema Version	Description
top	1	8.0		Higher order bits
bottom	2	8.0		Lower order bits

Table 33

10.2.15. Registry RIFT/encoding/KeyValueTIEElement

The name of the registry should be EncodingKeyValueTIEElement.

Generic key value pairs.

Schema Range	Registration Procedure
Major or Minor Change per Rules in section Appendix B	Expert Review
All Other Assignments	Specification Required

Table 34

Name	Value	Min. Schema Version	Max. Schema Version	Description
keyvalues	1	8.0		

Table 35

10.2.16. Registry RIFT/encoding/KeyValueTIEElementContent

The name of the registry should be EncodingKeyValueTIEElementContent.

Defines the targeted nodes and the value carried.

Schema Range	Registration Procedure
Major or Minor Change per Rules in section Appendix B	Expert Review
All Other Assignments	Specification Required

Table 36

Name	Value	Min. Schema Version	Max. Schema Version	Description
targets	1	8.0		
value	2	8.0		

Table 37

10.2.17. Registry RIFT/encoding/LIEPacket

The name of the registry should be EncodingLIEPacket.

RIFT LIE Packet.

@note: this node's level is already included on the packet header

Schema Range	Registration Procedure
Major or Minor Change per Rules in section Appendix B	Expert Review
All Other Assignments	Specification Required

Table 38

Name	Value	Min. Schema Version	Max. Schema Version	Description
name	1	8.0		Node or adjacency name.
local_id	2	8.0		Local link ID.
flood_port	3	8.0		UDP port to which we can receive flooded TIEs.

link_mtu_size	4	8.0		Layer 3 MTU, used to discover mismatch.
link_bandwidth	5	8.0		Local link bandwidth on the interface.
neighbor	6	8.0		Reflects the neighbor once received to provide 3-way connectivity.
pod	7	8.0		Node's PoD.
node_capabilities	10	8.0		Node capabilities supported.
link_capabilities	11	8.0		Capabilities of this link.
holdtime	12	8.0		Required holdtime of the adjacency, i.e. for how long a period should adjacency be kept up without valid LIE reception.
label	13	8.0		Optional, unsolicited, downstream assigned locally significant label value for the adjacency.

not_a_ztp_offer	21	8.0		Indicates that the level on the LIE must not be used to derive a ZTP level by the receiving node.
you_are_flood_repeater	22	8.0		Indicates to northbound neighbor that it should be reflooding TIEs received from this node to achieve flood reduction and balancing for northbound flooding.
you_are_sending_too_quickly	23	8.0		Indicates to neighbor to flood node TIEs only and slow down all other TIEs. Ignored when received from southbound neighbor.
instance_name	24	8.0		Instance name in case multiple RIFT instances running on same interface.
fabric_id	35	8.0		It provides the optional ID of the Fabric configured.

				This MUST match the information advertised on the node element.

Table 39

10.2.18. Registry RIFT/encoding/LinkCapabilities

The name of the registry should be EncodingLinkCapabilities.

Link capabilities.

Schema Range	Registration Procedure
Major or Minor Change per Rules in section Appendix B	Expert Review
All Other Assignments	Specification Required

Table 40

Name	Value	Min. Schema Version	Max. Schema Version	Description
bfd	1	8.0		Indicates that the link is supporting BFD.
ipv4_forwarding_capable	2	8.0		Indicates whether the interface will support IPv4 forwarding.

Table 41

10.2.19. Registry RIFT/encoding/LinkIDPair

The name of the registry should be EncodingLinkIDPair.

LinkID pair describes one of parallel links between two nodes.

Schema Range	Registration Procedure
Major or Minor Change per Rules in section Appendix B	Expert Review
All Other Assignments	Specification Required

Table 42

Name	Value	Min. Schema Version	Max. Schema Version	Description
local_id	1	8.0		Node-wide unique value for the local link.
remote_id	2	8.0		Received remote link ID for this link.
platform_interface_index	10	8.0		Describes the local interface index of the link.
platform_interface_name	11	8.0		Describes the local interface name.
trusted_outer_security_key	12	8.0		Indicates whether the link is

				secured, i.e. protected by outer key, absence of this element means no indication, undefined outer key means not secured.
bfd_up	13	8.0		Indicates whether the link is protected by established BFD session.
address_families	14	8.0		Optional indication which address families are up on the interface

Table 43

10.2.20. Registry RIFT/encoding/Neighbor

The name of the registry should be EncodingNeighbor.

Neighbor structure.

Schema Range	Registration Procedure
Major or Minor Change per Rules in section Appendix B	Expert Review
All Other Assignments	Specification Required

Table 44

Name	Value	Min. Schema Version	Max. Schema Version	Description
originator	1	8.0		System ID of the originator.
remote_id	2	8.0		ID of remote side of the link.

Table 45

10.2.21. Registry RIFT/encoding/NodeCapabilities

The name of the registry should be EncodingNodeCapabilities.

Capabilities the node supports.

Schema Range	Registration Procedure
Major or Minor Change per Rules in section Appendix B	Expert Review
All Other Assignments	Specification Required

Table 46

Name	Value	Min. Schema Version	Max. Schema Version	Description
protocol_minor_version	1	8.0		Must advertise supported minor version dialect that way.
flood_reduction	2	8.0		indicates that node supports flood reduction.
hierarchy_indications	3	8.0		indicates place in hierarchy, i.e. top-of-fabric or leaf only (in ZTP) or support for leaf-2-leaf procedures.

Table 47

10.2.22. Registry RIFT/encoding/NodeFlags

The name of the registry should be EncodingNodeFlags.

Indication flags of the node.

Schema Range	Registration Procedure
Major or Minor Change per Rules in section Appendix B	Expert Review
All Other Assignments	Specification Required

Table 48

Name	Value	Min. Schema Version	Max. Schema Version	Description
overload	1	8.0		Indicates that node is in overload, do not transit traffic through it.

Table 49

10.2.23. Registry RIFT/encoding/NodeNeighborsTIEElement

The name of the registry should be EncodingNodeNeighborsTIEElement.

neighbor of a node

Schema Range	Registration Procedure
Major or Minor Change per Rules in section Appendix B	Expert Review
All Other Assignments	Specification Required

Table 50

Name	Value	Min. Schema Version	Max. Schema Version	Description
level	1	8.0		level of neighbor
cost	3	8.0		Cost to neighbor. Ignore anything larger than 'infinite_distance' and 'invalid_distance'
link_ids	4	8.0		can carry description of multiple parallel links in a TIE
bandwidth	5	8.0		total bandwith to neighbor as sum of all parallel links

Table 51

10.2.24. Registry RIFT/encoding/NodeTIEElement

The name of the registry should be EncodingNodeTIEElement.

Description of a node.

Schema Range	Registration Procedure
Major or Minor Change per Rules in section Appendix B	Expert Review
All Other Assignments	Specification Required

Table 52

Name	Value	Min. Schema Version	Max. Schema Version	Description
level	1	8.0		Level of the node.
neighbors	2	8.0		Node's neighbors. Multiple node TIEs can carry disjoint sets of neighbors.
capabilities	3	8.0		Capabilities of the node.
flags	4	8.0		Flags of the node.
name	5	8.0		Optional node name for easier operations.
pod	6	8.0		PoD to which the node belongs.
startup_time	7	8.0		optional startup time of the node
miscabled_links	10	8.0		If any local links are miscabled, this indication is flooded.
same_plane_tofs	12	8.0		ToFs in the same plane. Only carried by ToF. Multiple Node TIEs can carry disjoint sets of ToFs which MUST be joined to form a single set.
fabric_id	20	8.0		It provides the optional ID of the Fabric configured

Table 53

10.2.25. Registry RIFT/encoding/PacketContent

The name of the registry should be EncodingPacketContent.

Content of a RIFT packet.

Schema Range	Registration Procedure
Major or Minor Change per Rules in section Appendix B	Expert Review
All Other Assignments	Specification Required

Table 54

Name	Value	Min. Schema Version	Max. Schema Version	Description
lie	1	8.0		
tide	2	8.0		
tire	3	8.0		
tie	4	8.0		

Table 55

10.2.26. Registry RIFT/encoding/PacketHeader

The name of the registry should be EncodingPacketHeader.

Common RIFT packet header.

Schema Range	Registration Procedure
Major or Minor Change per Rules in section Appendix B	Expert Review
All Other Assignments	Specification Required

Table 56

Name	Value	Min. Schema Version	Max. Schema Version	Description
major_version	1	8.0		Major version of protocol.
minor_version	2	8.0		Minor version of protocol.
sender	3	8.0		Node sending the packet, in case of LIE/TIRE/TIDE also the originator of it.
level	4	8.0		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.

Table 57

10.2.27. Registry RIFT/encoding/PrefixAttributes

The name of the registry should be EncodingPrefixAttributes.

Attributes of a prefix.

Schema Range	Registration Procedure
Major or Minor Change per Rules in section Appendix B	Expert Review
All Other Assignments	Specification Required

Table 58

Name	Value	Min. Schema Version	Max. Schema Version	Description
metric	2	8.0		Distance of the prefix.
tags	3	8.0		Generic unordered set of route tags, can be redistributed to other protocols or use within the context of real time analytics.
monotonic_clock	4	8.0		Monotonic clock for mobile addresses.
loopback	6	8.0		Indicates if the prefix is a node loopback.
directly_attached	7	8.0		Indicates that the prefix is directly attached.
from_link	10	8.0		link to which the address belongs to.
label	12	8.0		Optional, per prefix significant label.

Table 59

10.2.28. Registry RIFT/encoding/PrefixTIEElement

The name of the registry should be EncodingPrefixTIEElement.

TIE carrying prefixes

Schema Range	Registration Procedure
Major or Minor Change per Rules in section Appendix B	Expert Review
All Other Assignments	Specification Required

Table 60

Name	Value	Min. Schema Version	Max. Schema Version	Description
prefixes	1	8.0		Prefixes with the associated attributes.

Table 61

10.2.29. Registry RIFT/encoding/ProtocolPacket

The name of the registry should be EncodingProtocolPacket.

RIFT packet structure.

Schema Range	Registration Procedure
Major or Minor Change per Rules in section Appendix B	Expert Review
All Other Assignments	Specification Required

Table 62

Name	Value	Min. Schema Version	Max. Schema Version	Description
header	1	8.0		
content	2	8.0		

Table 63

10.2.30. Registry RIFT/encoding/TIDEPacket

The name of the registry should be EncodingTIDEPacket.

TIDE with *sorted* TIE headers.

Schema Range	Registration Procedure
Major or Minor Change per Rules in section Appendix B	Expert Review
All Other Assignments	Specification Required

Table 64

Name	Value	Min. Schema Version	Max. Schema Version	Description
start_range	1	8.0		First TIE header in the tide packet.
end_range	2	8.0		Last TIE header in the tide packet.
headers	3	8.0		_Sorted_ list of headers.

Table 65

10.2.31. Registry RIFT/encoding/TIEElement

The name of the registry should be EncodingTIEElement.

Single element in a TIE.

Schema Range	Registration Procedure
Major or Minor Change per Rules in section Appendix B	Expert Review
All Other Assignments	Specification Required

Table 66

Name	Value	Min. Schema Version	Max. Schema Version	Description
node	1	8.0		Used in case of enum common.TIETypeType.NodeTIEType.
prefixes	2	8.0		Used in case of enum common.TIETypeType.PrefixTIEType.
positive_disaggregation_prefixes	3	8.0		Positive prefixes (always southbound).
negative_disaggregation_prefixes	5	8.0		Transitive, negative prefixes (always southbound)
external_prefixes	6	8.0		Externally reimported prefixes.
positive_external_disaggregation_prefixes	7	8.0		Positive external disaggregated

				prefixes (always southbound).
keyvalues	9	8.0		Key-Value store elements.

Table 67

10.2.32. Registry RIFT/encoding/TIEHeader

The name of the registry should be EncodingTIEHeader.

Header of a TIE.

Schema Range	Registration Procedure
Major or Minor Change per Rules in section Appendix B	Expert Review
All Other Assignments	Specification Required

Table 68

Name	Value	Min. Schema Version	Max. Schema Version	Description
tieid	2	8.0		ID of the tie.
seq_nr	3	8.0		Sequence number of the tie.
origination_time	10	8.0		Absolute timestamp when the TIE was generated.
origination_lifetime	12	8.0		Original lifetime when the TIE was generated.

Table 69

10.2.33. Registry RIFT/encoding/TIEHeaderWithLifeTime

The name of the registry should be EncodingTIEHeaderWithLifeTime.

Header of a TIE as described in TIRE/TIDE.

Schema Range	Registration Procedure
Major or Minor Change per Rules in section Appendix B	Expert Review
All Other Assignments	Specification Required

Table 70

Name	Value	Min. Schema Version	Max. Schema Version	Description
header	1	8.0		
remaining_lifetime	2	8.0		Remaining lifetime.

Table 71

10.2.34. Registry RIFT/encoding/TIEID

The name of the registry should be EncodingTIEID.

Unique ID of a TIE.

Schema Range	Registration Procedure
Major or Minor Change per Rules in section Appendix B	Expert Review
All Other Assignments	Specification Required

Table 72

Name	Value	Min. Schema Version	Max. Schema Version	Description
direction	1	8.0		direction of TIE
originator	2	8.0		indicates originator of the TIE
tietype	3	8.0		type of the tie
tie_nr	4	8.0		number of the tie

Table 73

10.2.35. Registry RIFT/encoding/TIEPacket

The name of the registry should be EncodingTIEPacket.

TIE packet

Schema Range	Registration Procedure
Major or Minor Change per Rules in section Appendix B	Expert Review
All Other Assignments	Specification Required

Table 74

Name	Value	Min. Schema Version	Max. Schema Version	Description
header	1	8.0		
element	2	8.0		

Table 75

10.2.36. Registry RIFT/encoding/TIREPacket

The name of the registry should be EncodingTIREPacket.

TIRE packet

Schema Range	Registration Procedure
Major or Minor Change per Rules in section Appendix B	Expert Review
All Other Assignments	Specification Required

Table 76

Name	Value	Min. Schema Version	Max. Schema Version	Description
headers	1	8.0		

Table 77

11. Acknowledgments

A new routing protocol in its complexity is not a product of a parent but of a village as the author list shows already. However, many more people provided input, fine-combed the specification based on their experience in design, implementation or application of protocols in IP fabrics. This section will make an inadequate attempt in recording their contribution.

Many thanks to Naiming Shen for some of the early discussions around the topic of using IGPs for routing in topologies related to Clos. Russ White to be especially acknowledged for the key conversation on epistemology that allowed to tie current asynchronous distributed systems theory results to a modern protocol design presented in this scope. Adrian Farrel, Joel Halpern, Jeffrey Zhang, Krzysztof Szarkowicz, Nagendra Kumar, Melchior Aelmans, Kaushal Tank, Will Jones, Moin Ahmed, Sandy Zhang, Donald Eastlake provided thoughtful comments that improved the readability of the document and found good amount of corners where the light failed to shine. Kris Price was first to mention single router, single arm default considerations. Jeff Tantsura helped out with some initial thoughts on BFD

interactions while Jeff Haas corrected several misconceptions about BFD's finer points and helped to improve the security section around leaf considerations. Artur Makutunowicz pointed out many possible improvements and acted as sounding board in regard to modern protocol implementation techniques RIFT is exploring. Barak Gafni formalized first time clearly the problem of partitioned spine and fallen leaves on a (clean) napkin in Singapore that led to the very important part of the specification centered around multiple ToF planes and negative disaggregation. Igor Gashinsky and others shared many thoughts on problems encountered in design and operation of large-scale data center fabrics. Xu Benchong found a delicate error in the flooding procedures and a schema datatype size mismatch.

Last but not least, Alvaro Retana, John Scudder and Andrew Alaton guided the undertaking as ADs by asking many necessary procedural and technical questions which did not only improve the content but did also lay out the track towards publication.

12. Contributors

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

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

Table 78: RIFT Authors

13. References

13.1. Normative References

[EUI64] IEEE, "Guidelines for Use of Extended Unique Identifier (EUI), Organizationally Unique Identifier (OUI), and Company ID (CID)", IEEE EUI, <http://standards.ieee.org/develop/regauth/tut/eui.pdf>.

[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>.

[RFC2365] Meyer, D., "Administratively Scoped IP Multicast", BCP 23, RFC 2365, DOI 10.17487/RFC2365, July 1998, <https://www.rfc-editor.org/info/rfc2365>.

[RFC4291] Hinden, R. and S. Deering, "IP Version 6 Addressing Architecture", RFC 4291, DOI 10.17487/RFC4291, February 2006, <https://www.rfc-editor.org/info/rfc4291>.

[RFC5082] Gill, V., Heasley, J., Meyer, D., Savola, P., Ed., and C. Pignataro, "The Generalized TTL Security Mechanism (GTSM)", RFC 5082, DOI 10.17487/RFC5082, October 2007, <https://www.rfc-editor.org/info/rfc5082>.

[RFC5120] Przygienda, T., Shen, N., and N. Sheth, "M-ISIS: Multi Topology (MT) Routing in Intermediate System to Intermediate Systems (IS-ISs)", RFC 5120, DOI 10.17487/RFC5120, February 2008, <https://www.rfc-editor.org/info/rfc5120>.

[RFC5709] Bhatia, M., Manral, V., Fanto, M., White, R., Barnes, M., Li, T., and R. Atkinson, "OSPFv2 HMAC-SHA Cryptographic Authentication", RFC 5709, DOI 10.17487/RFC5709, October 2009, <https://www.rfc-editor.org/info/rfc5709>.

[RFC5881] Katz, D. and D. Ward, "Bidirectional Forwarding Detection (BFD) for IPv4 and IPv6 (Single Hop)", RFC 5881, DOI 10.17487/RFC5881, June 2010, <https://www.rfc-editor.org/info/rfc5881>.

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

[RFC7987] Ginsberg, L., Wells, P., Decraene, B., Przygienda, T., and H. Gredler, "IS-IS Minimum Remaining Lifetime", RFC 7987, DOI 10.17487/RFC7987, October 2016, <https://www.rfc-editor.org/info/rfc7987>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

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

[RFC8202] Ginsberg, L., Previdi, S., and W. Henderickx, "IS-IS Multi-Instance", RFC 8202, DOI 10.17487/RFC8202, June 2017, <https://www.rfc-editor.org/info/rfc8202>.

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

[RFC9300] Farinacci, D., Fuller, V., Meyer, D., Lewis, D., and A. Cabellos, Ed., "The Locator/ID Separation Protocol (LISP)", RFC 9300, DOI 10.17487/RFC9300, October 2022, <https://www.rfc-editor.org/info/rfc9300>.

[RFC9301] Farinacci, D., Maino, F., Fuller, V., and A. Cabellos, Ed., "Locator/ID Separation Protocol (LISP) Control Plane", RFC 9301, DOI 10.17487/RFC9301, October 2022, <https://www.rfc-editor.org/info/rfc9301>.

[thrift] Apache Software Foundation, "Thrift Language Implementation and Documentation", <https://github.com/apache/thrift/tree/0.15.0/doc>.

13.2. Informative References

[APPLICABILITY] Wei, Y., Zhang, Z., Afanasiev, D., Thubert, P., and T. Przygienda, "RIFT Applicability", Work in Progress, Internet-Draft, draft-ietf-rift-applicability-12, 25 December 2023, <https://datatracker.ietf.org/doc/html/draft-ietf-rift-applicability-12>.

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

[DayOne] Aelmans, M., Vandezande, O., Rijsman, B., Head, J., Graf, C., Alberro, L., Mali, H., and O. Steudler, "Day One: Routing in Fat Trees (RIFT)", Juniper DayOne .

[DIJKSTRA] Dijkstra, E. W., "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. E., "Fat-Trees: Universal Networks for Hardware-Efficient Supercomputing", 1985.

[IEEEstd1588]
IEEE, "IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems", IEEE Standard 1588, <https://ieeexplore.ieee.org/document/4579760/>.

[IEEEstd8021AS]
IEEE, "IEEE Standard for Local and Metropolitan Area Networks - Timing and Synchronization for Time-Sensitive Applications in Bridged Local Area Networks", IEEE Standard 802.1AS, <https://ieeexplore.ieee.org/document/5741898/>.

[RFC0826] Plummer, D., "An Ethernet Address Resolution Protocol: Or Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware", STD 37, RFC 826, DOI 10.17487/RFC0826, November 1982, <https://www.rfc-editor.org/info/rfc826>.

[RFC1982] Elz, R. and R. Bush, "Serial Number Arithmetic", RFC 1982, DOI 10.17487/RFC1982, August 1996, <https://www.rfc-editor.org/info/rfc1982>.

[RFC2131] Droms, R., "Dynamic Host Configuration Protocol", RFC 2131, DOI 10.17487/RFC2131, March 1997, <https://www.rfc-editor.org/info/rfc2131>.

[RFC2474] Nichols, K., Blake, S., Baker, F., and D. Black, "Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers", RFC 2474, DOI 10.17487/RFC2474, December 1998, <https://www.rfc-editor.org/info/rfc2474>.

[RFC4086] Eastlake 3rd, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, DOI 10.17487/RFC4086, June 2005, <https://www.rfc-editor.org/info/rfc4086>.

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

[RFC4862] Thomson, S., Narten, T., and T. Jinmei, "IPv6 Stateless Address Autoconfiguration", RFC 4862, DOI 10.17487/RFC4862, September 2007, <https://www.rfc-editor.org/info/rfc4862>.

[RFC5837] Atlas, A., Ed., Bonica, R., Ed., Pignataro, C., Ed., Shen, N., and JR. Rivers, "Extending ICMP for Interface and Next-Hop Identification", RFC 5837, DOI 10.17487/RFC5837, April 2010, <https://www.rfc-editor.org/info/rfc5837>.

[RFC5880] Katz, D. and D. Ward, "Bidirectional Forwarding Detection (BFD)", RFC 5880, DOI 10.17487/RFC5880, June 2010, <https://www.rfc-editor.org/info/rfc5880>.

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

[RFC8415] Mrugalski, T., Siodelski, M., Volz, B., Yourtchenko, A., Richardson, M., Jiang, S., Lemon, T., and T. Winters, "Dynamic Host Configuration Protocol for IPv6 (DHCPv6)", RFC 8415, DOI 10.17487/RFC8415, November 2018, <https://www.rfc-editor.org/info/rfc8415>.

[VAHDAT08] Al-Fares, M., Loukissas, A., and A. Vahdat, "A Scalable, Commodity Data Center Network Architecture", SIGCOMM , 2008.

[VFR] Giotsas, V. and S. Zhou, "Valley-free violation in Internet routing - Analysis based on BGP Community data", 2012 IEEE International Conference on Communications (ICC) , 2012.

Appendix A. Sequence Number Binary Arithmetic

Assuming straight two complement's subtractions on the bit-width of the sequence numbers, the corresponding >: and =: relations are defined as:

U_1, U_2 are 12-bits aligned unsigned version number

D_f is (U_1 - U_2) interpreted as two complement signed 12-bits
D_b is (U_2 - U_1) interpreted as two complement signed 12-bits

U_1 >: U_2 IIF D_f > 0 *and* D_b < 0
U_1 =: U_2 IIF D_f = 0

The >: relationship is anti-symmetric but not transitive. Observe that this leaves >: of the numbers having maximum two complement distance, e.g. (0 and 0x800) undefined in the 12-bits case since D_f and D_b are both -0x7ff.

A simple example of the relationship in case of 3-bit arithmetic follows as table indicating D_f/D_b values and then the relationship of U_1 to U_2:

U2 / U1	0	1	2	3	4	5	6	7
0	+/+	+/-	+/-	+/-	-/-	-/+	-/+	-/+
1	-/+	+/+	+/-	+/-	+/-	-/-	-/+	-/+
2	-/+	-/+	+/+	+/-	+/-	+/-	-/-	-/+
3	-/+	-/+	-/+	+/+	+/-	+/-	+/-	-/-
4	-/-	-/+	-/+	-/+	+/+	+/-	+/-	+/-
5	+/-	-/-	-/+	-/+	-/+	+/+	+/-	+/-
6	+/-	+/-	-/-	-/+	-/+	-/+	+/+	+/-
7	+/-	+/-	+/-	-/-	-/+	-/+	-/+	+/+

U2 / U1	0	1	2	3	4	5	6	7
0	=	>	>	>	?	<	<	<
1	<	=	>	>	>	?	<	<
2	<	<	=	>	>	>	?	<
3	<	<	<	=	>	>	>	?
4	?	<	<	<	=	>	>	>
5	>	?	<	<	<	=	>	>
6	>	>	?	<	<	<	=	>
7	>	>	>	?	<	<	<	=

Appendix B. Information Elements Schema

This section introduces the schema for information elements. The IDL is Thrift [thrift].

On schema changes that

1. change field numbers *or*
2. add new *required* fields *or*
3. remove any fields *or*
4. change lists into sets, unions into structures *or*
5. change multiplicity of fields *or*
6. changes type or name of any field *or*
7. change data types of the type of any field *or*
8. adds, changes or removes a default value of any *existing* field *or*
9. removes or changes any defined constant or constant value *or*
10. changes any enumeration type except extending ‘common.TIETypeType‘ (use of enumeration types is generally discouraged) *or*
11. adds new TIE type to _TIETypeType_ with flooding scope different from prefix TIE flooding scope

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

Introducing an optional field does not cause a major version increase even if the fields inside the structure are optional with defaults.

All signed integer as forced by Thrift [thrift] support must be cast for internal purposes to equivalent unsigned values without discarding the signedness bit. An implementation SHOULD try to avoid using the signedness bit when generating values.

The schema is normative.

B.1. Backwards-Compatible Extension of Schema

The set of rules in Appendix B guarantees that every decoder can process serialized content generated by a higher minor version of the schema and with that the protocol can progress without a 'flag-day'. Contrary to that, content serialized using a major version X is *not* expected to be decodable by any implementation using decoder for a model with a major version lower than X. Schema negotiation and translation within RIFT is outside the scope of this document.

Additionally, based on the propagated minor version in encoded content and added optional node capabilities new TIE types or even de-facto mandatory fields can be introduced without progressing the major version albeit only nodes supporting such new extensions would decode them. Given the model is encoded at the source and never re-encoded flooding through nodes not understanding any new extensions will preserve the corresponding fields. However, it is important to understand that a higher minor version of a schema does *not* guarantee that capabilities introduced in lower minors of the same major are supported. The _node_capabilities_ field is used to indicate which capabilities are supported.

Specifically, the schema SHOULD add elements to _NodeCapabilities_ field future capabilities to indicate whether it will support interpretation of schema extensions on the same major revision if they are present. Such fields MUST be optional and have an implicit or explicit false default value. If a future capability changes route selection or generates conditions that cause packet loss if some nodes are not supporting it then a major version increment will be however unavoidable. _NodeCapabilities_ shown in LIE MUST match the capabilities shown in the Node TIEs, otherwise the behavior is unspecified. A node detecting the mismatch SHOULD generate a notification.

Alternately or additionally, new optional fields can be introduced into e.g. _NodeTIEElement_ if a special field is chosen to indicate via its presence that an optional feature is enabled (since capability to support a feature does not necessarily mean that the feature is actually configured and operational).

To support new TIE types without increasing the major version enumeration _TIEElement_ can be extended with new optional elements for new `common.TIETypeType` values as long the scope of the new TIE matches the prefix TIE scope. In case it is necessary to understand whether all nodes can parse the new TIE type a node capability MUST be added in _NodeCapabilities_ to prevent a non-homogenous network.

B.2. common.thrift

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

namespace py common

/** @note MUST be interpreted in implementation as unsigned 64 bits.
 */
typedef i64      SystemIDType
typedef i32      IPv4Address
typedef i32      MTUSizeType
/** @note MUST be interpreted in implementation as unsigned
    rolling over number */
typedef i64      SeqNrType
/** @note MUST be interpreted in implementation as unsigned */
typedef i32      LifeTimeInSecType
/** @note MUST be interpreted in implementation as unsigned */
typedef i8       LevelType
typedef i16      PacketNumberType
/** @note MUST be interpreted in implementation as unsigned */
typedef i32      PodType
/** @note MUST be interpreted in implementation as unsigned.
/** this has to be long enough to accomodate prefix */
typedef binary   IPv6Address
/** @note MUST be interpreted in implementation as unsigned */
typedef i16      UDPPortType
/** @note MUST be interpreted in implementation as unsigned */
typedef i32      TIENrType
/** @note MUST be interpreted in implementation as unsigned
          This is carried in the
          security envelope and must hence fit into 8 bits. */
typedef i8       VersionType
/** @note MUST be interpreted in implementation as unsigned */
typedef i16      MinorVersionType
/** @note MUST be interpreted in implementation as unsigned */
typedef i32      MetricType
/** @note MUST be interpreted in implementation as unsigned
          and unstructured */
typedef i64      RouteTagType
/** @note MUST be interpreted in implementation as unstructured
          label value */
typedef i32      LabelType
/** @note MUST be interpreted in implementation as unsigned */
typedef i32      BandwithInMegaBitsType
/** @note Key Value Key ID type */
typedef i32      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
/** @note MUST be interpreted in implementation as unsigned,
          especially since we have the /128 IPv6 case. */
typedef i8     PrefixLenType
/** timestamp in seconds since the epoch */
typedef i64    TimestampInSecsType
/** security nonce.
    @note MUST be interpreted in implementation as rolling
          over unsigned value */
typedef i16    NonceType
/** LIE FSM holdtime type */
typedef i16    TimeIntervalInSecType
/** Transaction ID type for prefix mobility as specified by RFC6550,
    value MUST be interpreted in implementation as unsigned  */
typedef i8     PrefixTransactionIDType
/** Timestamp per IEEE 802.1AS, all values MUST be interpreted in
    implementation as unsigned.  */
struct IEEE802_1ASTimeStampType {
    1: required     i64     AS_sec;
    2: optional     i32     AS_nsec;
}
/** generic counter type */
typedef i64 CounterType
/** Platform Interface Index type, i.e. index of interface on hardware,
    can be used e.g. with RFC5837 */
typedef i32 PlatformInterfaceIndex

/** Flags indicating node configuration in case of ZTP.
 */
enum HierarchyIndications {
    /** forces level to 'leaf_level' and enables according procedures */
    leaf_only                           = 0,
    /** forces level to 'leaf_level' and enables according procedures */
    leaf_only_and_leaf_2_leaf_procedures = 1,
    /** forces level to 'top_of_fabric' and enables according
        procedures */
    top_of_fabric                       = 2,
}

const PacketNumberType  undefined_packet_number    = 0
/** used when node is configured as top of fabric in ZTP.*/
const LevelType   top_of_fabric_level              = 24
/** default bandwidth on a link */
const BandwithInMegaBitsType  default_bandwidth    = 100
```

```
/** fixed leaf level when ZTP is not used */
const LevelType   leaf_level                  = 0
const LevelType   default_level               = leaf_level
const PodType     default_pod                 = 0
const LinkIDType  undefined_linkid            = 0

/** invalid key for key value */
const KeyIDType   invalid_key_value_key    = 0
/** default distance used */
const MetricType  default_distance         = 1
/** any distance larger than this will be considered infinity */
const MetricType  infinite_distance       = 0x7FFFFFFF
/** represents invalid distance */
const MetricType  invalid_distance        = 0
const bool overload_default               = false
const bool flood_reduction_default        = true
/** default LIE FSM LIE TX internval time */
const TimeIntervalInSecType   default_lie_tx_interval  = 1
/** default LIE FSM holddown time */
const TimeIntervalInSecType   default_lie_holdtime  = 3
/** multipler for default_lie_holdtime to hold down multiple neighbors */
const i8                      multiple_neighbors_lie_holdtime_multipler = 4
/** default ZTP FSM holddown time */
const TimeIntervalInSecType   default_ztp_holdtime  = 1
/** by default LIE levels are ZTP offers */
const bool default_not_a_ztp_offer        = false
/** by default everyone is repeating flooding */
const bool default_you_are_flood_repeater = true
/** 0 is illegal for SystemID */
const SystemIDType IllegalSystemID        = 0
/** empty set of nodes */
const set<SystemIDType> empty_set_of_nodeids = {}
/** default lifetime of TIE is one week */
const LifeTimeInSecType default_lifetime      = 604800
/** default lifetime when TIEs are purged is 5 minutes */
const LifeTimeInSecType purge_lifetime        = 300
/** optional round down interval when TIEs are sent with security hashes
    to prevent excessive computation. **/
const LifeTimeInSecType rounddown_lifetime_interval = 60
/** any `TieHeader` that has a smaller lifetime difference
    than this constant is equal (if other fields equal). */
const LifeTimeInSecType lifetime_diff2ignore  = 400

/** default UDP port to run LIEs on */
const UDPPortType     default_lie_udp_port       =  914
/** default UDP port to receive TIEs on, that can be peer specific */
const UDPPortType     default_tie_udp_flood_port =  915
```

```
/** default MTU link size to use */
const MTUSizeType     default_mtu_size           = 1400
/** default link being BFD capable */
const bool            bfd_default                = true

/** type used to target nodes with key value */
typedef i64 KeyValueTargetType

/** default target for key value are all nodes. */
const KeyValueTargetType    keyvaluetarget_default = 0
/** value for _all leaves_ addressing. Represented by all bits set. */
const KeyValueTargetType    keyvaluetarget_all_south_leaves = -1

/** undefined nonce, equivalent to missing nonce */
const NonceType       undefined_nonce            = 0;
/** outer security Key ID, MUST be interpreted as in implementation
    as unsigned */
typedef i8            OuterSecurityKeyID
/** security Key ID, MUST be interpreted as in implementation
    as unsigned */
typedef i32           TIESecurityKeyID
/** undefined key */
const TIESecurityKeyID undefined_securitykey_id   = 0;
/** Maximum delta (negative or positive) that a mirrored nonce can
    deviate from local value to be considered valid. */
const i16                     maximum_valid_nonce_delta   = 5;
const TimeIntervalInSecType   nonce_regeneration_interval = 300;

/** Direction of TIEs. */
enum TieDirectionType {
    Illegal           = 0,
    South             = 1,
    North             = 2,
    DirectionMaxValue = 3,
}

/** Address family type. */
enum AddressFamilyType {
   Illegal                = 0,
   AddressFamilyMinValue  = 1,
   IPv4                   = 2,
   IPv6                   = 3,
   AddressFamilyMaxValue  = 4,
}

/** IPv4 prefix type. */
struct IPv4PrefixType {
    1: required IPv4Address    address;
```

```
    2: required PrefixLenType  prefixlen;
}

/** IPv6 prefix type. */
struct IPv6PrefixType {
    1: required IPv6Address    address;
    2: required PrefixLenType  prefixlen;
}

/** IP address type. */
union IPAddressType {
    /** Content is IPv4 */
    1: optional IPv4Address   ipv4address;
    /** Content is IPv6 */
    2: optional IPv6Address   ipv6address;
}

/** Prefix advertisement.

    @note: for interface
        addresses the protocol can propagate the address part beyond
        the subnet mask and on reachability computation that has to
        be normalized. The non-significant bits can be used
        for operational purposes.
*/
union IPPrefixType {
    1: optional IPv4PrefixType   ipv4prefix;
    2: optional IPv6PrefixType   ipv6prefix;
}

/** Sequence of a prefix in case of move.
 */
struct PrefixSequenceType {
    1: required IEEE802_1ASTimeStampType  timestamp;
    /** Transaction ID set by client in e.g. in 6LoWPAN. */
    2: optional PrefixTransactionIDType   transactionid;
}

/** Type of TIE.
*/
enum TIETypeType {
    Illegal                                     = 0,
    TIETypeMinValue                             = 1,
    /** first legal value */
    NodeTIEType                                 = 2,
    PrefixTIEType                               = 3,
    PositiveDisaggregationPrefixTIEType         = 4,
    NegativeDisaggregationPrefixTIEType         = 5,
```

```
    PGPrefixTIEType                              = 6,
    KeyValueTIEType                              = 7,
    ExternalPrefixTIEType                        = 8,
    PositiveExternalDisaggregationPrefixTIEType = 9,
    TIETypeMaxValue                              = 10,
}

/** RIFT route types.
    @note: The only purpose of those values is to introduce an
           ordering whereas an implementation can choose internally
           any other values as long the ordering is preserved
 */
enum RouteType {
    Illegal               =  0,
    RouteTypeMinValue     =  1,
    /** First legal value. */
    /** Discard routes are most preferred */
    Discard               =  2,

    /** Local prefixes are directly attached prefixes on the
     *  system such as e.g. interface routes.
     */
    LocalPrefix           =  3,
    /** Advertised in S-TIEs */
    SouthPGPPrefix        =  4,
    /** Advertised in N-TIEs */
    NorthPGPPrefix        =  5,
    /** Advertised in N-TIEs */
    NorthPrefix           =  6,
    /** Externally imported north */
    NorthExternalPrefix   =  7,
    /** Advertised in S-TIEs, either normal prefix or positive
        disaggregation */
    SouthPrefix           =  8,
    /** Externally imported south */
    SouthExternalPrefix   =  9,
    /** Negative, transitive prefixes are least preferred */
    NegativeSouthPrefix   = 10,
    RouteTypeMaxValue     = 11,
}

enum   KVTypes {
    Experimental = 1,
    WellKnown    = 2,
    OUI          = 3,
}
```

B.3. encoding.thrift

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

include "common.thrift"

namespace py encoding

/** Represents protocol encoding schema major version */
const common.VersionType protocol_major_version = 8
/** Represents protocol encoding schema minor version */
const common.MinorVersionType protocol_minor_version =  0

/** Common RIFT packet header. */
struct PacketHeader {
    /** Major version of protocol. */
    1: required common.VersionType      major_version =
            protocol_major_version;
    /** Minor version of protocol. */
    2: required common.MinorVersionType minor_version =
            protocol_minor_version;
    /** Node sending the packet, in case of LIE/TIRE/TIDE
        also the originator of it. */
    3: required common.SystemIDType  sender;
    /** Level of the node sending the packet, required on everything
        except LIEs. Lack of presence on LIEs indicates UNDEFINED_LEVEL
        and is used in ZTP procedures.
     */
    4: optional common.LevelType            level;
}

/** Prefix community. */
struct Community {
    /** Higher order bits */
    1: required i32          top;
    /** Lower order bits */
    2: required i32          bottom;
}

/** Neighbor structure.  */
struct Neighbor {
    /** System ID of the originator. */
    1: required common.SystemIDType        originator;
    /** ID of remote side of the link. */
    2: required common.LinkIDType          remote_id;
}
```

```
/** Capabilities the node supports. */
struct NodeCapabilities {
    /** Must advertise supported minor version dialect that way. */
    1: required common.MinorVersionType        protocol_minor_version =
            protocol_minor_version;
    /** indicates that node supports flood reduction. */
    2: optional bool                           flood_reduction =
            common.flood_reduction_default;
    /** indicates place in hierarchy, i.e. top-of-fabric or
        leaf only (in ZTP) or support for leaf-2-leaf
        procedures. */
    3: optional common.HierarchyIndications    hierarchy_indications;


}

/** Link capabilities. */
struct LinkCapabilities {
    /** Indicates that the link is supporting BFD. */
    1: optional bool                           bfd =
            common.bfd_default;
    /** Indicates whether the interface will support IPv4 forwarding. */
    2: optional bool                           ipv4_forwarding_capable =
            true;
}

/** RIFT LIE Packet.

    @note: this node's level is already included on the packet header
*/
struct LIEPacket {
    /** Node or adjacency name. */
    1: optional string                     name;
    /** Local link ID. */
    2: required common.LinkIDType          local_id;
    /** UDP port to which we can receive flooded TIEs. */
    3: required common.UDPPortType         flood_port =
            common.default_tie_udp_flood_port;
    /** Layer 3 MTU, used to discover mismatch. */
    4: optional common.MTUSizeType         link_mtu_size =
            common.default_mtu_size;
    /** Local link bandwidth on the interface. */
    5: optional common.BandwithInMegaBitsType
            link_bandwidth = common.default_bandwidth;
    /** Reflects the neighbor once received to provide
        3-way connectivity. */
    6: optional Neighbor                   neighbor;
    /** Node's PoD. */
```

```
    7: optional common.PodType            pod =
            common.default_pod;
    /** Node capabilities supported. */
   10: required NodeCapabilities          node_capabilities;
   /** Capabilities of this link. */
   11: optional LinkCapabilities          link_capabilities;
   /** Required holdtime of the adjacency, i.e. for how
       long a period should adjacency be kept up without valid LIE reception. */
   12: required common.TimeIntervalInSecType
            holdtime = common.default_lie_holdtime;
   /** Optional, unsolicited, downstream assigned locally significant label
       value for the adjacency. */
   13: optional common.LabelType          label;
    /** Indicates that the level on the LIE must not be used
        to derive a ZTP level by the receiving node. */
   21: optional bool                      not_a_ztp_offer =
            common.default_not_a_ztp_offer;
   /** Indicates to northbound neighbor that it should
       be reflooding TIEs received from this node to achieve flood
       reduction and balancing for northbound flooding. */
   22: optional bool                      you_are_flood_repeater =
             common.default_you_are_flood_repeater;
   /** Indicates to neighbor to flood node TIEs only and slow down
       all other TIEs. Ignored when received from southbound neighbor. */
   23: optional bool                      you_are_sending_too_quickly =
             false;
   /** Instance name in case multiple RIFT instances running on same
       interface. */
   24: optional string                    instance_name;
   /** It provides the optional ID of the Fabric configured. This MUST match the
information advertised
       on the node element. */
   35: optional common.FabricIDType       fabric_id = common.default_fabric_id;

}

/** LinkID pair describes one of parallel links between two nodes. */
struct LinkIDPair {
    /** Node-wide unique value for the local link. */
    1: required common.LinkIDType      local_id;
    /** Received remote link ID for this link. */
    2: required common.LinkIDType      remote_id;

    /** Describes the local interface index of the link. */
   10: optional common.PlatformInterfaceIndex platform_interface_index;
   /** Describes the local interface name. */
   11: optional string                        platform_interface_name;
   /** Indicates whether the link is secured, i.e. protected by
       outer key, absence of this element means no indication,
```

```
       undefined outer key means not secured. */
   12: optional common.OuterSecurityKeyID
               trusted_outer_security_key;
   /** Indicates whether the link is protected by established
       BFD session. */
   13: optional bool                          bfd_up;
   /** Optional indication which address families are up on the
       interface */
   14: optional set<common.AddressFamilyType>
                    address_families;
}

/** Unique ID of a TIE. */
struct TIEID {
    /** direction of TIE */
    1: required common.TieDirectionType    direction;
    /** indicates originator of the TIE */
    2: required common.SystemIDType        originator;
    /** type of the tie */
    3: required common.TIETypeType         tietype;
    /** number of the tie */
    4: required common.TIENrType           tie_nr;
}

/** Header of a TIE. */
struct TIEHeader {
    /** ID of the tie. */
    2: required TIEID                              tieid;
    /** Sequence number of the tie. */
    3: required common.SeqNrType                   seq_nr;

    /** Absolute timestamp when the TIE was generated. */
   10: optional common.IEEE802_1ASTimeStampType   origination_time;
   /** Original lifetime when the TIE was generated.  */
   12: optional common.LifeTimeInSecType          origination_lifetime;
}

/** Header of a TIE as described in TIRE/TIDE.
*/
struct TIEHeaderWithLifeTime {
    1: required     TIEHeader                       header;
    /** Remaining lifetime. */
    2: required     common.LifeTimeInSecType        remaining_lifetime;
}

/** TIDE with *sorted* TIE headers. */
struct TIDEPacket {
    /** First TIE header in the tide packet. */
```

```
    1: required TIEID                       start_range;
    /** Last TIE header in the tide packet. */
    2: required TIEID                       end_range;
    /** _Sorted_ list of headers. */
    3: required list<TIEHeaderWithLifeTime>
                    headers;
}

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

/** neighbor of a node */
struct NodeNeighborsTIEElement {
    /** level of neighbor */
    1: required common.LevelType                level;
    /**  Cost to neighbor. Ignore anything larger than `infinite_distance` and `i
nvalid_distance` */
    3: optional common.MetricType               cost
               = common.default_distance;
    /** can carry description of multiple parallel links in a TIE */
    4: optional set<LinkIDPair>
                        link_ids;
    /** total bandwith to neighbor as sum of all parallel links */
    5: optional common.BandwithInMegaBitsType
               bandwidth = common.default_bandwidth;
}

/** Indication flags of the node. */
struct NodeFlags {
    /** Indicates that node is in overload, do not transit traffic
        through it. */
     1: optional bool         overload = common.overload_default;
}

/** Description of a node. */
struct NodeTIEElement {
    /** Level of the node. */
    1: required common.LevelType            level;
    /** Node's neighbors. Multiple node TIEs can carry disjoint sets of neighbors
. */
    2: required map<common.SystemIDType,
               NodeNeighborsTIEElement>    neighbors;
    /** Capabilities of the node. */
    3: required NodeCapabilities            capabilities;
    /** Flags of the node. */
    4: optional NodeFlags                   flags;
    /** Optional node name for easier operations. */
```

```
    5: optional string                      name;
    /** PoD to which the node belongs. */
    6: optional common.PodType              pod;
    /** optional startup time of the node */
    7: optional common.TimestampInSecsType  startup_time;

    /** If any local links are miscabled, this indication is flooded. */
   10: optional set<common.LinkIDType>
                    miscabled_links;

   /** ToFs in the same plane. Only carried by ToF. Multiple Node TIEs can carry
disjoint sets of ToFs
       which MUST be joined to form a single set. */
   12: optional set<common.SystemIDType>
                    same_plane_tofs;

   /** It provides the optional ID of the Fabric configured */
   20: optional common.FabricIDType             fabric_id = common.default_fabric
_id;


}

/** Attributes of a prefix. */
struct PrefixAttributes {
    /** Distance of the prefix. */
    2: required common.MetricType           metric
            = common.default_distance;
    /** Generic unordered set of route tags, can be redistributed
        to other protocols or use within the context of real time
        analytics. */
    3: optional set<common.RouteTagType>
                     tags;
    /** Monotonic clock for mobile addresses.  */
    4: optional common.PrefixSequenceType    monotonic_clock;
    /** Indicates if the prefix is a node loopback. */
    6: optional bool                         loopback = false;
    /** Indicates that the prefix is directly attached. */
    7: optional bool                         directly_attached = true;
    /** link to which the address belongs to.  */
   10: optional common.LinkIDType            from_link;
    /** Optional, per prefix significant label. */
   12: optional common.LabelType             label;
}

/** TIE carrying prefixes */
struct PrefixTIEElement {
    /** Prefixes with the associated attributes. */
    1: required map<common.IPPrefixType, PrefixAttributes> prefixes;
}
```

```
/** Defines the targeted nodes and the value carried. */
struct KeyValueTIEElementContent {
    1: optional common.KeyValueTargetType        targets = common.keyvaluetarget_
default;
    2: optional binary                           value;
}

/** Generic key value pairs. */
struct KeyValueTIEElement {
    1: required map<common.KeyIDType, KeyValueTIEElementContent>    keyvalues;
}

/** Single element in a TIE. */
union TIEElement {
    /** Used in case of enum common.TIETypeType.NodeTIEType. */
    1: optional NodeTIEElement     node;
    /** Used in case of enum common.TIETypeType.PrefixTIEType. */
    2: optional PrefixTIEElement          prefixes;
    /** Positive prefixes (always southbound). */
    3: optional PrefixTIEElement   positive_disaggregation_prefixes;
    /** Transitive, negative prefixes (always southbound) */
    5: optional PrefixTIEElement   negative_disaggregation_prefixes;
    /** Externally reimported prefixes. */
    6: optional PrefixTIEElement          external_prefixes;
    /** Positive external disaggregated prefixes (always southbound). */
    7: optional PrefixTIEElement
            positive_external_disaggregation_prefixes;
    /** Key-Value store elements. */
    9: optional KeyValueTIEElement keyvalues;
}

/** TIE packet */
struct TIEPacket {
    1: required TIEHeader  header;
    2: required TIEElement element;
}

/** Content of a RIFT packet. */
union PacketContent {
    1: optional LIEPacket     lie;
    2: optional TIDEPacket    tide;
    3: optional TIREPacket    tire;
    4: optional TIEPacket     tie;
}

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

}

Authors' Addresses

Tony Przygienda (editor)
Juniper Networks
1137 Innovation Way
Sunnyvale, CA 94089
United States of America
Email: prz@juniper.net

Jordan Head (editor)
Juniper Networks
1137 Innovation Way
Sunnyvale, CA 94089
United States of America
Email: jhead@juniper.net

Alankar Sharma
Hudson River Trading
United States of America
Email: as3957@gmail.com

Pascal Thubert
Individual
France
Email: pascal.thubert@gmail.com

Bruno Rijsman
Individual
Email: brunorijsman@gmail.com

Dmitry Afanasiev
Yandex
Email: fl0w@yandex-team.ru

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

Table of Contents

1. Introduction

[I-D.ietf-rift-rift] introduces the protocol definition of RIFT. This document defines a YANG data model that can be used to configure and manage the RIFT protocol. The model is based on YANG 1.1 as defined in [RFC7950] and conforms to the Network Management Datastore Architecture (NDMA) as described in [RFC8342]

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 (algorighm-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
6
        |  |        +--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 elem
ent.";
            }
            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:rift')" {
            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.
                                 When the type of kv is set to 'node', node-elem
ent is
                                 making sense. When the type of kv is set to oth
er values
                                 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