RIFT Working Group
Internet-Draft
Intended status: Standards Track
Expires: March 11, 2020

A. Przygienda, Ed.
Juniper
A. Sharma
Comcast
P. Thubert
Cisco
Bruno. Rijsman
Individual
Dmitry. Afanasiev
Yandex

September 8, 2019

RIFT: Routing in Fat Trees
draft-ietf-rift-rift-08

Abstract

This document outlines a specialized, dynamic routing protocol for Clos and fat-tree network topologies. The protocol (1) deals with fully automated construction of fat-tree topologies based on detection of links, (2) minimizes the amount of routing state held at each level, (3) automatically prunes and load balances topology flooding exchanges over a sufficient subset of links, (4) supports automatic disaggregation of prefixes on link and node failures to prevent black-holing and suboptimal routing, (5) allows traffic steering and re-routing policies, (6) allows loop-free non-ECMP forwarding, (7) automatically re-balances traffic towards the spines based on bandwidth available and finally (8) provides mechanisms to synchronize a limited key-value data-store that can be used after protocol convergence to e.g. bootstrap higher levels of functionality on nodes.

Status of This Memo

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

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

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

This Internet-Draft will expire on March 11, 2020.

Table of Contents

1. Authors

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

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

Table 1: RIFT Authors

2. Introduction

Clos [CLOS] and Fat-Tree [FATTREE] topologies have gained prominence in today's networking, primarily as result of the paradigm shift towards a centralized data-center based architecture that is poised to deliver a majority of computation and storage services in the future. Today's current routing protocols were geared towards a network with an irregular topology and low degree of connectivity originally but given they were the only available options, consequently several attempts to apply those protocols to Clos have been made. Most successfully BGP [RFC4271] [RFC7938] has been

extended to this purpose, not as much due to its inherent suitability but rather because the perceived capability to easily modify BGP and the immanent difficulties with link-state [DIJKSTRA] based protocols to optimize topology exchange and converge quickly in large scale densely meshed topologies. The incumbent protocols precondition normally extensive configuration or provisioning during bring up and re-dimensioning which is only viable for a set of organizations with according networking operation skills and budgets. For the majority of data center consumers a preferable protocol would be one that auto-configures itself and deals with failures and misconfigurations with a minimum of human intervention only. Such a solution would allow local IP fabric bandwidth to be consumed in a 'standard component' fashion, i.e. provision it much faster and operate it at much lower costs, much like compute or storage is consumed today.

In looking at the problem through the lens of data center requirements, an optimal approach does not seem however to be a simple modification of either a link-state (distributed computation) or distance-vector (diffused computation) approach but rather a mixture of both, colloquially best described as "link-state towards the spine" and "distance vector towards the leafs". In other words, "bottom" levels are flooding their link-state information in the "northern" direction while each node generates under normal conditions a default route and floods it in the "southern" direction. This type of protocol allows naturally for highly desirable aggregation. Alas, such aggregation could blackhole traffic in cases of misconfiguration or while failures are being resolved or even cause partial network partitioning and this has to be addressed. The approach RIFT takes is described in Section 5.2.5 and is basically based on automatic, sufficient disaggregation of prefixes.

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

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

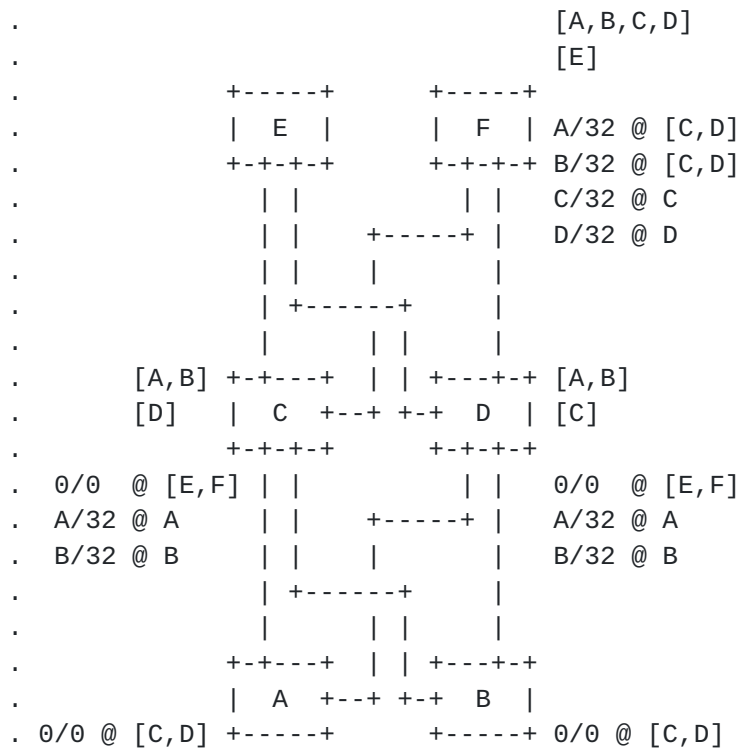

Figure 1: RIFT information distribution

2.1. Requirements Language

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

3. Reference Frame

3.1. Terminology

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

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

Clos/Fat Tree: This document uses the terms Clos and Fat Tree interchangeably whereas it always refers to a folded spine-and-leaf topology with possibly multiple PoDs and one or multiple ToF

planes. Several modifications such as leaf-2-leaf shortcuts and multiple level shortcuts are possible and described further in the document.

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

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

Superspine/Aggregation or Spine/Edge Levels: Traditional names in 5-stages folded Clos for Level 2, 1 and 0 respectively. Level 0 is often called leaf as well. We normalize this language to talk about leafs, spines and top-of-fabric (ToF).

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

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

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

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

Leaf: A node without southbound adjacencies. Its level is 0 (except cases where it is deriving its level via ZTP and is running without LEAF_ONLY which will be explained in Section 5.2.7).

Top-of-fabric Plane or Partition: In large fabrics top-of-fabric switches may not have enough ports to aggregate all switches south of them and with that, the ToF is 'split' into multiple

independent planes. Introduction and Section 5.1.2 explains the concept in more detail. A plane is subset of ToF nodes that see each other through south reflection or E-W links.

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

North Radix: Ports cabled northbound to higher level nodes.

South Radix: Ports cabled southbound to lower level nodes.

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

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

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

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

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

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

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

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

TIE: This is an acronym for a "Topology Information Element". TIEs are exchanged between RIFT nodes to describe parts of a network such as links and address prefixes, in a fashion similar to ISIS LSPs or OSPF LSAs. We will talk about N-TIEs when talking about TIEs in the northbound representation and S-TIEs for the southbound equivalent.

Node TIE: This stands as acronym for a "Node Topology Information Element" that contains all adjacencies the node discovered and information about node itself.

Prefix TIE: This is an acronym for a "Prefix Topology Information Element" and it contains all prefixes directly attached to this node in case of a N-TIE and in case of S-TIE the necessary default the node passes southbound.

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

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

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

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

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

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

Bandwidth Adjusted Distance (BAD): This is an acronym for Bandwidth Adjusted Distance. Each RIFT node calculates the amount of northbound bandwidth available towards a node compared to other nodes at the same level and modifies the default route distance accordingly to allow for the lower level to adjust their load balancing towards spines.

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

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

Adjacency: RIFT tries to form a unique adjacency over an interface and exchange local configuration and necessary ZTP information.

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

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

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

Metric: Without going deeper into the proper differentiation, a metric is equivalent to distance.

3.2. Topology

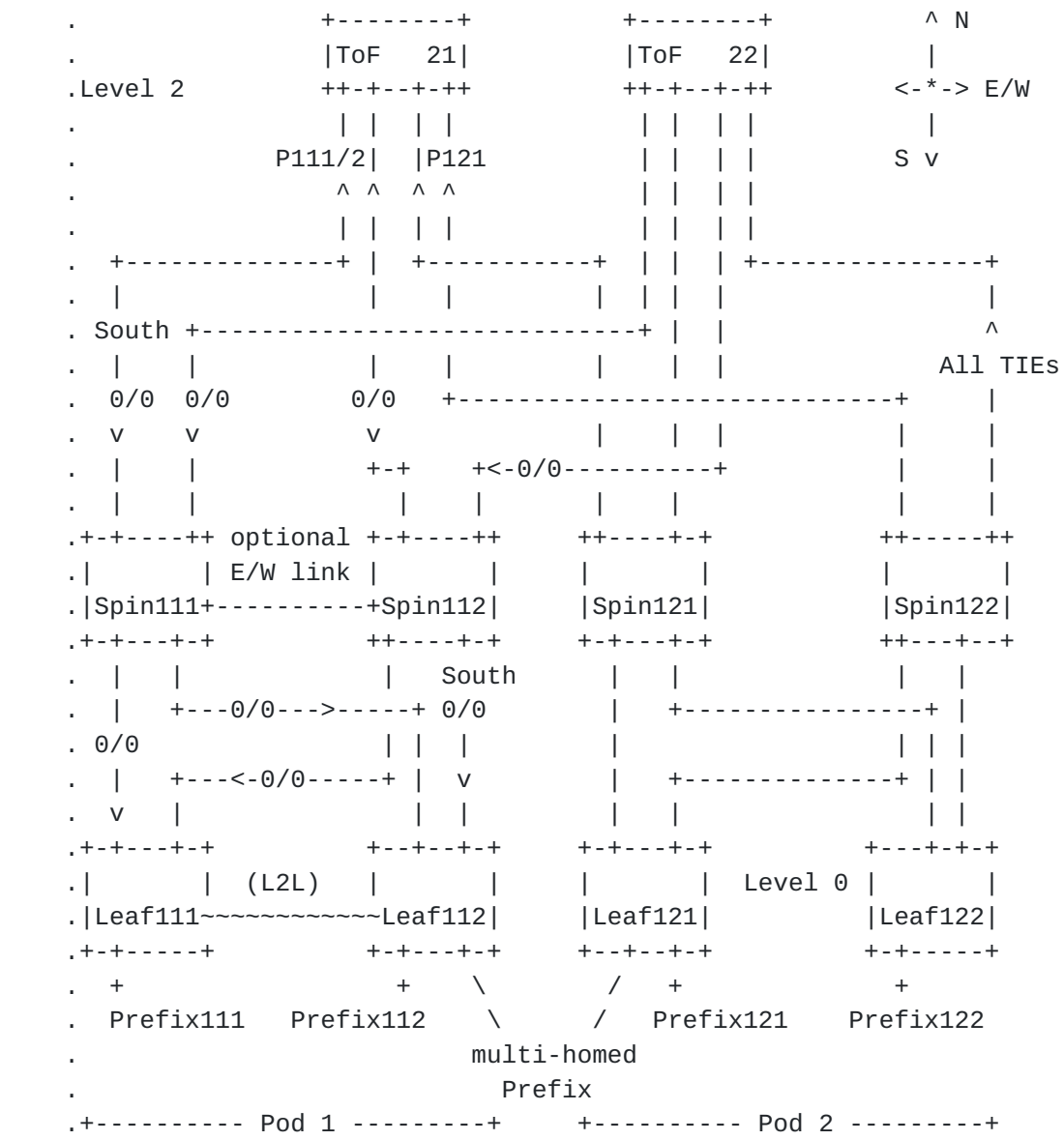

Figure 2: A three level spine-and-leaf topology

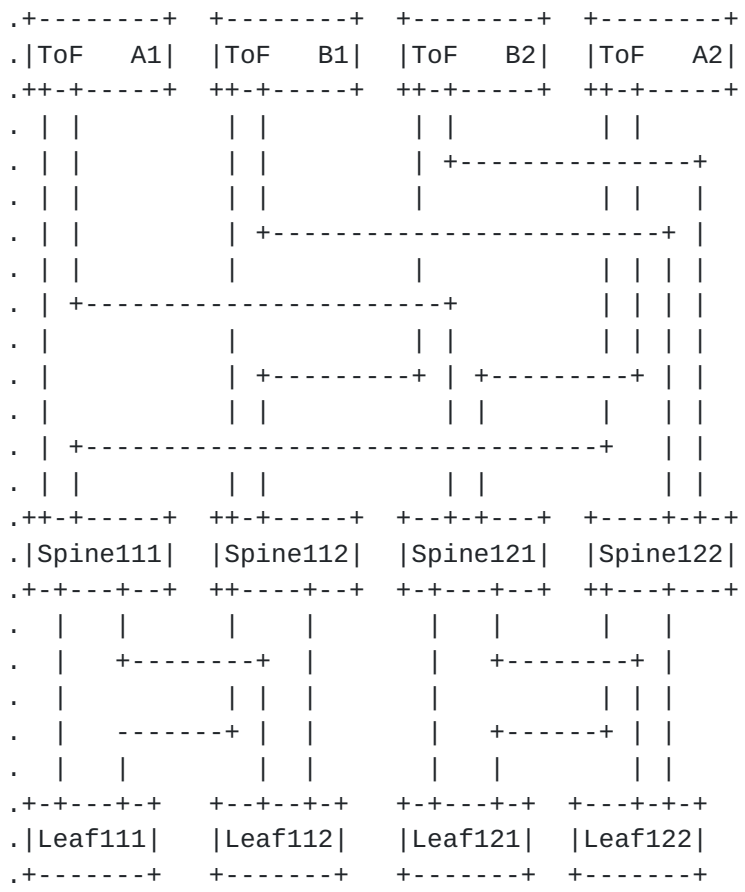

Figure 3: Topology with multiple planes

We will use topology in Figure 2 (called commonly a fat tree/network in modern IP fabric considerations [VAHDAT08] as homonym to the original definition of the term [FATTREE]) in all further considerations. This figure depicts a generic "single plane fat-tree" and the concepts explained using three levels apply by induction to further levels and higher degrees of connectivity. Further, this document will deal also with designs that provide only sparser connectivity and "partitioned spines" as shown in Figure 3 and explained further in Section 5.1.2.

4. Requirement Considerations

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

REQ1: The control protocol should discover the physical links automatically and be able to detect cabling that violates fat-tree topology constraints. It must react accordingly to such mis-cabling attempts, at a minimum preventing

adjacencies between nodes from being formed and traffic from being forwarded on those mis-cabled links. E.g. connecting a leaf to a spine at level 2 should be detected and ideally prevented.

REQ2: A node without any configuration beside default values should come up at the correct level in any PoD it is introduced into. Optionally, it must be possible to configure nodes to restrict their participation to the PoD(s) targeted at any level.

REQ3: Optionally, the protocol should allow to provision IP fabrics where the individual switches carry no configuration information and are all deriving their level from a "seed". Observe that this requirement may collide with the desire to detect cabling misconfiguration and with that only one of the requirements can be fully met in a chosen configuration mode.

REQ4: The solution should allow for minimum size routing information base and forwarding tables at leaf level for speed, cost and simplicity reasons. Holding excessive amount of information away from leaf nodes simplifies operation and lowers cost of the underlay and allows to scale and introduce proper multi-homing down to the server level. The routing solution should allow for easy instantiation of multiple routing planes. Coupled with mobility defined in Paragraph 17 this should allow for "light-weight" overlays on an IP fabric with e.g. native IPv6 mobility support.

REQ5: Very high degree of ECMP must be supported. Maximum ECMP is currently understood as the most efficient routing approach to maximize the throughput of switching fabrics [MAKSIC2013].

REQ6: Non equal cost anycast must be supported to allow for easy and robust multi-homing of services without regressing to careful balancing of link costs.

REQ7: Traffic engineering should be allowed by modification of prefixes and/or their next-hops.

REQ8: The solution should allow for access to link states of the whole topology to enable efficient support for modern control architectures like SPRING [RFC7855] or PCE [RFC4655].

REQ9: The solution should easily accommodate opaque data to be carried throughout the topology to subsets of nodes. This can be used for many purposes, one of them being a key-value store that allows bootstrapping of nodes based right at the time of topology discovery. Another use is distributing MAC to L3 address binding from the leafs up north in case of e.g. DHCP.

REQ10: Nodes should be taken out and introduced into production with minimum wait-times and minimum of "shaking" of the network, i.e. radius of propagation (often called "blast radius") of changed information should be as small as feasible.

REQ11: The protocol should allow for maximum aggregation of carried routing information while at the same time automatically de-aggregating the prefixes to prevent black-holing in case of failures. The de-aggregation should support maximum possible ECMP/N-ECMP remaining after failure.

REQ12: Reducing the scope of communication needed throughout the network on link and state failure, as well as reducing advertisements of repeating or idiomatic information in stable state is highly desirable since it leads to better stability and faster convergence behavior.

REQ13: Under normal, fully converged condition, once a packet is forwarded along a link in a "southbound" direction, it must not take any further "northbound" links (Valley Free Routing). Taking a path through the spine in cases where a shorter path is available is highly undesirable (Bow Tying).

REQ14: Parallel links between same set of nodes must be distinguishable for SPF, failure and traffic engineering purposes.

REQ15: The protocol must support interfaces sharing the same address. Specifically, it must operate in presence of unnumbered links (even parallel ones) and/or links of a single node being configured with same addresses.

REQ16: It would be desirable to achieve fast re-balancing of flows when links, especially towards the spines are lost or provisioned without regressing to per flow traffic engineering which introduces significant amount of complexity while possibly not being reactive enough to account for short-lived flows.

REQ17: The control plane should be able to unambiguously determine the current point of attachment (which port on which leaf node) of a prefix, even in a context of fast mobility, e.g., when the prefix is a host address on a wireless node that 1) may associate to any of multiple access points (APs) that are attached to different ports on a same leaf node or to different leaf nodes, and 2) may move and reassociate several times to a different access point within a sub-second period.

REQ18: The protocol must provide security mechanisms that allow the operator to restrict nodes, especially leaf nodes without proper credentials, from forming a three-way adjacency and participating in routing.

Following list represents non-requirements:

PEND1: Supporting anything but point-to-point links is not necessary.

Finally, following are the non-requirements:

NONREQ1: Broadcast media support is unnecessary. However, miscabling leading to multiple nodes on a broadcast segment must be operationally easily recognizable and detectable while not taxing the protocol excessively.

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

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

5. RIFT: Routing in Fat Trees

Derived from the above requirements we present a detailed outline of a protocol optimized for Routing in Fat Trees (RIFT) that in most abstract terms has many properties of a modified link-state protocol [RFC2328][ISO10589-Second-Edition] when "pointing north" and distance vector [RFC4271] protocol when "pointing south". While this is an unusual combination, it does quite naturally exhibit the desirable properties we seek.

5.1. Overview

5.1.1. Properties

The most singular property of RIFT is that it floods flat link-state information northbound only so that each level obtains the full topology of levels south of it. That information is never flooded East-West (we'll talk about exceptions later) or back South again. In the southbound direction the protocol operates like a "fully summarizing, unidirectional" path vector protocol or rather a distance vector with implicit split horizon whereas the information propagates one hop south and is 're-advertised' by nodes at next lower level, normally just the default route. However, RIFT uses flooding in the southern direction as well to avoid the necessity to build an update per adjacency. We omit describing the East-West direction out for the moment.

Those information flow constraints create not only an anisotropic protocol (i.e. the information is not distributed "evenly" or "clumped" but summarized along the N-S gradient) but also a "smooth" information propagation where nodes do not receive the same information from multiple directions at the same time. Normally, accepting the same reachability on any link without understanding its topological significance forces tie-breaking on some kind of distance metric and ultimately leads in hop-by-hop forwarding substrates to utilization of variants of shortest paths only. RIFT under normal conditions does not need to reconcile same reachability information from multiple directions and its computation principles (south forwarding direction is always prefered) leads to valley-free forwarding behavior. And since valley free routing is loop-free it can use all feasible paths, 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 accordingly partitioned into "north representation" and "south representation" TIEs. In simplest terms the N-TIEs contain a link state topology description of lower levels and and S-TIEs carry simply default routes of the level above. This oversimplified view will be refined gradually in following sections while introducing protocol procedures aimed to fulfill the described requirements.

5.1.2. Generalized Topology View

This section will shed some light on the topologies addresses by RIFT including multi plane fabrics and their related implications. Readers that are only interested in single plane designs, i.e. all

top-of-fabric nodes being topologically equal and initially connected to all the switches at the level below them can skip this section and resulting Section 5.2.5.2 as well.

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

As 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 introduced Section 3.1 and it is paramount to have it present to follow the rest of this section correctly.

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

Figure 4 provides a view of a leaf node as seen from the north, i.e. showing ports that connect northbound and for lack of a better symbol, we have chosen to use the "HH" symbol as ASCII visualisation of a RJ45 jack. In that example, K_LEAF is chosen to be 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.

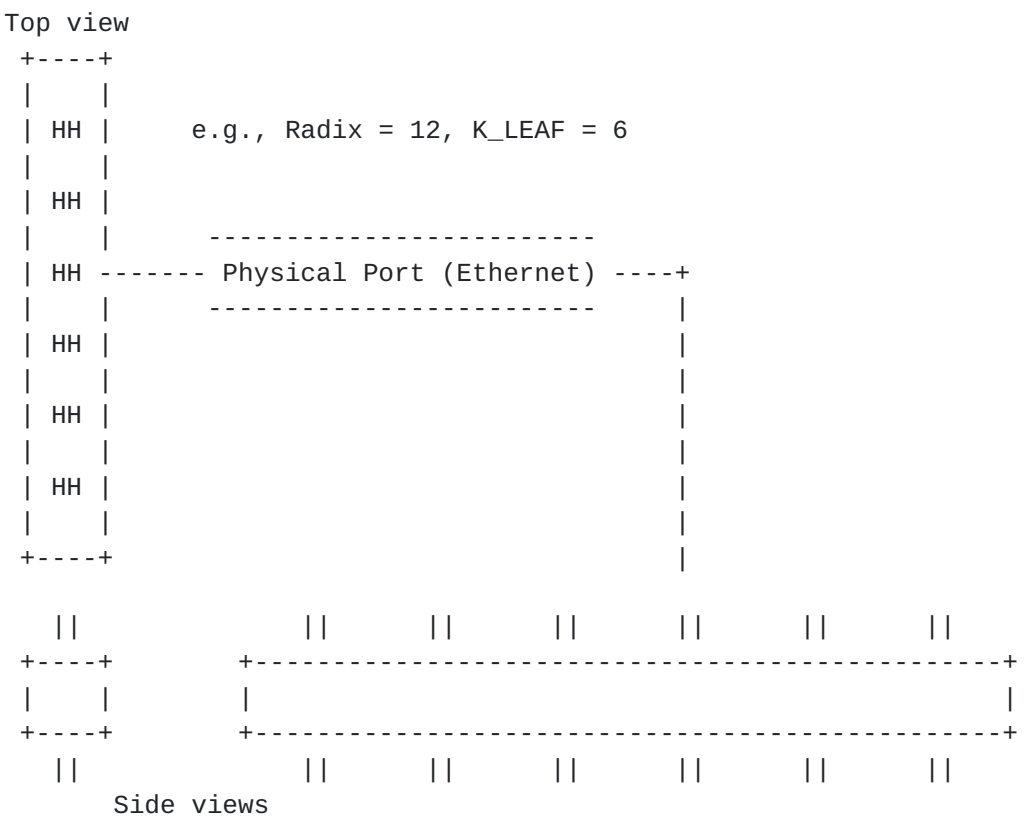

Figure 4: A Leaf Node, K_LEAF=6

The Radix of a node on top of a PoD 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, we could have switches with K_TOP southern ports on nodes at the top of the PoD that is not necessarily the same as K_LEAF; for instance, in the representations below, we pick a K_LEAF of 6 and a K_TOP of 8. In order to form a crossbar, we need K_TOP Leaf Nodes as illustrated in Figure 5.

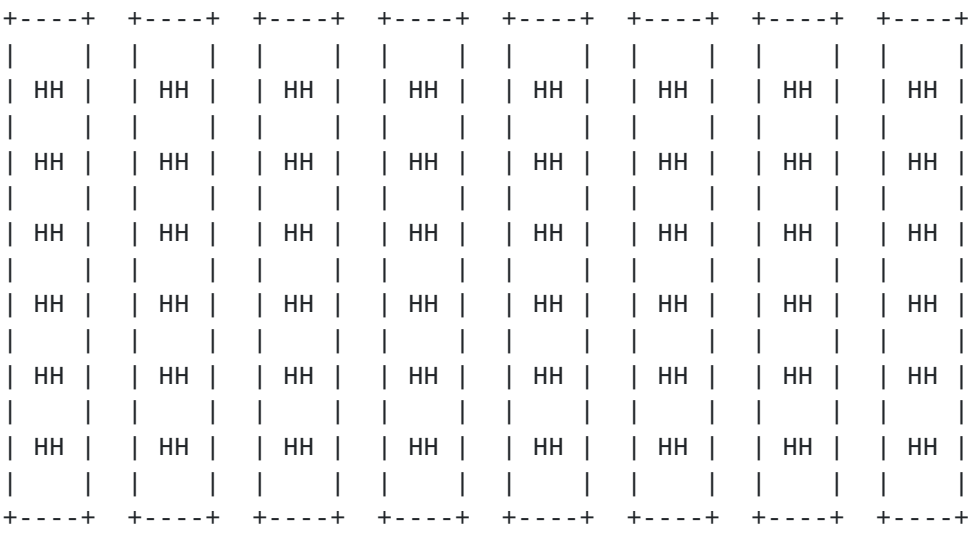

Figure 5: Southern View of a PoD, K_TOP=8

The K_TOP Leaf Nodes are fully interconnected with the K_LEAF PoD-top nodes, providing a connectivity that can be represented as a crossbar as seen from the north and illustrated in Figure 6. The result is that, in the absence of a breakage, a packet entering the PoD from North on any port can be routed to any port on the south of the PoD and vice versa.

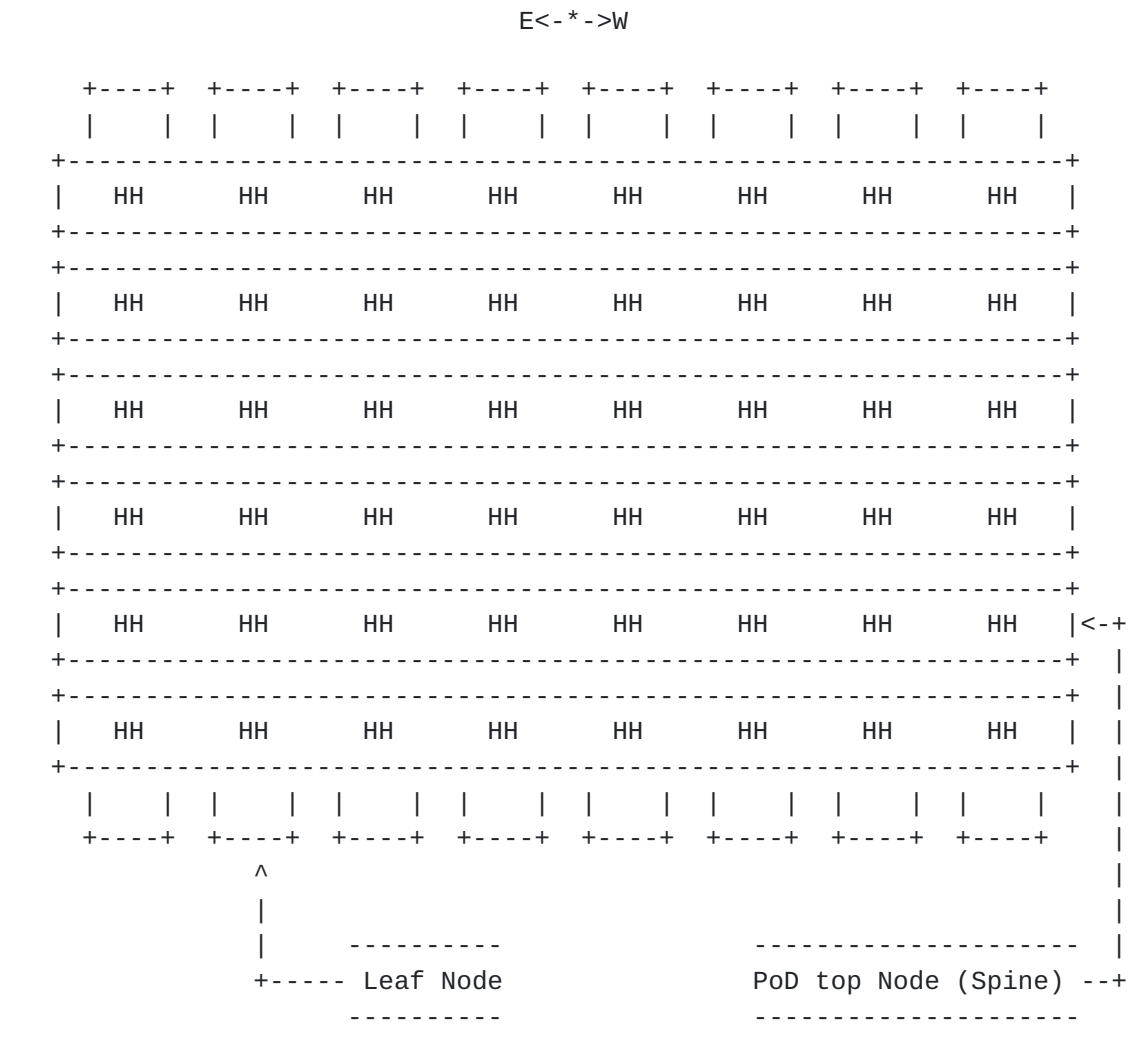

Figure 6: Northern View of a PoD's Spines, K_TOP=8

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

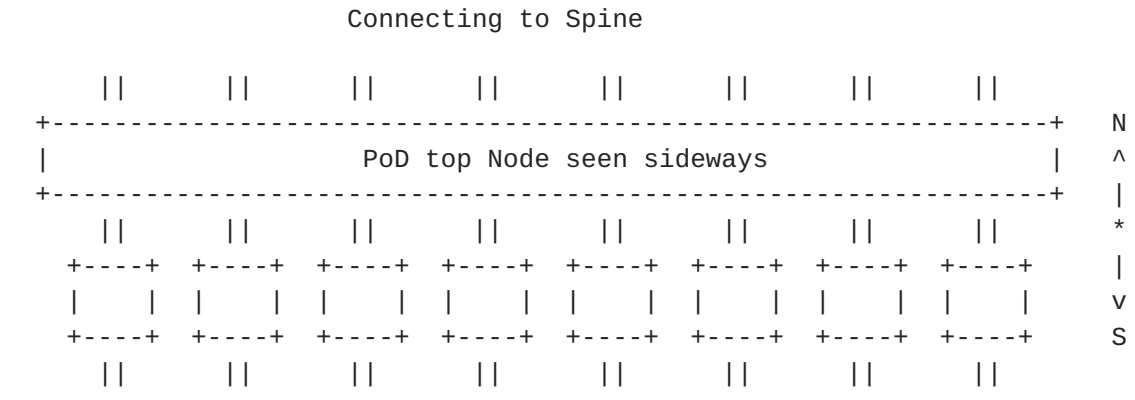

Figure 7: Side View of a PoD, K_TOP=8, K_LEAF=6

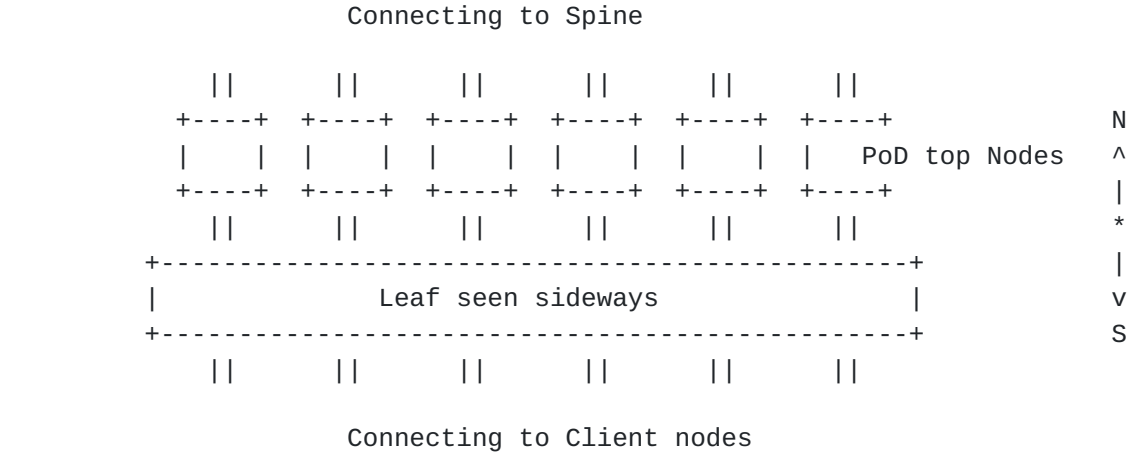

Figure 8: Other side View of a PoD, K_TOP=8, K_LEAF=6, 90o turn in E-W Plane

Note 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 is critical at this junction that the concept and the picture of those "crossed crossbars" is clear before progressing further, otherwise following considerations will be difficult to comprehend.

Further, the PoDs are interconnected with one another through a Top-of-Fabric 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 is also referred to as a single plane configuration. 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 takes (P * K_LEAF) ports on a ToF node to connect to each of the K_LEAF ToP nodes of the P PoDs, as illustrated in Figure 9.

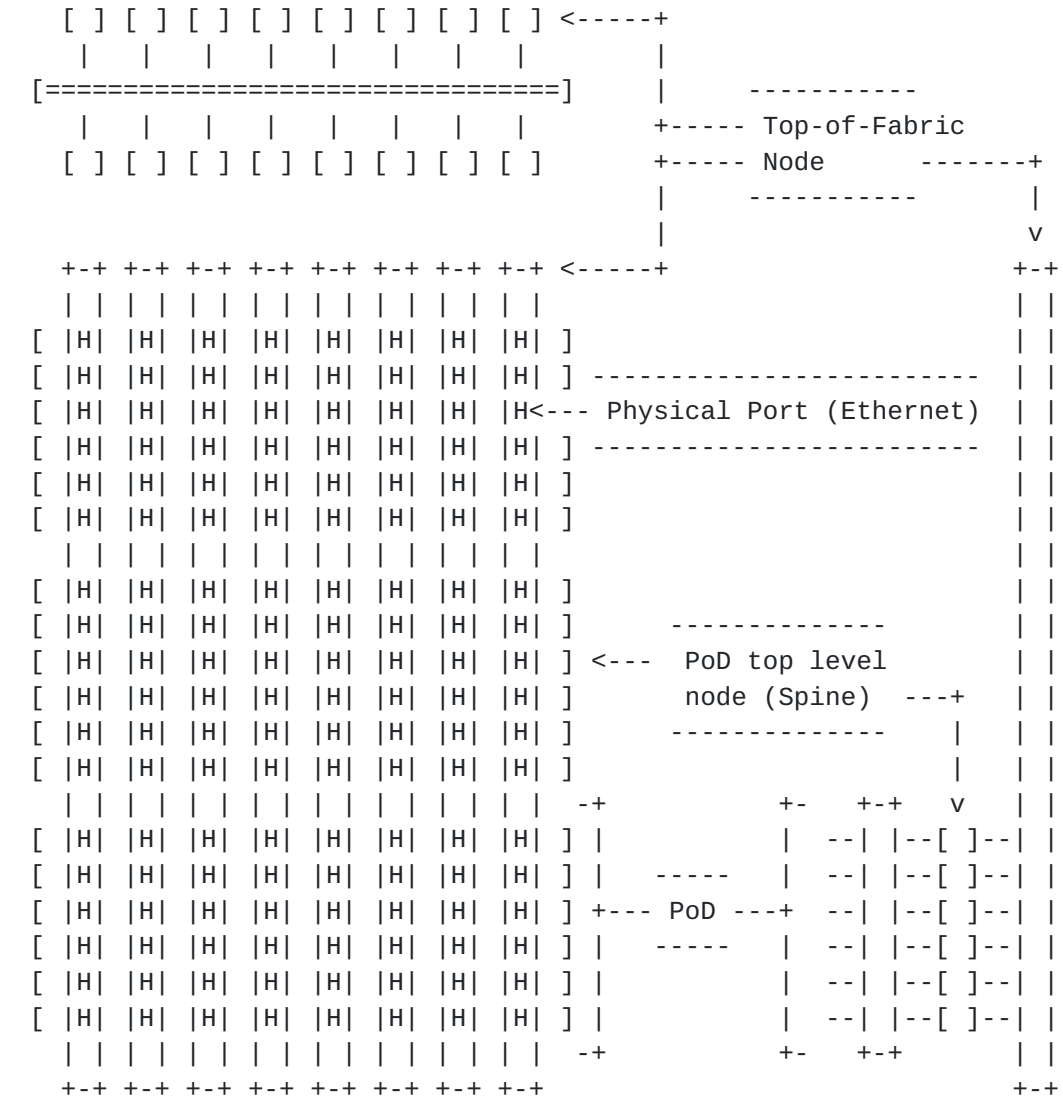

Figure 9: Fabric Spines and TOFs in Single Plane Design, 3 PoDs

The top view can be collapsed into a third dimension where the hidden depth index is representing the PoD number. So we can show one PoD as a class of PoDs and hence save one dimension in our representation. The Spine Node expands in the depth and the vertical dimensions whereas the PoD top level Nodes are constrained in

horizontal dimension. A port in the 2-D representation represents effectively the class of all the ports at the same position in all the PoDs that are projected in its position along the depth axis. This is shown in Figure 10.

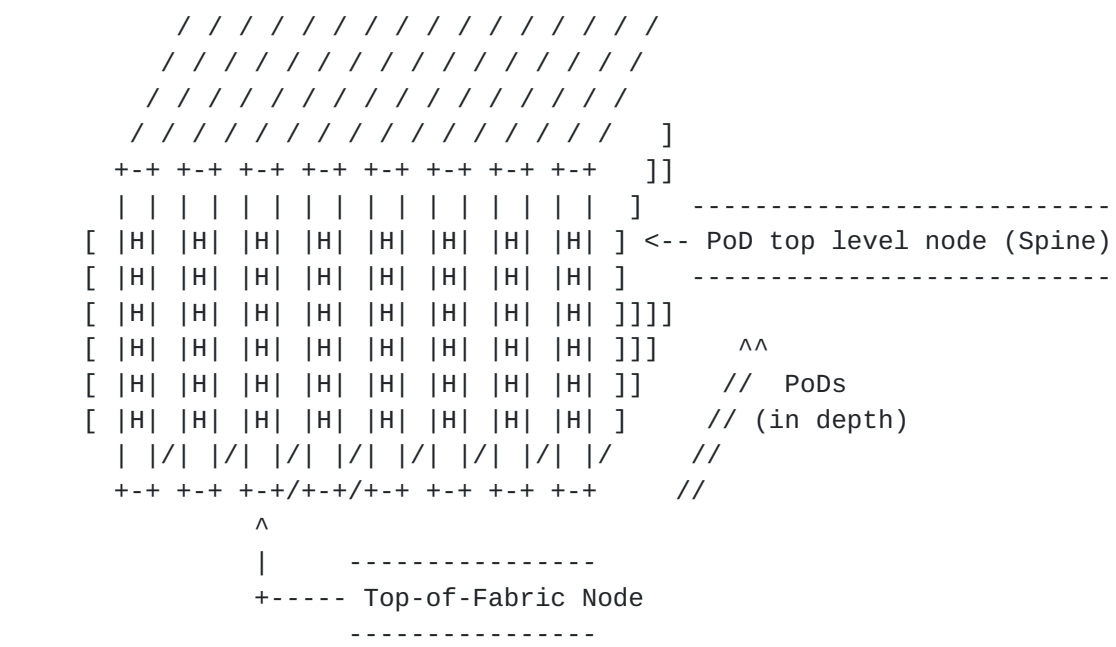

Figure 10: Collapsed Northern View of a Fabric for Any Number of PoDs

This type of deployment introduces a "single plane limit" where the bound is the available radix of the ToF nodes, which limits (P * K_LEAF). Nevertheless, a distinct advantage of a connected or unpartitioned Top-of-Fabric is that all failures can be resolved by simple, non-transitive, positive disaggregation described in Section 5.2.5.1 that propagates only within one level of the fabric. In other words unpartitoned ToF nodes can always reach nodes below or withdraw the routes from PoDs they cannot reach unambiguously. To be more precise, all failures which still allow all the ToF nodes to see each other via south reflection as explained in Section 5.2.5.

In order to scale beyond the "single plane limit", the Top-of-Fabric can be partitioned by a number N of identically wired planes, N being 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 divisor of K_LEAF, then (N=K_LEAF/R) is a feasible number of planes and R a redundancy factor. If proves convenient for deployments to use a radix for the leaf nodes that is a power of 2 so they can pick a number of planes that is a lower power of 2. The example in Figure 11 splits the Spine in 2 planes with a redundancy factor R=3,

meaning that there are 3 non-intersecting paths between any leaf node and any ToF node. A ToF node must have in this case at least 3*P ports, and be directly connected to 3 of the 6 PoD-ToP nodes (spines) in each PoD.

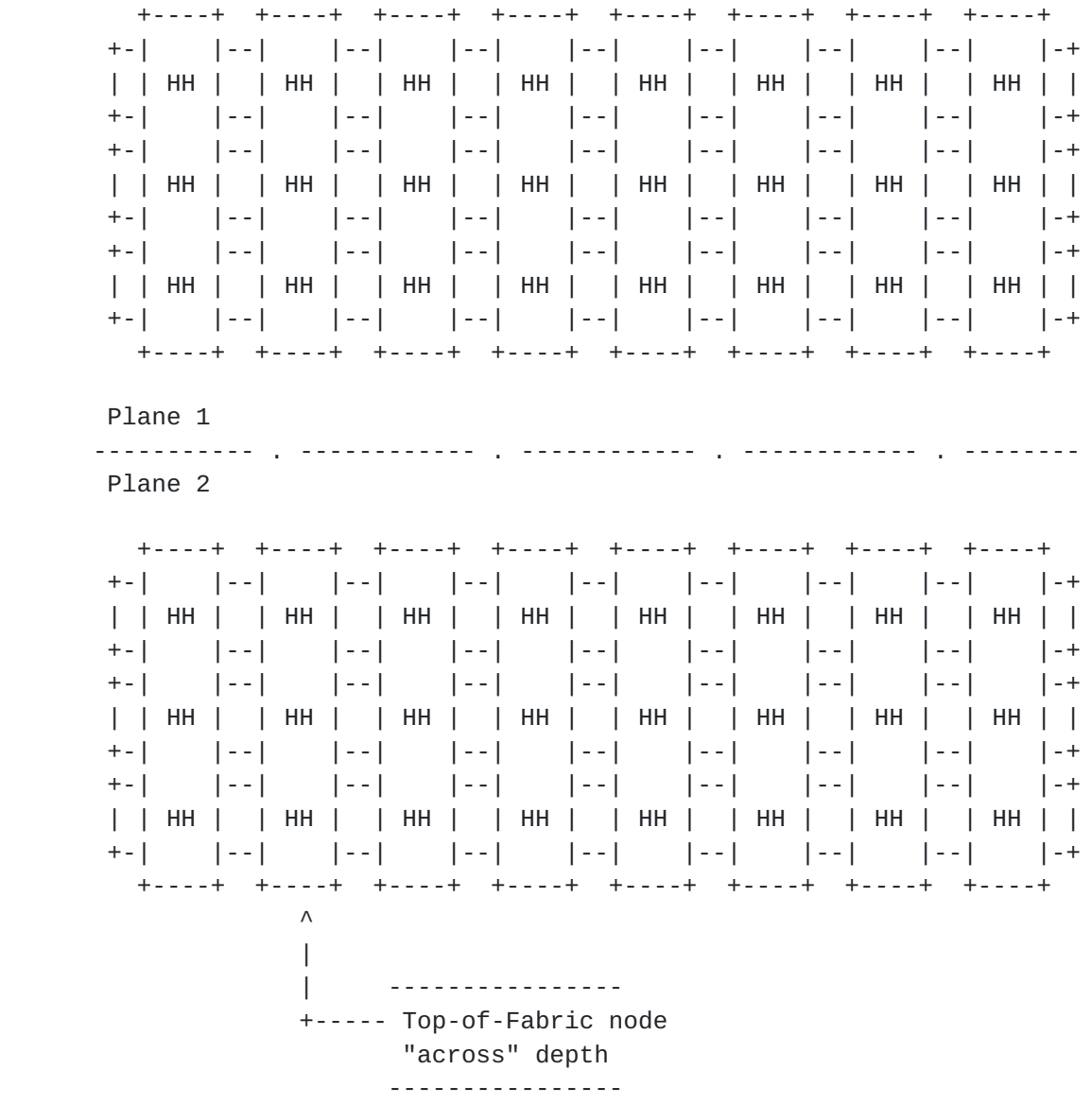

Figure 11: Northern View of a Multi-Plane ToF Level, K_LEAF=6, N=2

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

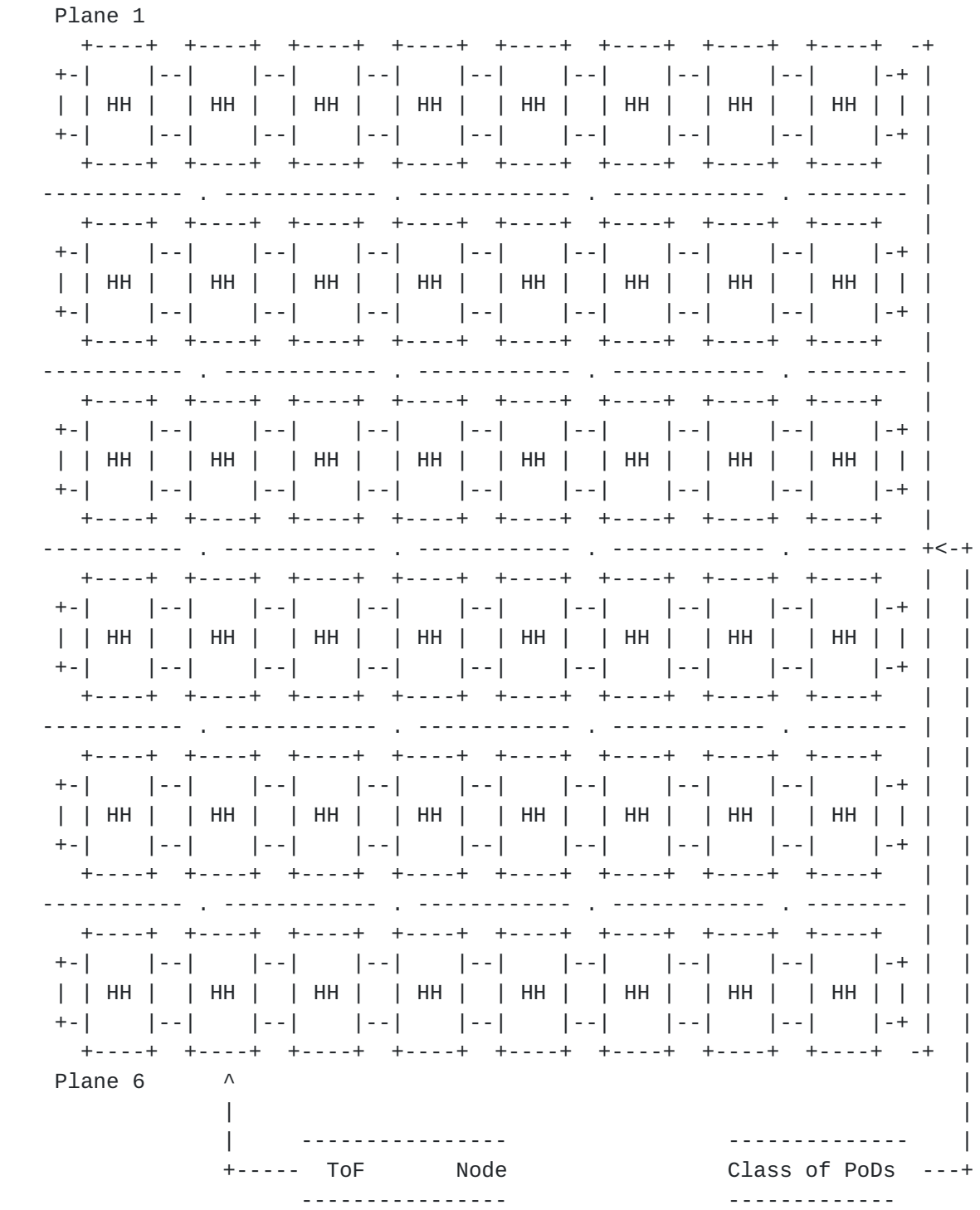

Figure 12: Northern View of a Maximally Partitioned ToF Level, R=1

5.1.3. Fallen Leaf Problem

As mentioned earlier, RIFT exhibits an anisotropic behavior tailored for fabrics with a North / South orientation and a high level of interleaving paths. A non-partitioned fabric makes a total loss of connectivity between a Top-of-Fabric node at the north and a leaf node at the south a very rare but yet possible occasion that is fully healed by positive disaggregation described in Section 5.2.5.1. In large fabrics or fabrics built from switches with low radix, the ToF ends often being partioned in planes which makes the occurrence of having a given leaf being only reachable from a subset of the ToF nodes more likely to happen. This makes some further considerations necessary.

We define a "Fallen Leaf" as a leaf that can be reached by only a subset of Top-of-Fabric nodes but cannot be reached by all 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 general manner, the mechanism of non-transitive positive disaggregation is sufficient when the disaggregating ToF nodes collectively connect to all the ToP nodes in the broken plane. This happens in the following case:

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

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

- If the breakage is the last northern link from a Leaf node within a plane - there is only one such link in a maximally partitioned fabric - that goes down, then connectivity to all unicast prefixes attached to the Leaf node is lost within the plane where the link is located. Southern Reflection by a Leaf Node - e.g., between ToP nodes if the PoD has only 2 levels - happens in between planes, allowing the ToP nodes to detect the problem within the PoD where it occurs and positively disaggregate. The breakage can be observed by the ToF nodes in the same plane through the flooding of N-TIEs from the ToP nodes, but the ToF nodes 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 N-TIEs from the affected Leaf nodes, together with non-node N-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, and 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 that 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 - all leaves fall. Since the Southern Reflection between the ToF nodes happens only within a plane, ToF nodes in other planes cannot discover the case of fallen leaves in a different plane, and cannot determine beyond their local plane whether a Leaf node that was initially reachable has become unreachable. As above, the breakage can be observed by the ToF nodes in the plane where the breakage happened, and then again, 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 N-TIEs from the affected Leaf nodes, if there are only 3 levels and the ToP nodes are directly connected to the Leaf nodes, and then again it can only be effective it is propagated transitively to the Leaf, and useless above that level.

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

- Leaf111 trying to route to Leaf122 MUST choose Spine 111 in plane A as its next hop since plane B will inevitably blackhole the packet when forwarding using default routes or do excessive bow tie'ing, i.e. this information must be in its routing table.

- any kind of "flooding" or distance vector trying to deal with the problem by distributing host routes will be able to converge only using paths through leafs, i.e. the flooding of information on Leaf122 will go up to Top-of-Fabric A and then "loopback" over other leafs to ToF B leading in extreme cases to traffic for Leaf122 when presented to plane B taking an "inverted fabric" path where leafs start to serve as TOFs.

5.1.4. Discovering Fallen Leaves

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

In more detail, by reserving two ports on each Top-of-Fabric node it is possible to connect them together in an interplane bi-directional ring as illustrated in Figure 13 (where we show a bi-directional ring connecting switches across planes). The rings will exchange full topology information between planes and with that allow consequently by the means of transitive, negative disaggregation described in Section 5.2.5.2 to efficiently fix any possible fallen leaf scenario. Somewhat as a side-effect, the exchange of information fulfills the requirement to present full view of the fabric topology at the Top-of-Fabric level without the need to collate it from multiple points by additional complexity of technologies like [RFC7752].

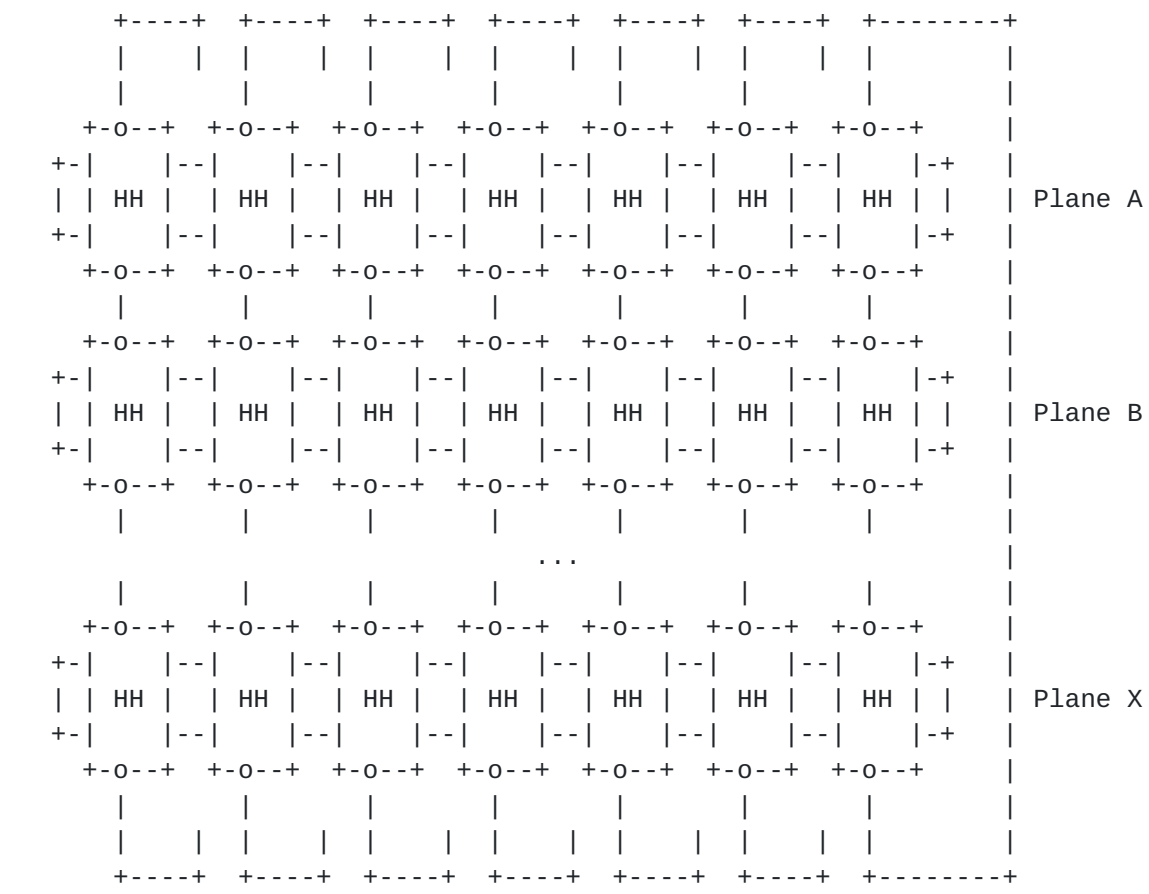

Figure 13: Connecting Top-of-Fabric Nodes Across Planes by Two Rings

5.1.5. Addressing the Fallen Leaves Problem

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

When used for the operation of disaggregation, a positive S-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, meaning that the receiver does not generate its own flooding south as

a consequence of receiving positive disaggregation advertisements from an higher level node. The effect of a positive disaggregation is that the traffic to the impacted prefix will follow the prefix longest match and will be limited to the northbound routers that advertised the more specific route.

In contrast, the negative disaggregation is transitive, and is propagated south when all the possible routes northwards are barred. A negative route advertisement is only actionable when the negative prefix is aggregated by a positive route advertisement for a shorter prefix. In that case, the negative advertisement carves an exception to the positive route in the routing table (one could think of "punching a hole"), making the positive prefix reachable through the originator with the special consideration of the negative prefix removing certain next hop neighbors.

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

On the other hand, when the fabric is partitioned in planes, the positive disaggregation from ToF nodes in different planes do not reach the ToP switches in the affected plane and cannot solve the fallen leaves problem. In other words, a breakage in a plane can only be solved in that plane. Also, the selection of the plane for a packet typically occurs at the leaf level and the disaggregation must be transitive and reach all the leaves. In that case, the negative disaggregation is necessary. The details on the RIFT approach to deal with fallen leafs in an optimal way is specified in Section 5.2.5.2.

5.2. Specification

5.2.1. Transport

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

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

5.2.2. Link (Neighbor) Discovery (LIE Exchange)

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

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

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

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

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

Unless Section 5.2.7 is used, each node is provisioned with the level at which it is operating and its PoD (or otherwise a default level and "undefined" PoD are assumed; meaning that leafs do not need to be

configured at all if initial configuration values are all left at 0). Nodes in the spine are configured with "any" PoD which has the same value "undefined" PoD hence we will talk about "undefined/any" PoD. This information is propagated in the LIEs exchanged.

Further definitions of leaf flags are found in Section 5.2.7 given they have implications in terms of level and adjacency forming here.

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

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

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

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

4. the neighboring node uses a valid System ID AND

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

6. the advertised MTUs match on both sides AND

7. both nodes advertise defined level values AND

8. [

   i) the node is at level 0 and has no three way adjacencies already to HAT nodes with level different than the adjacent node OR

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

   iii) both nodes are at level 0 AND both indicate support for Section 5.3.9 OR

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

   ].

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

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

LIEs arriving with a TTL larger than 1 MUST be ignored.

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

LIE exchange uses three way handshake mechanism which is a cleaned up version of [RFC5303]. Observe that for easier comprehension the terminology of one/two and three-way states does NOT align with OSPF or ISIS FSMs albeit they use roughly same mechanisms.

5.2.3. Topology Exchange (TIE Exchange)

5.2.3.1. Topology Information Elements

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

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

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

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

5.2.3.2. South- and Northbound Representation

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

The N-TIEs hold all of the node's adjacencies and local prefixes while the S-TIEs hold only all of the node's adjacencies, the default prefix with necessary disaggregated prefixes and local prefixes. We will explain this in detail further in Section 5.2.5.

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

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

Table 2: TIE Types

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

```
      Spine21 S-TIEs:
      Node S-TIE:
        NodeElement(level=2, neighbors((Spine 111, level 1, cost 1),
        (Spine 112, level 1, cost 1), (Spine 121, level 1, cost 1),
        (Spine 122, level 1, cost 1)))
      Prefix S-TIE:
        SouthPrefixesElement(prefixes(0/0, cost 1), (::/0, cost 1))

      Spine 111 S-TIEs:
      Node S-TIE:
        NodeElement(level=1, neighbors((Spine21, level 2, cost 1,
links(...)),
        (Spine22, 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 S-TIE:
        SouthPrefixesElement(prefixes(0/0, cost 1), (::/0, cost 1))

      Spine 111 N-TIEs:
      Node N-TIE:
        NodeElement(level=1,
        neighbors((Spine21, level 2, cost 1, links(...)),
        (Spine22, 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 N-TIE:
        NorthPrefixesElement(prefixes(Spine 111.loopback)

      Spine 121 S-TIEs:
      Node S-TIE:
        NodeElement(level=1, neighbors((Spine21,level 2,cost 1),
        (Spine22, level 2, cost 1), (Leaf121, level 0, cost 1),
        (Leaf122, level 0, cost 1)))
      Prefix S-TIE:
        SouthPrefixesElement(prefixes(0/0, cost 1), (::/0, cost 1))

      Spine 121 N-TIEs:
```

Node N-TIE:

```
        NodeElement(level=1,
        neighbors((Spine21, level 2, cost 1, links(...)),
        (Spine22, level 2, cost 1, links(...)),
        (Leaf121, level 0, cost 1, links(...)),
        (Leaf122, level 0, cost 1, links(...))))
      Prefix N-TIE:
        NorthPrefixesElement(prefixes(Spine 121.loopback)

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

Figure 14: example TIES generated in a 2 level spine-and-leaf topology

5.2.3.3. Flooding

The mechanism used to distribute TIEs is the well-known (albeit modified in several respects to address fat tree requirements) flooding mechanism used by today's link-state protocols. Although flooding is initially more demanding to implement it avoids many problems with update style used in diffused computation such as distance vector protocols. Since flooding tends to present an unscalable burden in large, densely meshed topologies (fat trees being unfortunately such a topology) we provide as solution a close to optimal global flood reduction and load balancing optimization in Section 5.2.3.9.

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

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

Precise finite state machines and procedures can be found in Appendix C.3.

5.2.3.4. TIE Flooding Scopes

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

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

A node's Node S-TIEs, consisting of all node's adjacencies and prefix S-TIEs limited to those related to default IP prefix and disaggregated prefixes, are flooded southbound in order to allow the nodes one level down to see connectivity of the higher level as well as reachability to the rest of the fabric. In order to allow an E-W disconnected node in a given level to receive the S-TIEs of other nodes at its level, every *NODE* S-TIE is "reflected" northbound to level from which it was received. It should be noted that East-West links are included in South TIE flooding (except at ToF level); those TIEs need to be flooded to satisfy algorithms in Section 5.2.4. In that way nodes at same level can learn about each other without a lower level, e.g. in case of leaf level. The precise flooding scopes are given in Table 3. Those rules govern as well what SHOULD be included in TIDEs on the adjacency. Again, East-West flooding scopes are identical to South flooding scopes except in case of ToF East-West links (rings) which are basically performing northbound flooding.

Node S-TIE "south reflection" allows to support positive disaggregation on failures describes in Section 5.2.5 and flooding reduction in Section 5.2.3.9.

Type / Direction	South	North	East-West
node S-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 S-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 N-TIEs	never flood	flood always	flood only if this node is ToF
TIDE	include at least all non-self originated N-TIE headers and self-originated S-TIE headers and node S-TIEs of nodes at same level	include at least all node S-TIEs and all S-TIEs originated by peer and all N-TIEs	if this node is ToF then include all N-TIEs, otherwise only self-originated TIEs
TIRE as Request	request all N-TIEs and all peer's self-originated TIEs and all node S-TIEs	request all S-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: Flooding Scopes

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

As an example to illustrate these rules, consider using the topology in Figure 2, with the optional link between spine 111 and spine 112,

and the associated TIEs given in Figure 14. The flooding from particular nodes of the TIEs is given in Table 4.

Router floods to	Neighbor	TIEs
Leaf111	Spine 112	Leaf111 N-TIEs, Spine 111 node S-TIE
Leaf111	Spine 111	Leaf111 N-TIEs, Spine 112 node S-TIE
Spine 111	Leaf111	Spine 111 S-TIEs
Spine 111	Leaf112	Spine 111 S-TIEs
Spine 111	Spine 112	Spine 111 S-TIEs
Spine 111	Spine21	Spine 111 N-TIEs, Leaf111 N-TIEs, Leaf112 N-TIEs, Spine22 node S-TIE
Spine 111	Spine22	Spine 111 N-TIEs, Leaf111 N-TIEs, Leaf112 N-TIEs, Spine21 node S-TIE
...	...	...
Spine21	Spine 111	Spine21 S-TIEs
Spine21	Spine 112	Spine21 S-TIEs
Spine21	Spine 121	Spine21 S-TIEs
Spine21	Spine 122	Spine21 S-TIEs
...	...	...

Table 4: Flooding some TIEs from example topology

5.2.3.5. 'Flood Only Node TIEs' Bit

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

Obviously this mechanism is most useful in southbound direction. The distribution of node TIEs guarantees correct behavior of algorithms like disaggregation or default route origination. Furthermore

though, the use of this bit presents an inherent trade-off between processing load and convergence speed since suppressing flooding of northbound prefixes from neighbors will lead to blackholes.

5.2.3.6. Initial and Periodic Database Synchronization

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

5.2.3.7. Purging and Roll-Overs

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

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

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

TIE sequence numbers are rolled over using the method described in Appendix A. First sequence number of any spontaneously originated TIE (i.e. not originated to override a detected older copy in the network) MUST be a reasonbly unpredictable random number in the interval [0, 2^10-1] which will prevent otherwise identical TIE headers to remain "stuck" in the network with content different from TIE originated after reboot.

5.2.3.8. Southbound Default Route Origination

Under certain conditions nodes issue a default route in their South Prefix TIEs with costs as computed in Section 5.3.6.1.

A node X that

1. is NOT overloaded AND

2. has southbound or East-West adjacencies

originates in its south prefix TIE such a default route IIF

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

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

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

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

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

5.2.3.9. Northbound TIE Flooding Reduction

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

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

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

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

3. to control the volume of its flooding two hops North and yet keep it robust enough, it is advantageous for a node to select a subset of its parents as "Flood Repeaters" (FRs), which combined

together deliver two or more copies of its flooding to all of its parents, i.e. the originating node's grandparents;

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

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

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

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

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

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

set to true, then the northbound neighbor is a flood repeater for the node that sent the LIE and MUST reflood northbound TIEs received from that node.

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

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

(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):

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

    2. set i=0;

    3. while i < CN(N) do

        1. set j=i;

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

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

            2. set i=i+1;

        3. /* 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) */

        4. set k=k+1;

    4. /* 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:

    1. while k > 0 do

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

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

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

2. 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):

    1. 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:

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

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

            1. place P in FR set;

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

    2. 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 S-TIE with changed adjacencies. A node MAY apply a hysteresis to prevent excessive amount of computation during periods of network instability just like in case of reachability computation.

2. A node SHOULD send out LIEs that grant flood repeater status before LIEs that revoke it on flood repeater set changes to prevent transient behavior where the full coverage of grand parents is not guaranteed. Albeit the condition will correct in positively stable manner due to LIE retransmission and periodic TIDEs, it can slow down flooding convergence on flood repeater status changes.

3. A node always floods its self-originated TIEs.

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

5. The indication of flood reduction capability is carried in the node TIEs and can 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 first request ONLY. 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 should load-balance the incoming database between adjacent nodes and flood repeaters should guarantee that two copies are sent by different nodes to ensure against any losses.

7. Obviously sine flooding reduction does NOT apply to self originated TIEs and since all policy-guided information consists of self-originated TIEs those are unaffected.

5.2.3.10. Special Considerations

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

More difficult is a condition where a node floods a TIE north towards a super-spine, then its spine reboots, in fact partitioning superspine from it directly and then the node itself reboots. That leaves in a sense the super-spine holding the "primary copy" of the node's TIE. Normally this condition is resolved easily by the node re-originating its TIE with a higher sequence number than it sees in northbound TIEs, here however, when spine comes back it won't be able to obtain a N-TIE from its superspine easily and with that the node below may issue the same version of the TIE with a lower sequence number. Flooding procedures are are extended to deal with the problem by the means of special clauses that override the database of a lower level with headers of newer TIEs seen in TIDEs coming from the north.

5.2.4. Reachability Computation

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

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

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

5.2.4.1. Northbound SPF

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

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

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

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

2. the adjacent node has one or more northbound adjacencies

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

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

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

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

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

A detailed example can be found in Section 6.4.

5.2.4.2. Southbound SPF

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

S-SPF uses northbound adjacencies in node N-TIEs to verify backlink connectivity.

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

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

Using south prefixes over horizontal links is optional and can protect against pathological fabric partitioning cases that leave only paths to destinations that would necessitate multiple changes of forwarding direction between north and south.

5.2.4.4. East-West Links Within ToF Level

E-W ToF links behave in terms of flooding scopes defined in Section 5.2.3.4 like northbound links. Even though a ToF node could be tempted to use those links during southbound SPF this MUST NOT be attempted since it may lead in, e.g. anycast cases to routing loops. An implemention could 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.

5.2.5. Automatic Disaggregation on Link & Node Failures

5.2.5.1. Positive, Non-transitive Disaggregation

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

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

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

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

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

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

4. Identical set of de-aggregated prefixes is flooded on each of the node's southbound adjacencies. In accordance with the normal flooding rules for an S-TIE, a node at the lower level that receives this S-TIE will not propagate it south-bound. Neither is it necessary for the receiving node to reflect the disaggregated prefixes back over its adjacencies to nodes at the level from which it was received.

To summarize the above in simplest terms: if a node detects that its default route encompasses prefixes for which one of the other nodes in its level has no possible next-hops in the level below, it has to disaggregate it to prevent black-holing or suboptimal routing 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 6.3.

A possible algorithm is described last:

1. Create partial_neighbors = (empty), a set of neighbors with partial connectivity to the node X's level from X's perspective. Each entry is a list of 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 S-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 to disaggregate any prefixes. If node X is advertising disaggregated prefixes in its S-TIE, X SHOULD remove them and re-advertise its according S-TIEs.

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

```
disaggregated_prefixes = { empty }
nodes_same_level = { empty }
for each S-TIE
  if (S-TIE.level == X.level and
      X shares at least one S-neighbor with X)
    add S-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 S-TIE
if X's S-TIE is different
  schedule S-TIE for flooding
  end if
```

Figure 15: Computation of Disaggregated Prefixes

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

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

1. all neighbor relationships MUST perform backlink checks.

2. overload bits as introduced in Section 5.3.1 have to be respected during the computation.

3. all the lower level nodes are flooded the same disaggregated prefixes since we don't want to build an S-TIE per node and

complicate things unnecessarily. The PoD containing the prefix will prefer southbound anyway.

4. 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 S-TIEs are not "reflected" by the lower level, i.e. nodes within same level do NOT need to be aware which node computed the need for disaggregation.

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

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

To close this section it is worth to observe that in a single plane ToF this disaggregation prevents blackholing up to (K_LEAF * P) link failures in terms of Section 5.1.2 or in other terms, it takes at minimum that many link failures to partition the ToF into multiple planes.

5.2.5.2. Negative, Transitive Disaggregation for Fallen Leafs

As explained in Section 5.1.3 failures in multi-plane Top-of-Fabric or more than (K_LEAF * P) links failing in single plane design can generate fallen leafs. Such scenario cannot be addressed by positive disaggregation only and needs a further mechanism.

5.2.5.2.1. Cabling of Multiple Top-of-Fabric Planes

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

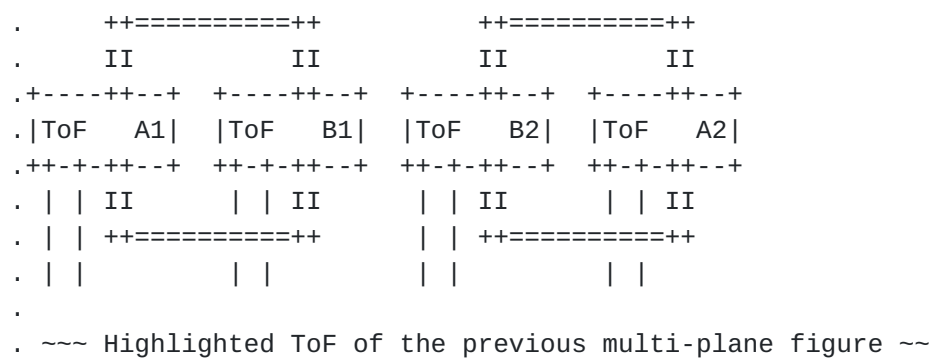

Figure 16: Topologically connected planes

As described in Section 5.1.3 failures in multi-plane fabrics can lead to blackholes which normal positive disaggregation cannot fix. The mechanism of negative, transitive disaggregation incorporated in RIFT provides the according solution.

5.2.5.2.2. Transitive Advertisement of Negative Disaggregates

A ToF node that discovers that it cannot reach a fallen leaf disaggregates all the prefixes of such leafs. It uses for that purpose negative prefix S-TIEs that are, as usual, flooded southwards with the scope defined in Section 5.2.3.4.

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

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

5.2.5.2.3. Computation of Negative Disaggregates

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

The first origination reason is a computation that uses all the node N-TIEs to build the set of all reachable nodes by reachability computation over the complete graph and including ToF links. The computation uses the node itself as root. This is compared with the result of the normal southbound SPF as described in Section 5.2.4.2. The difference are the fallen leafs and all their attached prefixes are advertised as negative prefixes southbound if the node does not see the prefix being reachable within southbound SPF.

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

5.2.6. Attaching Prefixes

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

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

Each prefix has its cost adjusted before being added into the RIFT route database. The cost of the prefix is set to the cost received plus the cost of the minimum distance next-hop to that neighbor while taking into account its attributes such as mobility per Section 5.3.3 necessary. 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 attributes and only the best combination is used for forwarding. RIFT route preferences are normalized by the according Thrift [thrift] model type.

An example implementation for node X follows:

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

Figure 17: Adding Routes from S-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 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. Similarily, if a nexthop is deleted for a given prefix, then it is

deleted for all the immediately aggregated negative routes. This will recurse in the case of nested negative prefix aggregations.

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

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

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

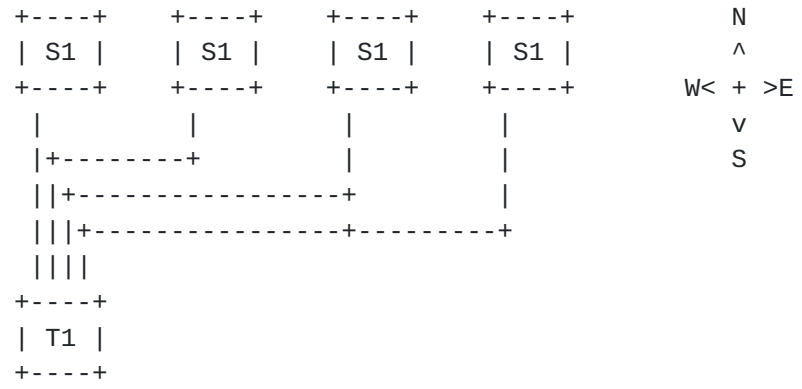

Figure 18: A ToP node with 4 parents

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

Figure 19: Abstract RIB

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

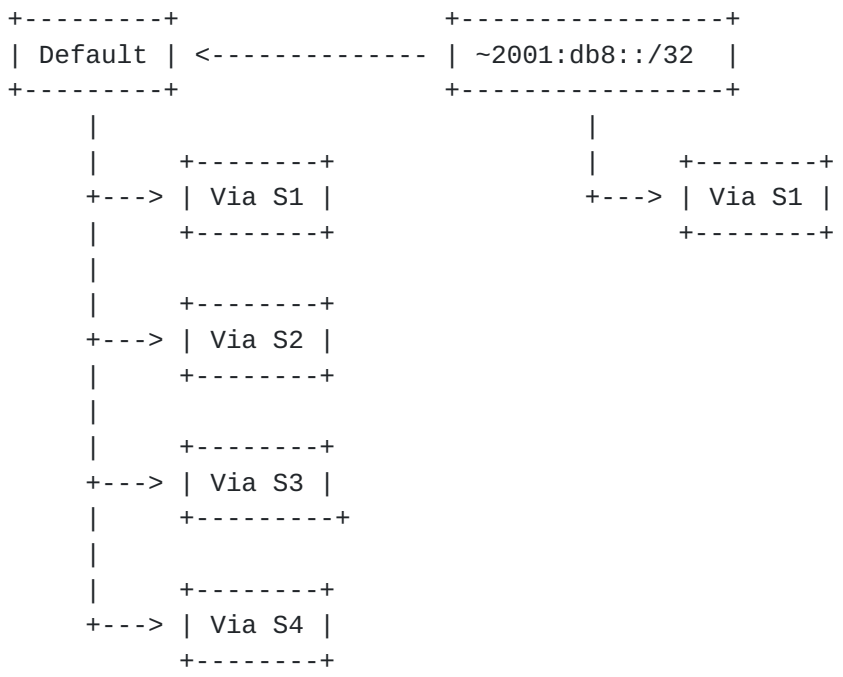

Figure 20: Abstract RIB after negative 2001:db8::/32 from S1

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

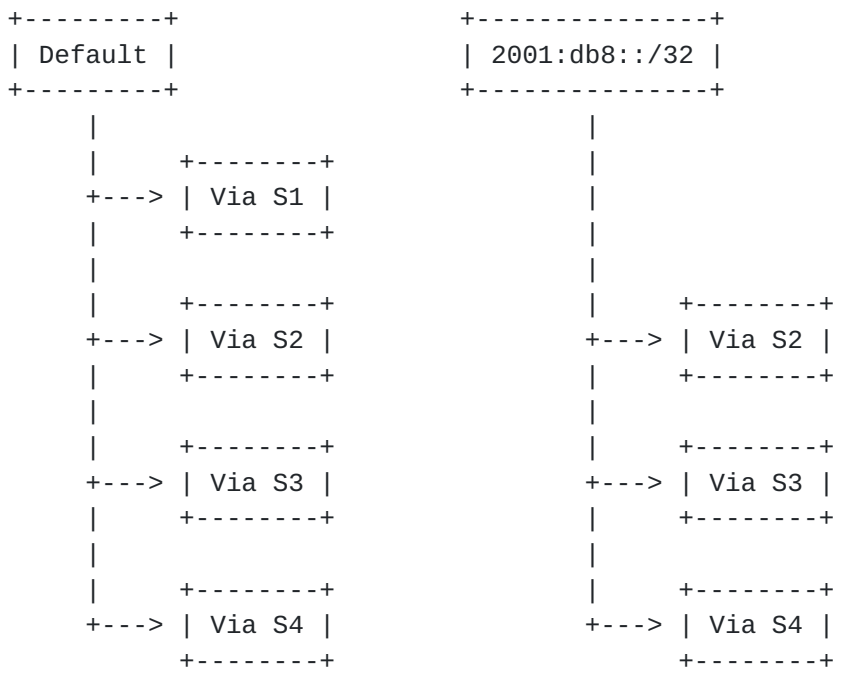

Figure 21: Abstract FIB after negative 2001:db8::/32 from S1

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

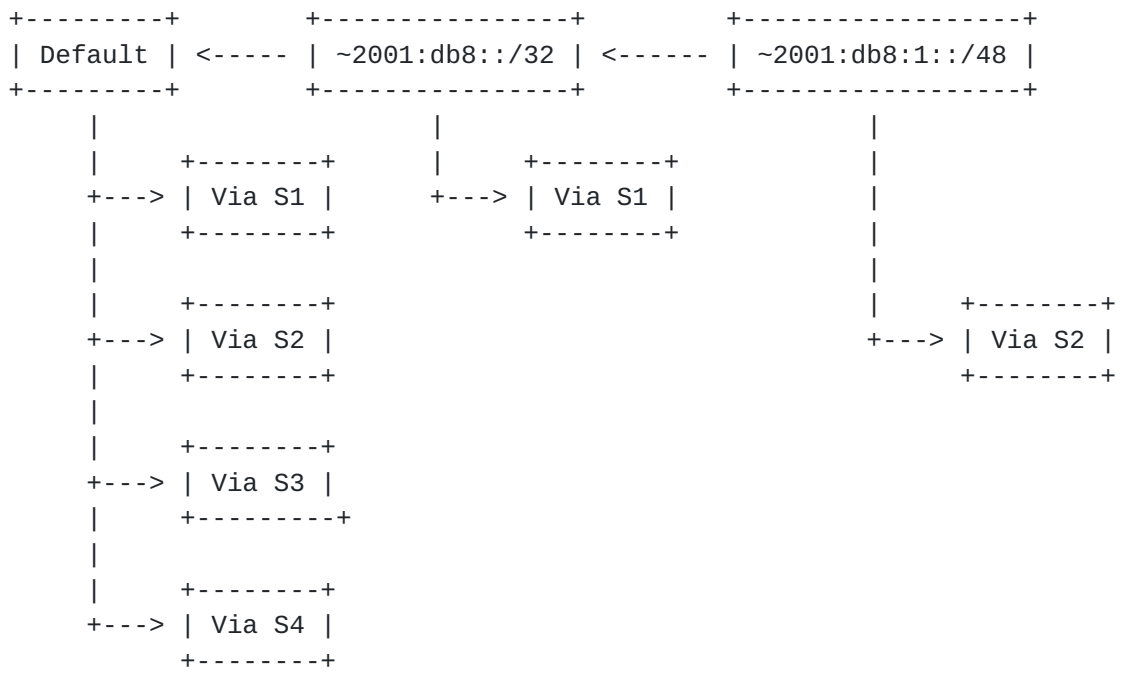

Figure 22: Abstract RIB after negative 2001:db8:1::/48 from S2

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

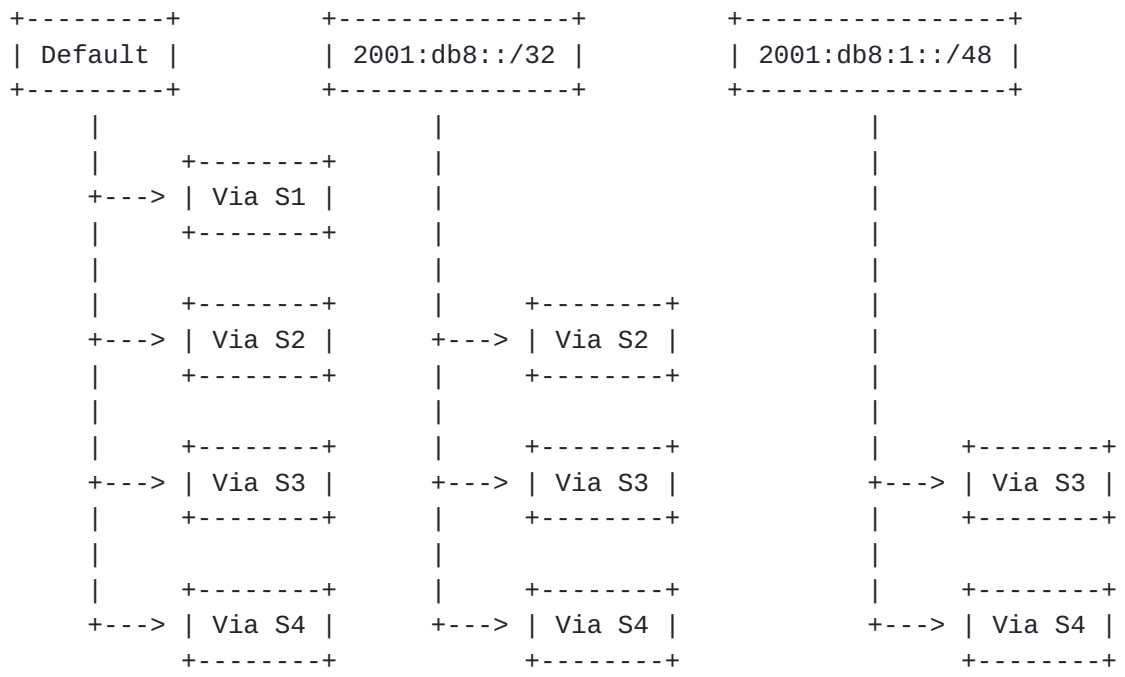

Figure 23: Abstract FIB after negative 2001:db8:1::/48 from S2

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

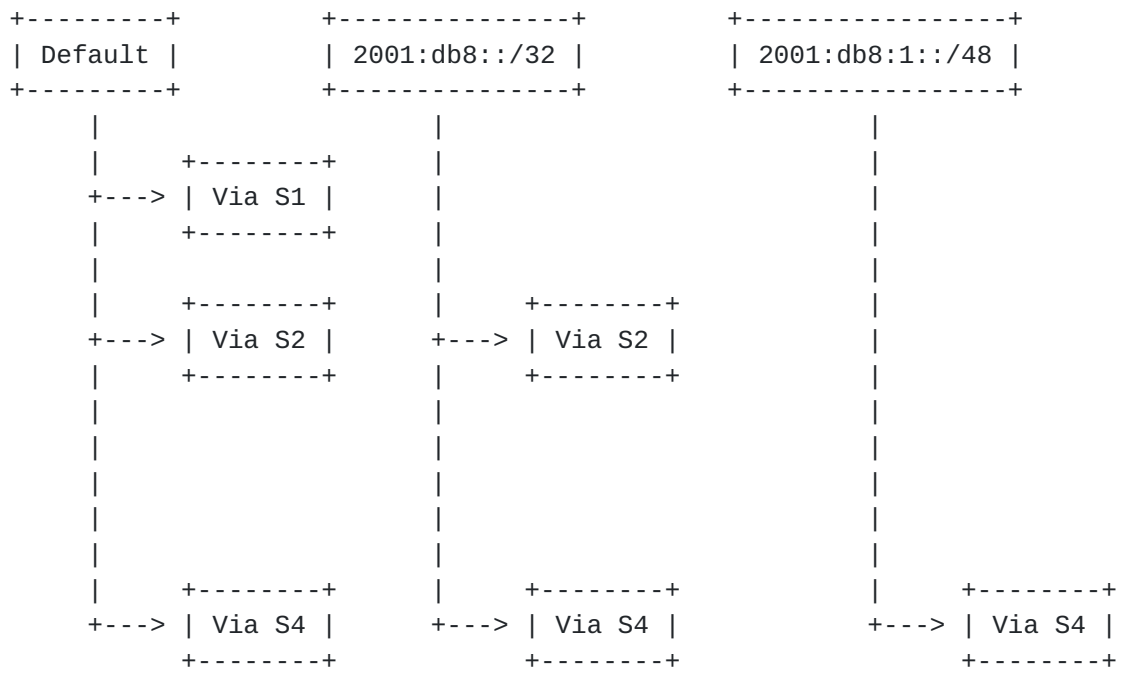

Figure 24: Abstract FIB after loss of S3

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

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

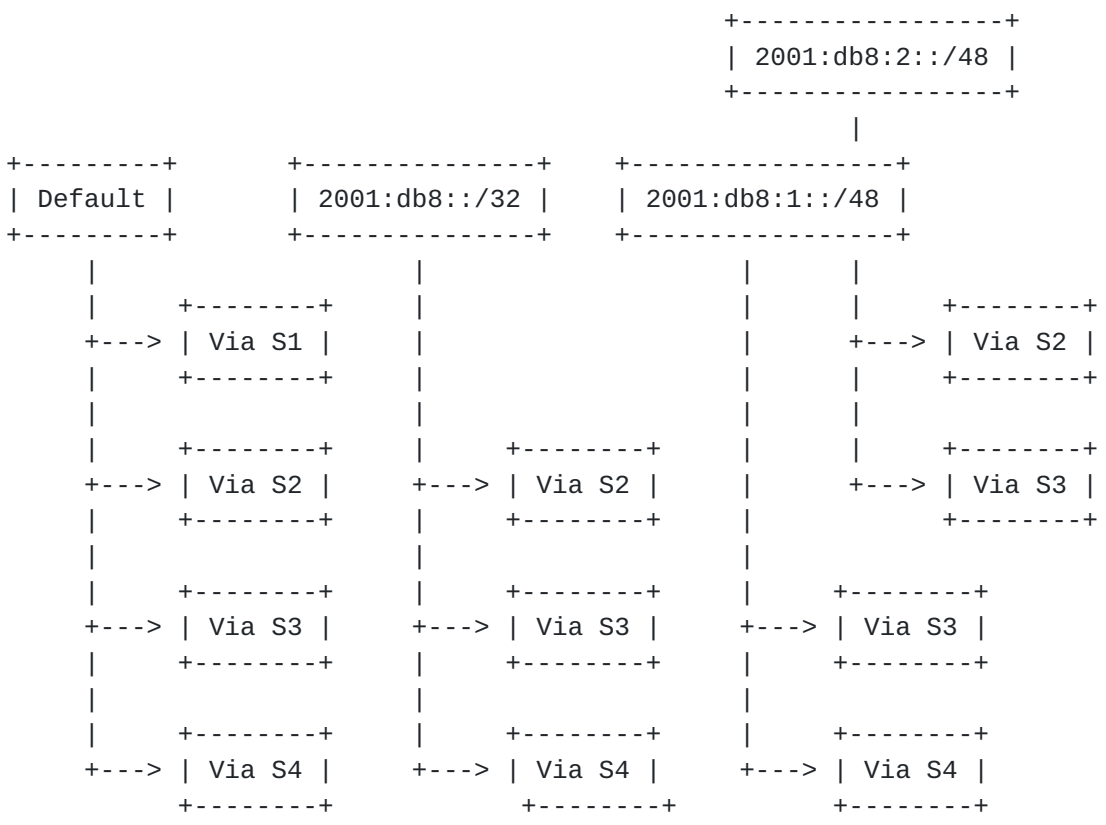

Figure 25: Abstract FIB after negative 2001:db8:2::/48 from S4

5.2.7. Optional Zero Touch Provisioning (ZTP)

Each RIFT node can operate in zero touch provisioning (ZTP) mode, i.e. it has no configuration (unless it is a Top-of-Fabric at the top of the topology or the must operate in the topology as leaf and/or support leaf-2-leaf procedures) and it will fully configure itself after being attached to the topology. 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 leafs) tries to attach at the highest possible point in the fabric. This guarantees that even if the diffusion front reaches a node from "below" faster than from "above", it will greedily abandon already negotiated level derived from nodes topologically below it and properly peers with nodes above.

The fabric is very conciously numbered from the top to allow for PoDs of different heights and minimize number of provisioning necessary,

in this case just a TOP_OF_FABRIC flag on every node at the top of the fabric.

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

5.2.7.1. Terminology

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

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

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

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

TOP_OF_FABRIC flag: Configuration flag that MUST be provided to all Top-of-Fabric nodes. LEAF_FLAG and CONFIGURED_LEVEL cannot be defined at the same time as this flag. It implies a CONFIGURED_LEVEL value. In fact, it is basically a shortcut for configuring same level at all Top-of-Fabric nodes which is unavoidable since an initial 'seed' is needed for other ZTP nodes to derive their level in the topology. The flag plays an important role in fabrics with multiple planes to enable successful negative disaggregation (Section 5.2.5.2).

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

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

LEAF_2_LEAF: An optional flag that can be configured on a node to make sure it supports procedures defined in Section 5.3.9. In a strict sense it is a capability that implies LEAF_ONLY and the

according restrictions. TOP_OF_FABRIC flag is ignored when set at the same time as this flag.

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

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

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

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

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

5.2.7.2. Automatic SystemID Selection

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

As matter of operational concern, the router MUST ensure that such identifier is not changing very frequently (or at least not without sending all its TIEs with fairly short lifetimes) since otherwise the network may be left with large amounts of stale TIEs in other nodes (though this is not necessarily a serious problem if the procedures described in Section 8 are implemented).

5.2.7.3. Generic Fabric Example

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

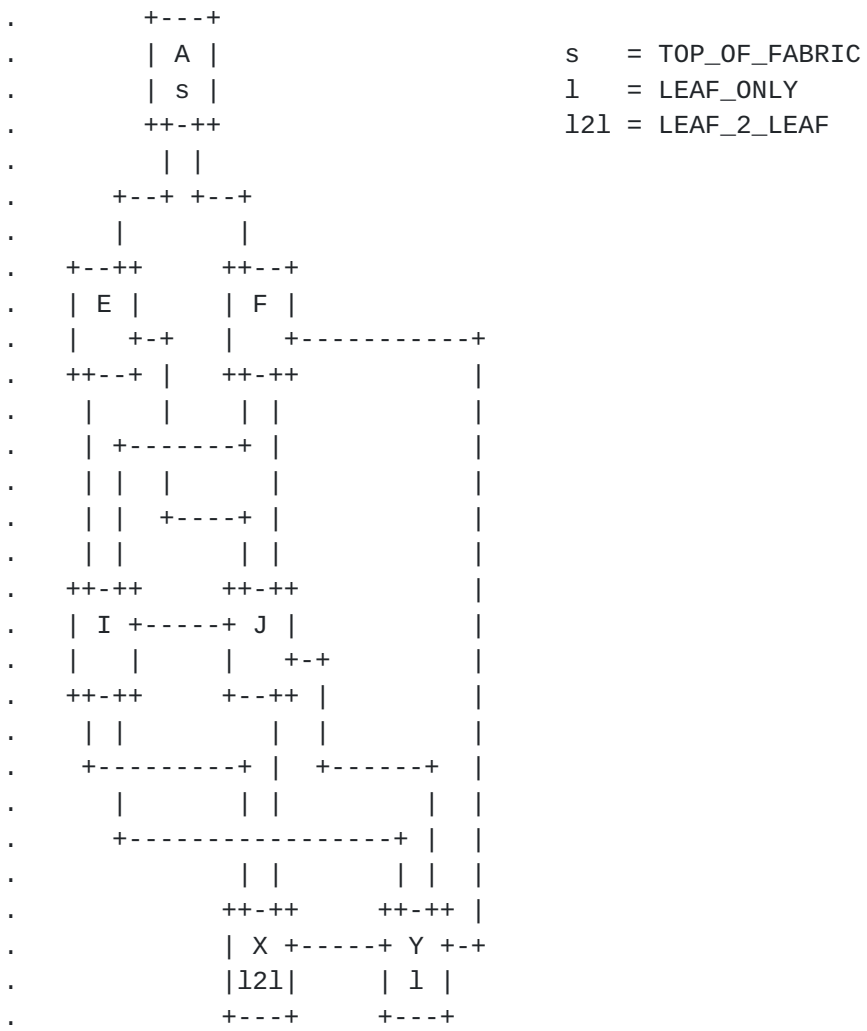

Figure 26: Generic ZTP Cabling Considerations

First, we must anchor the "top" of the cabling and that's what the TOP_OF_FABRIC flag at node A is for. Then things look smooth until we have to decide whether node Y is at the same level as I, J or at the same level as Y and consequently, X is south of it. This is unresolvable here until we "nail down the bottom" of the topology. To achieve that we choose to use in this example the leaf flags. We

will see further then whether Y chooses to form adjacencies to F or I, J successively.

5.2.7.4. Level Determination Procedure

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

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

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

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

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

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

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

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

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

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

1. It computes HAT per procedures above but does NOT use it to compute DERIVED_LEVEL. HAT is used to limit adjacency formation per Section 5.2.2.

It MAY also follow modified procedures:

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

5.2.7.5. Resulting Topologies

The procedures defined in Section 5.2.7.4 will lead to the RIFT topology and levels depicted in Figure 27.

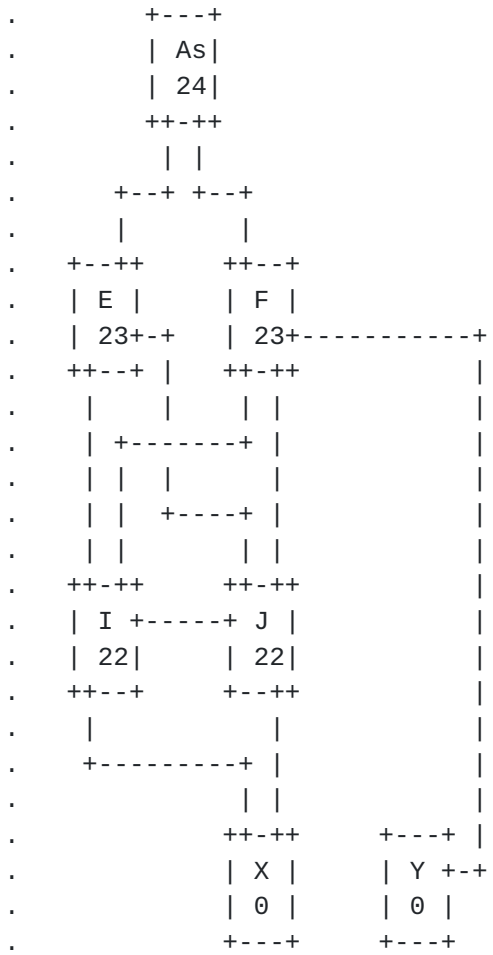

Figure 27: Generic ZTP Topology Autoconfigured

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

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

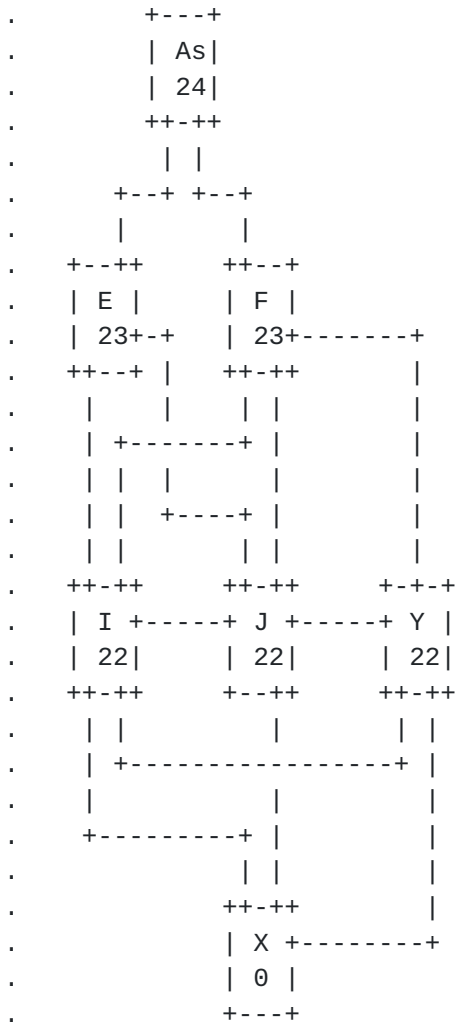

Figure 28: Generic ZTP Topology Autoconfigured

5.2.8. Stability Considerations

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

5.3. Further Mechanisms

5.3.1. Overload Bit

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

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

5.3.2. Optimized Route Computation on Leafs

Since the leafs do see only "one hop away" they do not need to run a "proper" SPF. Instead, they can gather the available prefix candidates from their neighbors and build the routing table accordingly.

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

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

Then a leaf attaches prefixes as described in Section 5.2.6.

5.3.3. Mobility

It is a requirement for RIFT to maintain at the control plane a real time status of which prefix is attached to which port of which leaf, even in a context of mobility where the point of attachement may change several times in a subsecond period of time.

There are two classical approaches to maintain such knowledge in an unambiguous fashion:

time stamp: With this method, the infrastructure records 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 to be able to compare time stamps as observed by the various points of attachment, e.g., using the variation of the Precision Time Protocol (PTP) IEEE Std. 1588 [IEEEstd1588], [IEEEstd8021AS] designed for bridged LANs IEEE Std. 802.1AS [IEEEstd8021AS]. Both the precision of the synchronisation protocol and the resolution of the time stamp must beat the highest possible roaming time on the fabric. Another drawback is that the presence of the mobile device may be observed only asynchronously, e.g., after it starts using an IP protocol such as ARP [RFC0826], IPv6 Neighbor Discovery [RFC4861][RFC4862], or DHCP [RFC2131][RFC8415].

sequence counter: With this method, a mobile node 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 device. The disadvantage of this approach is the lack of support for protocols that may be used by the mobile node to register its presence to the leaf node with the capability to provide a sequence counter. Well-known issues with wrapping sequence counters must be addressed properly, and many forms of sequence counters that vary in both wrapping rules and comparison rules. A particular knowledge of the source of the sequence counter is required to operate it, and the comparison between sequence counters from heterogeneous sources can be hard to impossible.

RIFT supports a hybrid approach contained in an optional `PrefixSequenceType` prefix attribute that we call a `monotonic clock` consisting of a timestamp and optional sequence number. In case of presence of the attribute:

- The leaf node MAY advertise a time stamp 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 time stamp within the desired prefix N-TIEs as 802.1AS timestamp.

- RIFT may interoperate with the "update to 6LoWPAN Neighbor Discovery" [RFC8505], which provides a method for registering a prefix with a sequence counter called a Transaction ID (TID). RIFT transports in such case the TID in its native form.

- RIFT also defines an abstract negative clock (ANSC) that compares as less than any other clock. By default, the lack of a `PrefixSequenceType` in a Prefix N-TIE is interpreted as ANSC. We call this also an `undefined` clock.

- Any prefix present on the fabric in multiple nodes that has the `same` clock is considered as anycast. ASNC is always considered smaller than any defined clock.

- RIFT implementation assumes by default that all nodes are being synchronized to 200 milliseconds precision which is easily achievable even in very large fabrics using [RFC5905]. An implementation MAY provide a way to reconfigure a domain to a different value. We call this variable MAXIMUM_CLOCK_DELTA.

5.3.3.1. Clock Comparison

All monotonic clock values are comparable to each other using the following rules:

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

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

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

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

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

5.3.3.2. Interaction between Time Stamps and Sequence Counters

For slow movements that occur less frequently than e.g. once per second, the time stamp that the RIFT infrastruture captures is enough to determine the freshest discovery. If the point of attachement changes faster than the maximum drift of the time stamping mechanism (i.e. MAXIMUM_CLOCK_DELTA), then a sequence counter is required to add resolution to the freshness evaluation, and it must be sized so

that the counters stay comparable within the resolution of the time stampling mechanism.

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

Within the resolution of MAXIMUM_CLOCK_DELTA the sequence counters captured during 2 sequential values of the time stamp SHOULD be comparable. This means with default values that a node may move up to 127 times during a 200 milliseconds period and the clocks remain still comparable thus allowing the infrastructure to assert the freshest advertisement with no ambiguity.

5.3.3.3. Anycast vs. Unicast

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

If a monotonic clock attribute is provided on the prefix, then the prefix with the `newest` clock value is strictly prefered. An anycast prefix does not carry a clock or all clock attributes MUST be the same under the rules of Section 5.3.3.1.

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

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

5.3.3.4. Overlays and Signaling

RIFT is agnostic whether any overlay technology like [MIP, LISP, VxLAN, NVO3] and the associated signaling is deployed over it. But it is expected that leaf nodes, and possibly Top-of-Fabric nodes can perform the correct encapsulation.

In the context of mobility, overlays provide a classical solution to avoid injecting mobile prefixes in the fabric and improve the scalability of the solution. It makes sense on a data center that already uses overlays to consider their applicability to the mobility solution; as an example, a mobility protocol such as LISP may inform the ingress leaf of the location of the egress leaf in real time.

Another possibility is to consider that mobility as an underlay service and support it in RIFT to an extent. The load on the fabric augments with the amount of mobility obviously since a move forces flooding and computation on all nodes in the scope of the move so

tunneling from leaf to the Top-of-Fabric may be desired. Future versions of this document may describe support for such tunneling in RIFT.

5.3.4. Key/Value Store

5.3.4.1. Southbound

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

1. Only KV TIEs originated by nodes to which the receiver has a bi-directional adjacency are considered.

2. Within all such valid KV S-TIEs containing the key, the value of the KV S-TIE for which the according node S-TIE is present, has the highest level and within the same level has highest originating system ID is preferred. If keys in the most preferred TIEs are overlapping, the behavior is undefined.

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

5.3.4.2. Northbound

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

5.3.5. Interactions with BFD

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

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

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

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

In case RIFT changes link identifiers or BFD capability indication both the LIE as well as the BFD sessions SHOULD be brought down and back up again.

Multiple RIFT instances MAY choose to share a single BFD session (in such case it is undefined what discriminators are used albeit RIFT CAN advertise the same link ID for the same interface in multiple instances and with that "share" the discriminators).

BFD TTL follows [RFC5082].

5.3.6. Fabric Bandwidth Balancing

A well understood problem in fabrics is that in case of link losses it would be ideal to rebalance how much traffic is offered to switches in the next level based on the ingress and egress bandwidth they have. Current attempts rely mostly on specialized traffic engineering via controller or leafs being aware of complete topology with according cost and complexity.

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

5.3.6.1. Northbound Direction

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

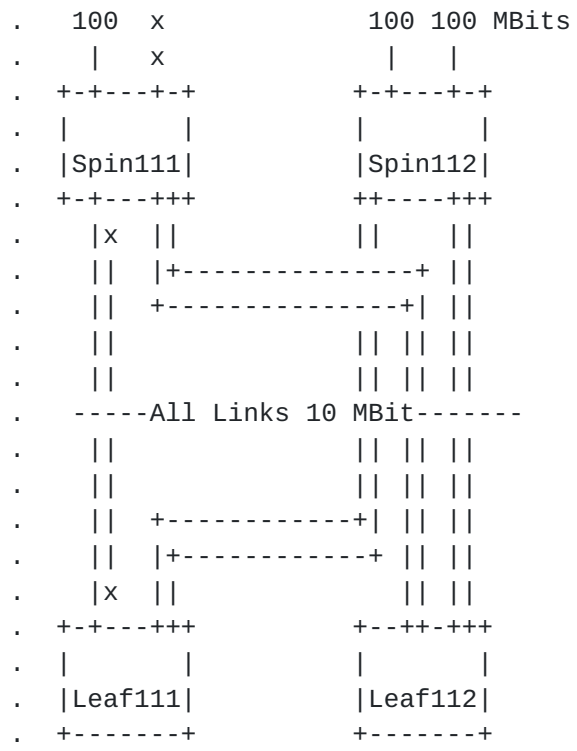

Figure 29: Balancing Bandwidth

All links from Leafs in Figure 29 are assumed to 10 MBit/s bandwidth while the uplinks one level further up are assumed to be 100 MBit/s. Further, in Figure 29 we assume that Leaf111 lost one of the parallel links to Spine 111 and with that wants to possibly push more traffic onto Spine 112. Leaf 112 has equal bandwidth to Spine 111 and Spine 112 but Spine 111 lost one of its uplinks.

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 adjustements.

On a node L use Node TIEs to compute for each non-overloaded northbound neighbor N three values:

L_N_u: as sum of the bandwidth available to N

N_u: as sum of the uplink bandwidth available on N

T_N_u: as sum of L_N_u * OVERSUBSCRIPTION_CONSTANT + N_u

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

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 the understanding. We assume the default route distance is advertised with D=1 everywhere and OVERSUBSCRIPTION_CONSTANT = 1.

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

Table 5: BAD Computation

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

BAD is only computed for default routes. A node MAY compute and use BAD for any disaggregated prefixes or other RIFT routes. A node MAY use another algorithm than BAD to weight northbound traffic based on bandwidth given that the algorithm is distributed and un-synchronized and ultimately, its correct behavior does not depend on uniformity of balancing algorithms used in the fabric. E.g. it is conceivable that leafs could use real time link loads gathered by analytics to change the amount of traffic assigned to each default route next hop.

Observe further that a change in available bandwidth will only affect at maximum two levels down in the fabric, i.e. blast radius of bandwidth changes is contained no matter its height.

5.3.6.2. Southbound Direction

Due to its loop free properties a node CAN take during S-SPF into account the available bandwidth on the nodes in lower levels and modify the amount of traffic offered to next level's "southbound"

nodes based as what it sees is the total achievable maximum flow through those nodes. It is worth observing that such computations may work better if standardized but does not have to be necessarily. As long the packet keeps on heading south it will take one of the available paths and arrive at the intended destination.

5.3.7. Label Binding

A node MAY advertise on its TIEs a locally significant, downstream assigned label for the according interface. One use of such label is a hop-by-hop encapsulation allowing to easily distinguish forwarding planes served by a multiplicity of RIFT instances.

5.3.8. Segment Routing Support with RIFT

Recently, alternative architecture to reuse labels as segment identifiers [RFC8402] has gained traction and may present use cases in IP fabric that would justify its deployment. Such use cases will either precondition an assignment of a label per node (or other entities where the mechanisms are equivalent) or a global assignment and a knowledge of topology everywhere to compute segment stacks of interest. We deal with the two issues separately.

5.3.8.1. Global Segment Identifiers Assignment

Global segment identifiers are normally assumed to be provided by some kind of a centralized "controller" instance and distributed to other entities. This can be performed in RIFT by attaching a controller to the Top-of-Fabric nodes at the top of the fabric where the whole topology is always visible, assign such identifiers and then distribute those via the KV mechanism towards all nodes so they can perform things like probing the fabric for failures using a stack of segments.

5.3.8.2. Distribution of Topology Information

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

Beside blast radius problem, excessive flooding may present significant load on implementations.

5.3.9. Leaf to Leaf Procedures

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

advertise the LEAF_2_LEAF flag in node capabilities AND

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

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

always use E-W leaf adjacency in both north as well as south computation AND

install a discard route for any advertised aggregate in leaf's TIEs AND

never form southbound adjacencies.

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

5.3.10. Address Family and Multi Topology Considerations

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

BFD interactions in Section 5.3.5 are implementation dependent when multiple RIFT instances run on the same link.

5.3.11. Reachability of Internal Nodes in the Fabric

RIFT does not precondition that its nodes have reachable addresses albeit for operational purposes this is clearly desirable. Under normal operating conditions this can be easily achieved by e.g. injecting the node's loopback address into North and South Prefix TIEs or other implementation specific mechanisms.

Things get more interesting in case a node looses all its northbound adjacencies but is not at the top of the fabric. That is outside the

scope of this document and may be covered in a separate document about policy guided prefixes [PGP reference].

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

Based on the rules defined in Section 5.2.4, Section 5.2.3.8 and given presence of E-W links, RIFT can provide a one-hop protection of nodes that lost all their northbound links or in other complex link set failure scenarios except at Top-of-Fabric where the links are used exclusively to flood topology information in multi-plane designs. Section 6.4 explains the resulting behavior based on one such example.

5.4. Security

5.4.1. Security Model

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

The most security conscious operators will want to have full control over which port on which router/switch is connected to the respective port on the "other side", which we will call the "port-association model" (PAM) achievable e.g. by configuring on each port pair a designated shared key or pair of private/public keys. In secure data center locations, operators may want to control which router/switch is connected to which other router/switch only or choose a "node-association model" (NAM) which allows, for example, simplified port sparing. In an even more relaxed environment, an operator may only be concerned that the router/switches share credentials ensuring that they belong to this particular data center network hence allowing the flexible sparing of whole routers/switches. We will define that case as the "fabric-association model" (FAM), equivalent to using a shared secret for the whole fabric. Such flexibility may make sense for leaf nodes such as servers where the addition and swapping of servers is more frequent than the rest of the data center network. Generally, leafs of the fabric tend to be less trusted than switches. The different models could be mixed throughout the fabric if the benefits outweigh the cost of increased complexity in provisioning.

In each of the above cases, some configuration mechanism is needed to allow the operator to specify which connections are allowed, and some mechanism is needed to:

a. specify the according level in the fabric,

b. discover and report missing connections,

c. discover and report unexpected connections, and prevent such adjacencies from forming.

On the more relaxed configuration side of the spectrum, operators might only configure the level of each switch, but don't explicitly configure which connections are allowed. In this case, RIFT will only allow adjacencies to come up between nodes are that in adjacent levels. The operators with lowest security requirements may not use any configuration to specify which connections are allowed. Such fabrics could rely fully on ZTP for each router/switch to discover its level and would only allow adjacencies between adjacent levels to come up. Figure 30 illustrates the tradeoffs inherent in the different security models.

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

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

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

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

c. each pair of levels or

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

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

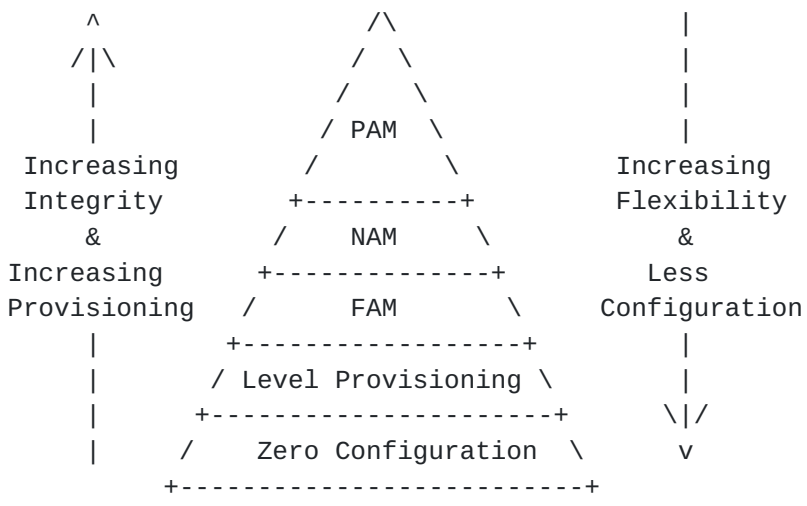

Figure 30: Security Model

5.4.2. Security Mechanisms

RIFT Security goals are to ensure authentication, message integrity and prevention of replay attacks. Low processing overhead and efficient messaging are also a goal. 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 it specifies how the key can be used to achieve goals.

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

5.4.3. Security Envelope

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

Local configuration MAY allow to skip the checking of the envelope's integrity.

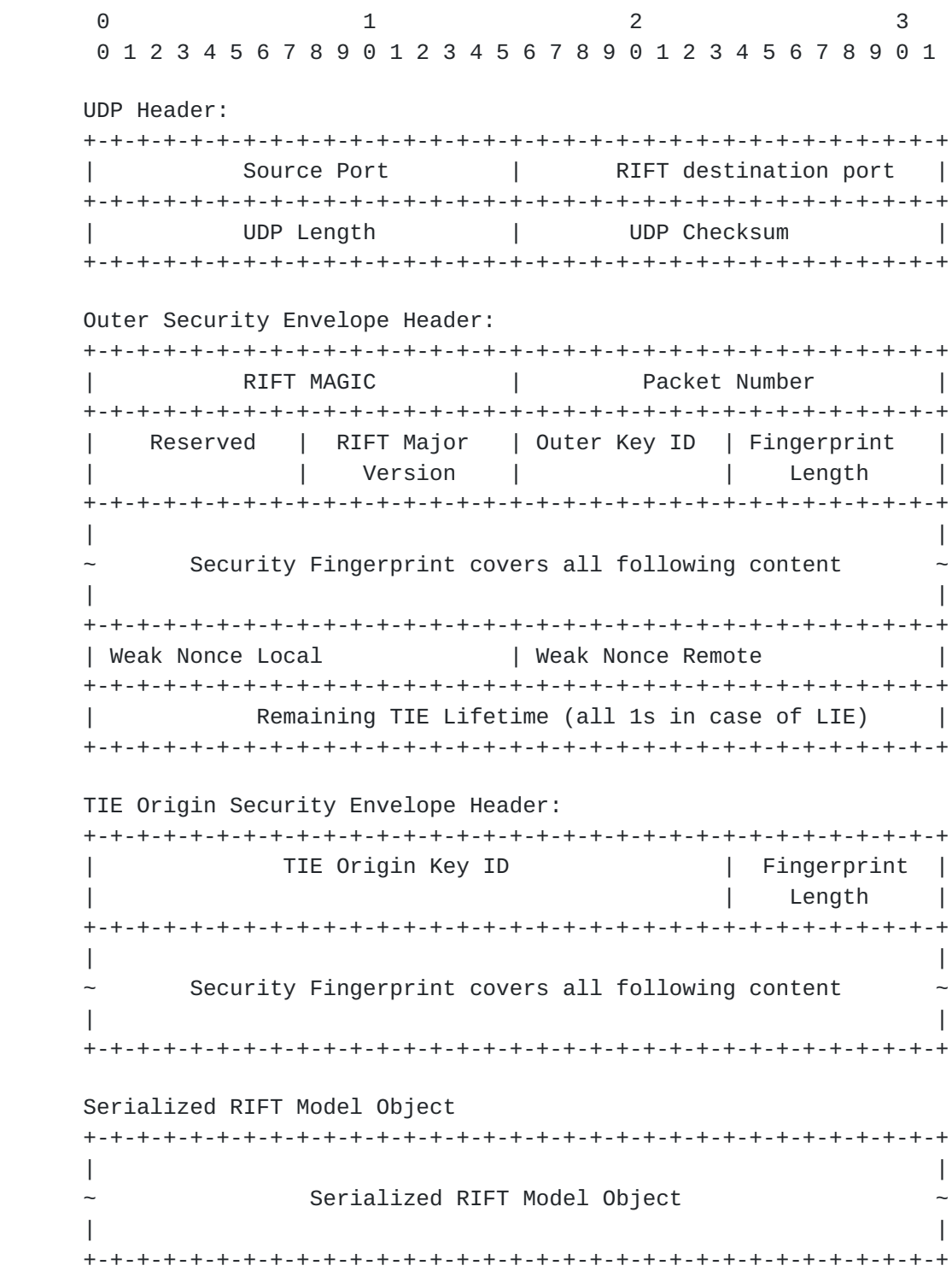

Figure 31: Security Envelope

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

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

RIFT Major Version: 8 bits. It allows to check whether protocol versions are compatible, i.e. the serialized object can be decoded at all. An implementation MUST drop packets with unexpected value and MAY report a problem. Must be same as in encoded model object, otherwise packet is dropped.

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

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

Length of Fingerprint: 8 bits. Length in 32-bit multiples of the following fingerprint not including lifetime or weak nonces. It allows to navigate the structure when an unknown key type is present. To clarify a common cornercase when this value is set to 0 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 signficant bits of fingerprint are fewer than the 32 bits padded length than the signficant bits MUST be left aligned and remaining bits on the right padded with 0s. When using PKI the Security fingerprint originating node uses its private key to create the signature. The original packet can then be verified provided the public key is shared and current.

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

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

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

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

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

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

5.4.4. Weak Nonces

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

Any implementation including RIFT security MUST generate and wrap around local nonces properly. When a nonce increment leads to `undefined_nonce` value the value SHOULD be incremented again immediately. All implementation MUST reflect the neighbor's nonces. An implementation SHOULD increment a chosen nonce on every LIE FSM transition that ends up in a different state from the previous and MUST increment its nonce at least every 5 minutes (such considerations allow for efficient implementations without opening a significant security risk). When flooding TIEs, the implementation MUST use recent (i.e. within allowed difference) nonces reflected in the LIE exchange. The schema specifies maximum allowable nonce value difference on a packet compared to reflected nonces in the LIEs. Any packet received with nonces deviating more than the allowed delta MUST be discarded without further computation of signatures to prevent computation load attacks.

In case where a secure implementation does not receive signatures or receives undefined nonces from neighbor indicating that it does not support or verify signatures, it is a matter of local policy how such packets are treated. Any secure implementation may choose to either refuse forming an adjacency with an implementation not advertising signatures or valid nonces or simply keep on signing local packets while accepting neighbor's packets without further security verification.

As a necessary exception, an implementation MUST advertise `undefined_nonce` for remote nonce value when the FSM is not in 2-way or 3-way state and accept an `undefined_nonce` for its local nonce value on packets in any other state than 3-way.

As optional optimization, an implemenation MAY send one LIE with previously negotiated neighbor's nonce to try to speed up a neighbor's transition from 3-way to 1-way and MUST revert to sending `undefined_nonce` after that.

5.4.5. Lifetime

Protecting lifetime on flooding may lead to excessive number of security fingerprint computation and hence an application generating such fingerprints on TIEs MAY round the value down to the next `rounddown_lifetime_interval` defined in the schema when sending TIEs albeit such optimization in presence of security hashes over advancing weak nonces may not be feasible.

5.4.6. Key Management

As outlined in the Security Model a private shared key or a public/private key pair is used to Authenticate the adjacency. The actual method of key distribution and key synchronization is assumed to be out of band from RIFT's perspective. Both nodes in the adjacency must share the same keys and configuration of key type and algorithm for a key ID. Mismatched keys will obviously not inter-operate due to unverifiable security envelope.

Key roll-over while the adjacency is active is allowed and the technique is well known and described in e.g. [RFC6518]. Key distribution procedures are out of scope for RIFT.

5.4.7. Security Association Changes

There in no mechanism to convert a security envelope for the same key ID from one algorithm to another once the envelope is operational. The recommended procedure to change to a new algorithm is to take the adjacency down and make the changes and then bring the adjacency up.

Obviously, an implementation may choose to stop verifying security envelope for the duration of key change to keep the adjacency up but since this introduces a security vulnerability window, such roll-over is not recommended.

6. Examples

6.1. Normal Operation

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

As first step, the following bi-directional adjacencies will be created (and any other links that do not fulfill LIE rules in Section 5.2.2 disregarded):

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

2. Spine 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

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

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

At this point both Spine 21 and Spine 22, as well as any controller to which they are connected, would have the complete network topology. At the same time, Spine 111/112/121/122 hold only the

N-ties of level 0 of their respective PoD. Leafs hold only their own N-TIEs.

S-TIEs with adjacencies and a default IP prefix would then be originated by Spine 21 and Spine 22 and each would be flooded to Spine 111, Spine 112, Spine 121, and Spine 122. Spine 111, Spine 112, Spine 121, and Spine 122 would each send the S-TIE from Spine 21 to Spine 22 and the S-TIE from Spine 22 to Spine 21. (S-TIEs are reflected up to level from which they are received but they are NOT propagated southbound.)

A S-TIE with a default IP prefix would be originated by Node 111 and Spine 112 and each would be sent to Leaf 111 and Leaf 112.

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

6.2. Leaf Link Failure

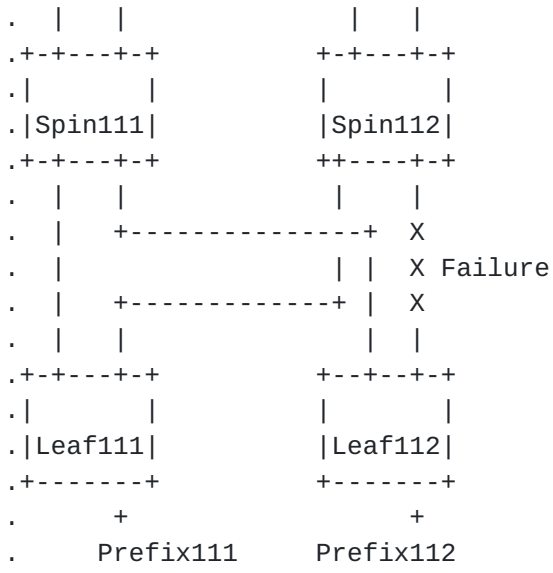

Figure 32: Single Leaf link failure

In case of a failing leaf link between spine 112 and leaf 112 the link-state information will cause re-computation of the necessary SPF and the higher levels will stop forwarding towards prefix 112 through spine 112. Only spines 111 and 112, as well as both spines will see control traffic. Leaf 111 will receive a new S-TIE from spine 112

and reflect back to spine 111. Spine 111 will de-aggregate prefix 111 and prefix 112 but we will not describe it further here since de-aggregation is emphasized in the next example. It is worth observing however in this example that if leaf 111 would keep on forwarding traffic towards prefix 112 using the advertised south-bound default of spine 112 the traffic would end up on Top-of-Fabric 21 and ToF 22 and cross back into pod 1 using spine 111. This is arguably not as bad as black-holing present in the next example but clearly undesirable. Fortunately, de-aggregation prevents this type of behavior except for a transitory period of time.

6.3. Partitioned Fabric

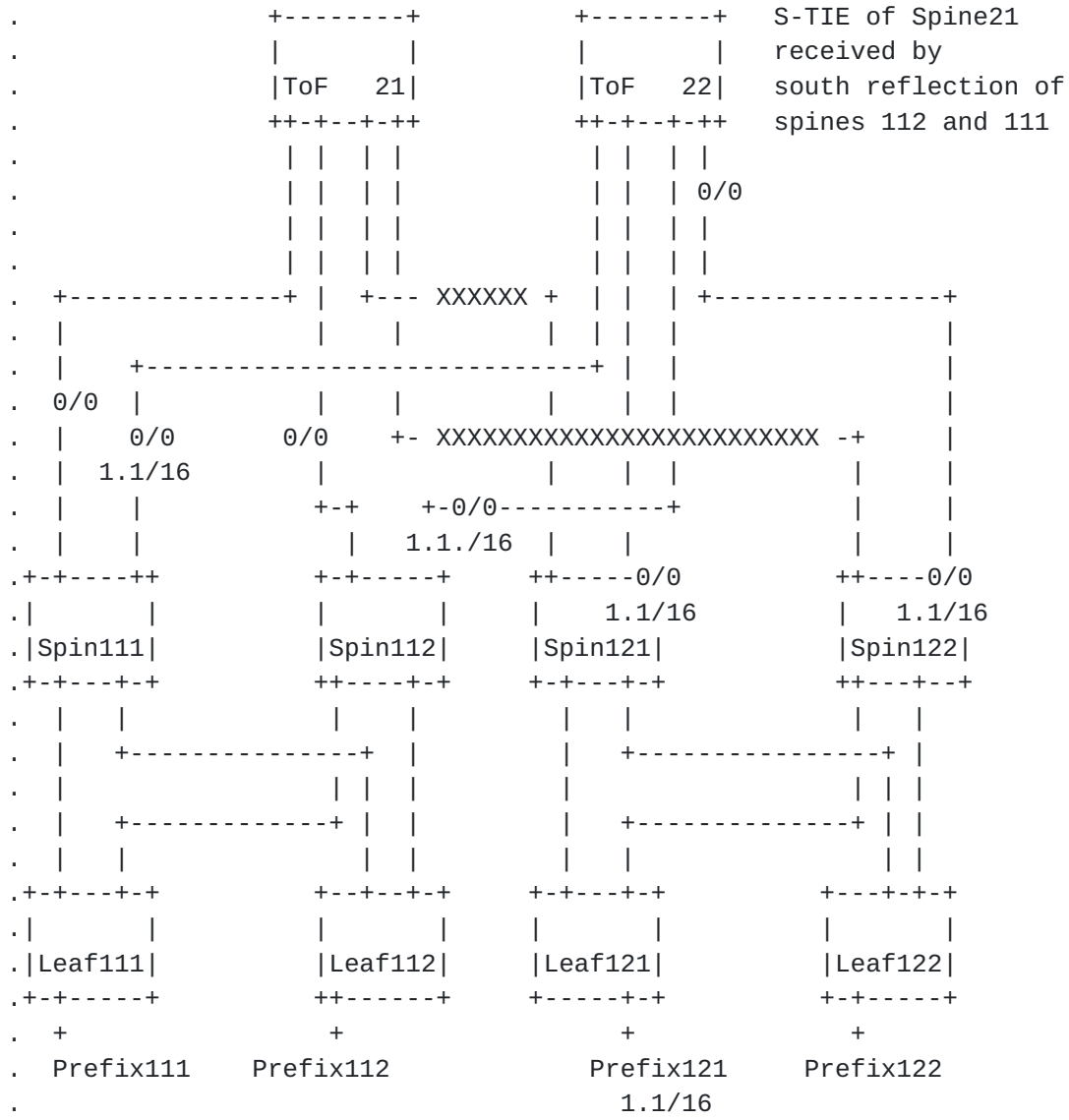

Figure 33: Fabric partition

Figure 33 shows the arguably a more catastrophic but also a more interesting case. Spine 21 is completely severed from access to Prefix 121 (we use in the figure 1.1/16 as example) by double link failure. However unlikely, if left unresolved, forwarding from leaf 111 and leaf 112 to prefix 121 would suffer 50% black-holing based on pure default route advertisements by Top-of-Fabric 21 and ToF 22.

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

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

To finish with an example of the resulting sets computed using notation introduced in Section 5.2.5, Top-of-Fabric 22 constructs the following sets:

|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 Spine 21) = Spine 111, Spine 112

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

6.4. Northbound Partitioned Router and Optional East-West Links

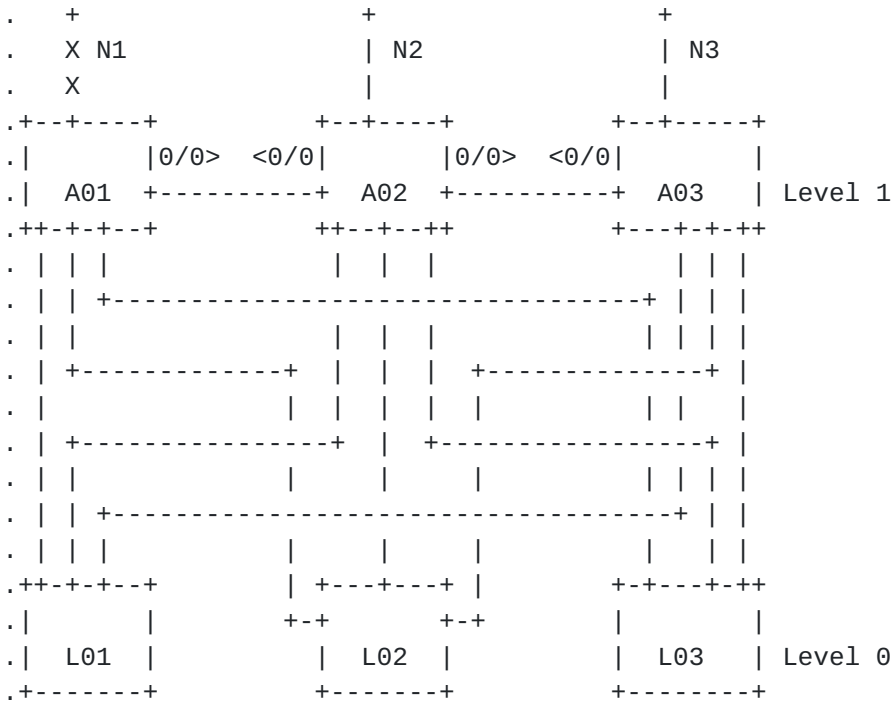

Figure 34: North Partitioned Router

Figure 34 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 5.2.4.1 A01 will compute northbound reachability by using the link A01 to A02 (whereas A02 will NOT use this link during N-SPF). Hence A01 will still advertise the default towards level 0 and route unidirectionally using the horizontal link.

As further consideration, the moment A02 looses link N2 the situation evolves again. A01 will have no more northbound reachability while still seeing A03 advertising northbound adjacencies in its south node tie. With that it will stop advertising a default route due to Section 5.2.3.8.

7. Implementation and Operation: Further Details

7.1. 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 leafs only, it is worth to observe that a leaf only version is significantly simpler to implement and requires much less resources:

1. Under normal conditions, the leaf needs to support a multipath default route only. In most catastrophic partitioning case it has to be capable of accommodating all the leaf routes in its own PoD to prevent black-holing.

2. Leaf nodes hold only their own N-TIEs and S-TIEs of Level 1 nodes they are connected to; so overall few in numbers.

3. Leaf node does not have to support any type of de-aggregation computation or propagation.

4. Leaf nodes do not have to support overload bit normally.

5. Unless optional leaf-2-leaf procedures are desired default route origination and S-TIE origination is unnecessary.

7.2. Considerations for Spine Implementation

In case of spines, i.e. nodes that will never act as Top of Fabric a full implementation is not required, specifically the node does not need to perform any computation of negative disaggregation except respecting northbound disaggregation advertised from the north.

7.3. Adaptations to Other Proposed Data Center Topologies

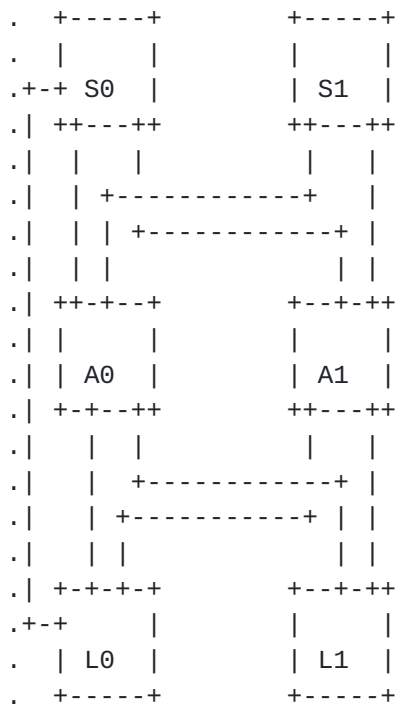

Figure 35: Level Shortcut

Strictly speaking, RIFT is not limited to Clos variations only. The protocol preconditions only a sense of 'compass rose direction' achieved by configuration (or derivation) of levels and other topologies are possible within this framework. So, conceptually, one could include leaf to leaf links and even shortcut between levels but certain requirements in Section 4 will not be met anymore. As an example, shortcutting levels illustrated in Figure 35 will lead either to suboptimal routing when L0 sends traffic to L1 (since using S0's default route will lead to the traffic being sent back to A0 or A1) or the leafs need each other's routes installed to understand that only A0 and A1 should be used to talk to each other.

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

7.4. Originating Non-Default Route Southbound

Obviously, an implementation may choose to originate southbound instead of a strict default route (as described in Section 5.2.3.8) a shorter prefix P' but in such a scenario all addresses carried within the RIFT domain must be contained within P'.

8. Security Considerations

8.1. General

One can consider attack vectors where a router may reboot many times while changing its system ID and pollute the network with many stale TIEs or TIEs are sent with very long lifetimes and not cleaned up when the routes vanishes. Those attack vectors are not unique to RIFT. Given large memory footprints available today those attacks should be relatively benign. Otherwise a node SHOULD implement a strategy of discarding contents of all TIEs that were not present in the SPF tree over a certain, configurable period of time. Since the protocol, like all modern link-state protocols, is self-stabilizing and will advertise the presence of such TIEs to its neighbors, they can be re-requested again if a computation finds that it sees an adjacency formed towards the system ID of the discarded TIEs.

8.2. ZTP

Section 5.2.7 presents many attack vectors in untrusted environments, starting with nodes that oscillate their level offers to the possiblity of a node offering a three way adjacency with the highest possible level value with a very long holdtime trying to put itself "on top of the lattice" and with that gaining access to the whole southbound topology. Session authentication mechanisms are necessary

in environments where this is possible and RIFT provides the according security envelope to ensure this if desired.

8.3. Lifetime

Traditional IGP protocols are vulnerable to lifetime modification and replay attacks that can be somewhat mitigated by using techniques like [RFC7987]. RIFT removes this attack vector by protecting the lifetime behind a signature computed over it and additional nonce combination which makes even the replay attack window very small and for practical purposes irrelevant since lifetime cannot be artificially shortened by the attacker.

8.4. Packet Number

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

8.5. Outer Fingerprint Attacks

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

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

8.6. TIE Origin Fingerprint DoS Attacks

A compromised node can attempt to generate "fake TIEs" using other nodes' TIE origin key identifiers. Albeit the ultimate validation of the origin fingerprint will fail in such scenarios and not progress further than immediately peering nodes, the resulting denial of

service attack seems unavoidable since the TIE origin key id is only protected by the, here assumed to be compromised, node.

8.7. Host Implementations

It can be reasonably expected that with the proliferation of RotH servers, rather than dedicated networking devices, will constitute 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 addressed within the protocol if the trust model is breached, i.e. the server presents valid credentials to form an adjacency and issue TIEs. However, in a move devious way, the servers can present DoS (or even DDos) vectors of issuing too many LIE packets, flood large amount of N-TIEs and similar anomalies. A prudent implementation hosting leafs should implement thresholds and raise warnings when leaf is advertising number of TIEs in excess of those.

9. 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.

9.1. Requested Multicast and Port Numbers

This document requests allocation in the 'IPv4 Multicast Address Space' registry the suggested value of 224.0.0.120 as 'ALL_V4_RIFT_ROUTERS' and in the 'IPv6 Multicast Address Space' registry the suggested value of FF02::A1F7 as 'ALL_V6_RIFT_ROUTERS'.

This document requests allocation in the 'Service Name and Transport Protocol Port Number Registry' the allocation of a suggested value of 914 on udp for 'RIFT_LIES_PORT' and suggested value of 915 for 'RIFT_TIES_PORT'.

9.2. Requested Registries with Suggested Values

This section requests registries that help govern the schema via usual IANA registry procedures. Allocation of new values is always performed via `Expert Review` action. IANA is requested to store the schema version introducing the allocated value as well as, optionally, its description when present. All values not suggested as to be considered `Unassigned`. The range of every registry is a 16-bit integer.

9.2.1. RIFT/common/AddressFamilyType

address family

9.2.1.1. Requested Entries

Name	Value	Schema Version	Description
Illegal	0	2.0	
AddressFamilyMinValue	1	2.0	
IPv4	2	2.0	
IPv6	3	2.0	
AddressFamilyMaxValue	4	2.0	

9.2.2. RIFT/common/HierarchyIndications

flags indicating nodes behavior in case of ZTP

9.2.2.1. Requested Entries

Name	Value	Schema Version	Description
leaf_only	0	2.0	
leaf_only_and_leaf_2_leaf_procedures	1	2.0	
top_of_fabric	2	2.0	

9.2.3. RIFT/common/IEEE802_1ASTimeStampType

timestamp per IEEE 802.1AS, values MUST be interpreted in implementation as unsigned

9.2.3.1. Requested Entries

Name	Value	Schema Version	Description
AS_sec	1	2.0	
AS_nsec	2	2.0	

9.2.4. RIFT/common/IPAddressType

IP address type

9.2.4.1. Requested Entries

Name	Value	Schema Version	Description
ipv4address	1	2.0	
ipv6address	2	2.0	

9.2.5. RIFT/common/IPPrefixType

prefix representing reachablity.

@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.

9.2.5.1. Requested Entries

Name	Value	Schema Version	Description
ipv4prefix	1	2.0	
ipv6prefix	2	2.0	

9.2.6. RIFT/common/IPv4PrefixType

IP v4 prefix type

9.2.6.1. Requested Entries

Name	Value	Schema Version	Description
address	1	2.0	
prefixlen	2	2.0	

9.2.7. RIFT/common/IPv6PrefixType

IP v6 prefix type

9.2.7.1. Requested Entries

Name	Value	Schema Version	Description
address	1	2.0	
prefixlen	2	2.0	

9.2.8. RIFT/common/PrefixSequenceType

sequence of a prefix when it moves

9.2.8.1. Requested Entries

Name	Value	Schema Version	Description
timestamp	1	2.0	
transactionid	2	2.0	transaction ID set by client in e.g. in 6LoWPAN

9.2.9. RIFT/common/RouteType

RIFT route types.

@note: route types which MUST be ordered on their preference PGP prefixes are most preferred attracting traffic north (towards spine) and then south normal prefixes are attracting traffic south (towards leafs), i.e. prefix in NORTH PREFIX TIE is preferred over SOUTH PREFIX TIE.

@note: The only purpose of those values is to introduce an ordering whereas an implementation can choose internally any other values as long the ordering is preserved

9.2.9.1. Requested Entries

Name	Value	Schema Version	Description
Illegal	0	2.0	
RouteTypeMinValue	1	2.0	
Discard	2	2.0	
LocalPrefix	3	2.0	
SouthPGPPrefix	4	2.0	
NorthPGPPrefix	5	2.0	
NorthPrefix	6	2.0	
NorthExternalPrefix	7	2.0	
SouthPrefix	8	2.0	
SouthExternalPrefix	9	2.0	
NegativeSouthPrefix	10	2.0	
RouteTypeMaxValue	11	2.0	

9.2.10. RIFT/common/TIETypeType

type of TIE.

This enum indicates what TIE type the TIE is carrying. In case the value is not known to the receiver, re-flooded the same way as prefix TIEs. This allows for future extensions of the protocol within the same schema major with types opaque to some nodes unless the flooding scope is not the same as prefix TIE, then a major version revision MUST be performed.

9.2.10.1. Requested Entries

Name	Value	Schema Version	Description
Illegal	0	2.0	
TIETypeMinValue	1	2.0	
NodeTIEType	2	2.0	
PrefixTIEType	3	2.0	
PositiveDisaggregationPrefixTIEType	4	2.0	
NegativeDisaggregationPrefixTIEType	5	2.0	
PGPrefixTIEType	6	2.0	
KeyValueTIEType	7	2.0	
ExternalPrefixTIEType	8	2.0	
PositiveExternalDisaggregationPrefixTIEType	9	2.0	
TIETypeMaxValue	10	2.0	

9.2.11. RIFT/common/TieDirectionType

direction of tie

9.2.11.1. Requested Entries

Name	Value	Schema Version	Description
Illegal	0	2.0	
South	1	2.0	
North	2	2.0	
DirectionMaxValue	3	2.0	

9.2.12. RIFT/encoding/Community

community

9.2.12.1. Requested Entries

Name	Value	Schema Version	Description
top	1	2.0	
bottom	2	2.0	

9.2.13. RIFT/encoding/KeyValueTIEElement

Generic key value pairs

9.2.13.1. Requested Entries

Name	Value	Schema Version	Description
keyvalues	1	2.0	if the same key repeats in multiple TIEs of same node or with different values, behavior is unspecified

9.2.14. RIFT/encoding/LIEPacket

RIFT LIE packet

@note this node's level is already included on the packet header

9.2.14.1. Requested Entries

Name	Value	Schema Version	Description
name	1	2.0	node or adjacency name
local_id	2	2.0	local link ID
flood_port	3	2.0	UDP port to which we can receive flooded TIEs
link_mtu_size	4	2.0	layer 3 MTU, used to discover to mismatch.
link_bandwidth	5	2.0	local link bandwidth on the interface
neighbor	6	2.0	reflects the neighbor once received to provide 3-way connectivity
pod	7	2.0	node's PoD
node_capabilities	10	2.0	node capabilities shown in the LIE. The capabilies MUST match the capabilities shown in the Node TIEs, otherwise the behavior is unspecified. A node detecting the mismatch SHOULD generate according error
link_capabilities	11	2.0	capabilities of this link
holdtime	12	2.0	required holdtime of the adjacency, i.e. how much time MUST expire without LIE for the adjacency to drop
label	13	2.0	unsolicited, downstream assigned locally significant label value for the adjacency
not_a_ztp_offer	21	2.0	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	2.0	indicates to northbound neighbor that it should be reflooding this node's

			N-TIEs to achieve flood reduction and balancing for northbound flooding. To be ignored if received from a northbound adjacency
you_are_sending_too_quickly	23	2.0	can be optionally set to indicate to neighbor that packet losses are seen on reception based on packet numbers or the rate is too high. The receiver SHOULD temporarily slow down flooding rates
instance_name	24	2.0	instance name in case multiple RIFT instances running on same interface

9.2.15. RIFT/encoding/LinkCapabilities

link capabilities

9.2.15.1. Requested Entries

Name	Value	Schema Version	Description
bfd	1	2.0	indicates that the link is supporting BFD
v4_forwarding_capable	2	2.0	indicates whether the interface will support v4 forwarding. This MUST be set to true when LIEs from a v4 address are sent and MAY be set to true in LIEs on v6 address. If v4 and v6 LIEs indicate contradicting information the behavior is unspecified.

9.2.16. RIFT/encoding/LinkIDPair

LinkID pair describes one of parallel links between two nodes

9.2.16.1. Requested Entries

Name	Value	Schema Version	Description
local_id	1	2.0	node-wide unique value for the local link
remote_id	2	2.0	received remote link ID for this link
platform_interface_index	10	2.0	describes the local interface index of the link
platform_interface_name	11	2.0	describes the local interface name
trusted_outer_security_key	12	2.0	indication whether the link is secured, i.e. protected by outer key, absence of this element means no indication, undefined outer key means not secured
bfd_up	13	2.0	indication whether the link is protected by established BFD session

9.2.17. RIFT/encoding/Neighbor

neighbor structure

9.2.17.1. Requested Entries

Name	Value	Schema Version	Description
originator	1	2.0	system ID of the originator
remote_id	2	2.0	ID of remote side of the link

9.2.18. RIFT/encoding/NodeCapabilities

capabilities the node supports. The schema may add to this field future capabilities to indicate whether it will support interpretation of future schema extensions on the same major revision. Such fields MUST be optional and have an implicit or explicit false default value. If a future capability changes route selection or generates blackholes if some nodes are not supporting it then a major version increment is unavoidable.

9.2.18.1. Requested Entries

Name	Value	Schema Version	Description
protocol_minor_version	1	2.0	must advertise supported minor version dialect that way
flood_reduction	2	2.0	can this node participate in flood reduction
hierarchy_indications	3	2.0	does this node restrict itself to be top-of-fabric or leaf only (in ZTP) and does it support leaf-2-leaf procedures

9.2.19. RIFT/encoding/NodeFlags

Flags the node sets

9.2.19.1. Requested Entries

Name	Value	Schema Version	Description
overload	1	2.0	indicates that node is in overload, do not transit traffic through it

9.2.20. RIFT/encoding/NodeNeighborsTIEElement

neighbor of a node

9.2.20.1. Requested Entries

Name	Value	Schema Version	Description
level	1	2.0	level of neighbor
cost	3	2.0	
link_ids	4	2.0	can carry description of multiple parallel links in a TIE
bandwidth	5	2.0	total bandwith to neighbor, this will be normally sum of the bandwidths of all the parallel links.

9.2.21. RIFT/encoding/NodeTIEElement

Description of a node.

It may occur multiple times in different TIEs but if either * capabilities values do not match or * flags values do not match or * neighbors repeat with different values

the behavior is undefined and a warning SHOULD be generated. Neighbors can be distributed across multiple TIEs however if the sets

are disjoint. Miscablings SHOULD be repeated in every node TIE, otherwise the behavior is undefined.

@note: observe that absence of fields implies defined defaults

9.2.21.1. Requested Entries

Name	Value	Schema Version	Description
level	1	2.0	level of the node
neighbors	2	2.0	node's neighbors. If neighbor systemID repeats in other node TIEs of same node the behavior is undefined
capabilities	3	2.0	capabilities of the node
flags	4	2.0	flags of the node
name	5	2.0	optional node name for easier operations
pod	6	2.0	PoD to which the node belongs
miscabled_links	10	2.0	if any local links are miscabled, the indication is flooded

9.2.22. RIFT/encoding/PacketContent

content of a RIFT packet

9.2.22.1. Requested Entries

Name	Value	Schema Version	Description
lie	1	2.0	
tide	2	2.0	
tire	3	2.0	
tie	4	2.0	

9.2.23. RIFT/encoding/PacketHeader

common RIFT packet header

9.2.23.1. Requested Entries

Name	Value	Schema Version	Description
major_version	1	2.0	major version type of protocol
minor_version	2	2.0	minor version type of protocol
sender	3	2.0	node sending the packet, in case of LIE/TIRE/TIDE also the originator of it
level	4	2.0	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.

9.2.24. RIFT/encoding/PrefixAttributes

9.2.24.1. Requested Entries

Name	Value	Schema Version	Description
metric	2	2.0	distance of the prefix
tags	3	2.0	generic unordered set of route tags, can be redistributed to other protocols or use within the context of real time analytics
monotonic_clock	4	2.0	monotonic clock for mobile addresses
loopback	6	2.0	indicates if the interface is a node loopback
directly_attached	7	2.0	indicates that the prefix is directly attached, i.e. should be routed to even if the node is in overload. *
from_link	10	2.0	in case of locally originated prefixes, i.e. interface addresses this can describe which link the address belongs to.

9.2.25. RIFT/encoding/PrefixTIEElement

TIE carrying prefixes

9.2.25.1. Requested Entries

Name	Value	Schema Version	Description
prefixes	1	2.0	prefixes with the associated attributes. if the same prefix repeats in multiple TIEs of same node behavior is unspecified

9.2.26. RIFT/encoding/ProtocolPacket

RIFT packet structure

9.2.26.1. Requested Entries

Name	Value	Schema Version	Description
header	1	2.0	
content	2	2.0	

9.2.27. RIFT/encoding/TIDEPacket

TIDE with sorted TIE headers, if headers are unsorted, behavior is undefined

9.2.27.1. Requested Entries

Name	Value	Schema Version	Description
start_range	1	2.0	first TIE header in the tide packet
end_range	2	2.0	last TIE header in the tide packet
headers	3	2.0	_sorted_ list of headers

9.2.28. RIFT/encoding/TIEElement

single element in a TIE. enum `common.TIETypeType` in TIEID indicates which elements MUST be present in the TIEElement. In case of mismatch the unexpected elements MUST be ignored. In case of lack of expected element the TIE an error MUST be reported and the TIE MUST be ignored.

This type can be extended with new optional elements for new `common.TIETypeType` values without breaking the major but if it is necessary to understand whether all nodes support the new type a node capability must be added as well.

9.2.28.1. Requested Entries

Name	Value	Schema Version	Description
node	1	2.0	used in case of enum common.TIETypeType.NodeTIEType
prefixes	2	2.0	used in case of enum common.TIETypeType.PrefixTIEType
positive_disaggregation_prefixes	3	2.0	positive prefixes (always southbound) It MUST NOT be advertised within a North TIE and ignored otherwise
negative_disaggregation_prefixes	5	2.0	transitive, negative prefixes (always southbound) which MUST be aggregated and propagated according to the specification southwards towards lower levels to heal pathological upper level partitioning, otherwise blackholes may occur in multiplane fabrics. It MUST NOT be advertised within a North TIE.
external_prefixes	6	2.0	externally reimported prefixes
positive_external_disaggregation_prefixes	7	2.0	positive external disaggregated prefixes (always southbound). It MUST NOT be advertised within a North TIE and ignored otherwise
keyvalues	9	2.0	Key-Value store elements

9.2.29. RIFT/encoding/TIEHeader

Header of a TIE.

@note: TIEID space is a total order achieved by comparing the elements in sequence defined and comparing each value as an unsigned integer of according length.

@note: After sequence number the lifetime received on the envelope must be used for comparison before further fields.

@note: `origination_time` and `origination_lifetime` are disregarded for comparison purposes and carried purely for debugging/security purposes if present.

9.2.29.1. Requested Entries

Name	Value	Schema Version	Description
tieid	2	2.0	ID of the tie
seq_nr	3	2.0	sequence number of the tie
origination_time	10	2.0	absolute timestamp when the TIE was generated. This can be used on fabrics with synchronized clock to prevent lifetime modification attacks.
origination_lifetime	12	2.0	original lifetime when the TIE was generated. This can be used on fabrics with synchronized clock to prevent lifetime modification attacks.

9.2.30. RIFT/encoding/TIEHeaderWithLifeTime

Header of a TIE as described in TIRE/TIDE.

9.2.30.1. Requested Entries

Name	Value	Schema Version	Description
header	1	2.0	
remaining_lifetime	2	2.0	remaining lifetime that expires down to 0 just like in ISIS. TIEs with lifetimes differing by less than `lifetime_diff2ignore` MUST be considered EQUAL.

9.2.31. RIFT/encoding/TIEID

ID of a TIE

@note: TIEID space is a total order achieved by comparing the elements in sequence defined and comparing each value as an unsigned integer of according length.

9.2.31.1. Requested Entries

Name	Value	Schema Version	Description
direction	1	2.0	direction of TIE
originator	2	2.0	indicates originator of the TIE
tietype	3	2.0	type of the tie
tie_nr	4	2.0	number of the tie

9.2.32. RIFT/encoding/TIEPacket

TIE packet

9.2.32.1. Requested Entries

Name	Value	Schema Version	Description
header	1	2.0	
element	2	2.0	

9.2.33. RIFT/encoding/TIREPacket

TIRE packet

9.2.33.1. Requested Entries

Name	Value	Schema Version	Description
headers	1	2.0	

10. 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 or implementation. 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 epistomology that allowed to tie current asynchronous distributed systems theory results to a modern protocol design presented here. Adrian Farrel, Joel Halpern, Jeffrey Zhang, Krzysztof Szarkowicz, Nagendra Kumar provided thoughtful comments that improved the readability of the document and found good amount of corners where the light failed to shine. Kris Price was first to mention single router, single arm default considerations. Jeff Tantsura helped out with some initial thoughts on BFD interactions while Jeff Haas corrected several misconceptions about BFD's finer points. Artur Makutunowicz pointed out many possible improvements and acted as

sounding board in regard to modern protocol implementation techniques RIFT is exploring. Barak Gafni formalized first time clearly the problem of partitioned spine and fallen leafs on a (clean) napkin in Singapore that led to the very important part of the specification centered around multiple Top-of-Fabric planes and negative disaggregation. Igor Gashinsky and others shared many thoughts on problems encountered in design and operation of large-scale data center fabrics. Xu Benchong found a delicate error in the flooding procedures while implementing.

11. References

11.1. Normative References

[ISO10589] ISO "International Organization for Standardization", "Intermediate system to Intermediate system intra-domain routeing information exchange protocol for use in conjunction with the protocol for providing the connectionless-mode Network Service (ISO 8473), ISO/IEC 10589:2002, Second Edition.", Nov 2002.

[RFC1982] Elz, R. and R. Bush, "Serial Number Arithmetic", RFC 1982, DOI 10.17487/RFC1982, August 1996, <https://www.rfc-editor.org/info/rfc1982>.

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

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

[RFC2365] Meyer, D., "Administratively Scoped IP Multicast", BCP 23, RFC 2365, DOI 10.17487/RFC2365, July 1998, <https://www.rfc-editor.org/info/rfc2365>.

[RFC4271] Rekhter, Y., Ed., Li, T., Ed., and S. Hares, Ed., "A Border Gateway Protocol 4 (BGP-4)", RFC 4271, DOI 10.17487/RFC4271, January 2006, <https://www.rfc-editor.org/info/rfc4271>.

[RFC4291] Hinden, R. and S. Deering, "IP Version 6 Addressing Architecture", RFC 4291, DOI 10.17487/RFC4291, February 2006, <https://www.rfc-editor.org/info/rfc4291>.

[RFC5082] Gill, V., Heasley, J., Meyer, D., Savola, P., Ed., and C. Pignataro, "The Generalized TTL Security Mechanism (GTSM)", RFC 5082, DOI 10.17487/RFC5082, October 2007, <https://www.rfc-editor.org/info/rfc5082>.

[RFC5120] Przygienda, T., Shen, N., and N. Sheth, "M-ISIS: Multi Topology (MT) Routing in Intermediate System to Intermediate Systems (IS-ISs)", RFC 5120, DOI 10.17487/RFC5120, February 2008, <https://www.rfc-editor.org/info/rfc5120>.

[RFC5303] Katz, D., Saluja, R., and D. Eastlake 3rd, "Three-Way Handshake for IS-IS Point-to-Point Adjacencies", RFC 5303, DOI 10.17487/RFC5303, October 2008, <https://www.rfc-editor.org/info/rfc5303>.

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

[RFC5709] Bhatia, M., Manral, V., Fanto, M., White, R., Barnes, M., Li, T., and R. Atkinson, "OSPFv2 HMAC-SHA Cryptographic Authentication", RFC 5709, DOI 10.17487/RFC5709, October 2009, <https://www.rfc-editor.org/info/rfc5709>.

[RFC5881] Katz, D. and D. Ward, "Bidirectional Forwarding Detection (BFD) for IPv4 and IPv6 (Single Hop)", RFC 5881, DOI 10.17487/RFC5881, June 2010, <https://www.rfc-editor.org/info/rfc5881>.

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

[RFC7752] Gredler, H., Ed., Medved, J., Previdi, S., Farrel, A., and S. Ray, "North-Bound Distribution of Link-State and Traffic Engineering (TE) Information Using BGP", RFC 7752, DOI 10.17487/RFC7752, March 2016, <https://www.rfc-editor.org/info/rfc7752>.

[RFC7987] Ginsberg, L., Wells, P., Decraene, B., Przygienda, T., and H. Gredler, "IS-IS Minimum Remaining Lifetime", RFC 7987, DOI 10.17487/RFC7987, October 2016, <https://www.rfc-editor.org/info/rfc7987>.

[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>.

[RFC8402] Filsfils, C., Ed., Previdi, S., Ed., Ginsberg, L., Decraene, B., Litkowski, S., and R. Shakir, "Segment Routing Architecture", RFC 8402, DOI 10.17487/RFC8402, July 2018, <https://www.rfc-editor.org/info/rfc8402>.

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

[thrift] Apache Software Foundation, "Thrift Interface Description Language", <https://thrift.apache.org/docs/idl>.

11.2. Informative References

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

[DIJKSTRA] Dijkstra, E., "A Note on Two Problems in Connexion with Graphs", Journal Numer. Math. , 1959.

[DOT] Ellson, J. and L. Koutsofios, "Graphviz: open source graph drawing tools", Springer-Verlag , 2001.

[DYNAMO] De Candia et al., G., "Dynamo: amazon's highly available key-value store", ACM SIGOPS symposium on Operating systems principles (SOSP '07), 2007.

[EPPSTEIN] Eppstein, D., "Finding the k-Shortest Paths", 1997.

[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>.

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

[IEEEstd1588] IEEE, "IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems", IEEE Standard 1588, <https://ieeexplore.ieee.org/document/4579760/>.

[IEEEstd8021AS] IEEE, "IEEE Standard for Local and Metropolitan Area Networks - Timing and Synchronization for Time-Sensitive Applications in Bridged Local Area Networks", IEEE Standard 802.1AS, <https://ieeexplore.ieee.org/document/5741898/>.

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

[MAKSIC2013] Maksic et al., N., "Improving Utilization of Data Center Networks", IEEE Communications Magazine, Nov 2013.

[RFC0826] Plummer, D., "An Ethernet Address Resolution Protocol: Or Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware", STD 37, RFC 826, DOI 10.17487/RFC0826, November 1982, <https://www.rfc-editor.org/info/rfc826>.

[RFC2131] Droms, R., "Dynamic Host Configuration Protocol", RFC 2131, DOI 10.17487/RFC2131, March 1997, <https://www.rfc-editor.org/info/rfc2131>.

[RFC3626] Clausen, T., Ed. and P. Jacquet, Ed., "Optimized Link State Routing Protocol (OLSR)", RFC 3626, DOI 10.17487/RFC3626, October 2003, <https://www.rfc-editor.org/info/rfc3626>.

[RFC4655] Farrel, A., Vasseur, J., and J. Ash, "A Path Computation Element (PCE)-Based Architecture", RFC 4655, DOI 10.17487/RFC4655, August 2006, <https://www.rfc-editor.org/info/rfc4655>.

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

[RFC4862] Thomson, S., Narten, T., and T. Jinmei, "IPv6 Stateless Address Autoconfiguration", RFC 4862, DOI 10.17487/RFC4862, September 2007, <https://www.rfc-editor.org/info/rfc4862>.

[RFC6518] Lebovitz, G. and M. Bhatia, "Keying and Authentication for Routing Protocols (KARP) Design Guidelines", RFC 6518, DOI 10.17487/RFC6518, February 2012, <https://www.rfc-editor.org/info/rfc6518>.

[RFC7855] Previdi, S., Ed., Filsfils, C., Ed., Decraene, B., Litkowski, S., Horneffer, M., and R. Shakir, "Source Packet Routing in Networking (SPRING) Problem Statement and Requirements", RFC 7855, DOI 10.17487/RFC7855, May 2016, <https://www.rfc-editor.org/info/rfc7855>.

[RFC7938] Lapukhov, P., Premji, A., and J. Mitchell, Ed., "Use of BGP for Routing in Large-Scale Data Centers", RFC 7938, DOI 10.17487/RFC7938, August 2016, <https://www.rfc-editor.org/info/rfc7938>.

[RFC8415] Mrugalski, T., Siodelski, M., Volz, B., Yourtchenko, A., Richardson, M., Jiang, S., Lemon, T., and T. Winters, "Dynamic Host Configuration Protocol for IPv6 (DHCPv6)", RFC 8415, DOI 10.17487/RFC8415, November 2018, <https://www.rfc-editor.org/info/rfc8415>.

[VAHDAT08] Al-Fares, M., Loukissas, A., and A. Vahdat, "A Scalable, Commodity Data Center Network Architecture", SIGCOMM , 2008.

[Wikipedia] Wikipedia, "https://en.wikipedia.org/wiki/Serial_number_arithmetic", 2016.

Appendix A. Sequence Number Binary Arithmetic

The only reasonably reference to a cleaner than [RFC1982] sequence number solution is given in [Wikipedia]. It basically converts the problem into two complement's arithmetic. Assuming a straight two

complement's substractions on the bit-width of the sequence number the according >: and =: relations are defined as:

U_1, U_2 are 12-bits aligned unsigned version number

D_f is (U_1 - U_2) interpreted as two complement signed 12-bits
D_b is (U_2 - U_1) interpreted as two complement signed 12-bits

U_1 >: U_2 IIF D_f > 0 AND D_b < 0
U_1 =: U_2 IIF D_f = 0

The >: relationsship is symmetric but not transitive. Observe that this leaves the case of the numbers having maximum two complement distance, e.g. (0 and 0x800) undefined in our 12-bits case since D_f and D_b are both -0x7ff.

A simple example of the relationship in case of 3-bit arithmetic follows as table indicating D_f/D_b values and then the relationship of U_1 to U_2:

U2 / U1	0	1	2	3	4	5	6	7
0	+/+	+/-	+/-	+/-	-/-	-/+	-/+	-/+
1	-/+	+/+	+/-	+/-	+/-	-/-	-/+	-/+
2	-/+	-/+	+/+	+/-	+/-	+/-	-/-	-/+
3	-/+	-/+	-/+	+/+	+/-	+/-	+/-	-/-
4	-/-	-/+	-/+	-/+	+/+	+/-	+/-	+/-
5	+/-	-/-	-/+	-/+	-/+	+/+	+/-	+/-
6	+/-	+/-	-/-	-/+	-/+	-/+	+/+	+/-
7	+/-	+/-	+/-	-/-	-/+	-/+	-/+	+/+

U2 / U1	0	1	2	3	4	5	6	7
0	=	>	>	>	?	<	<	<
1	<	=	>	>	>	?	<	<
2	<	<	=	>	>	>	?	<
3	<	<	<	=	>	>	>	?
4	?	<	<	<	=	>	>	>
5	>	?	<	<	<	=	>	>
6	>	>	?	<	<	<	=	>
7	>	>	>	?	<	<	<	=

Appendix B. Information Elements Schema

This section introduces the schema for information elements. The IDL is Thrift [thrift].

On schema changes that

1. change field numbers or

2. add new *required* fields or

3. remove any fields or

4. change lists into sets, unions into structures or

5. change multiplicity of fields or

6. changes name of any field or type or

7. change datatypes 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.TIEType` (use of enumeration types is generally discouraged)

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

Observe however that introducing an optional field does not cause a major version increase even if the fields inside the structure are optional with defaults.

All signed integer as forced by Thrift [thrift] support must be cast for internal purposes to equivalent unsigned values without discarding the signedness bit. An implementation SHOULD try to avoid using the signedness bit when generating values.

The schema is normative.

B.1. common.thrift

```
/** @note MUST be interpreted in implementation as unsigned 64 bits.
 *        The implementation SHOULD NOT use the MSB.
 */
typedef i64      SystemIDType
typedef i32      IPv4Address
/** this has to be of length long enough to accomodate prefix */
typedef binary   IPv6Address
/** @note MUST be interpreted in implementation as unsigned */
typedef i16      UDPPortType
/** @note MUST be interpreted in implementation as unsigned */
typedef i32      TIENrType
```

```
/** @note MUST be interpreted in implementation as unsigned */
typedef i32      MTUSizeType
/** @note MUST be interpreted in implementation as unsigned rollling over
number */
typedef i16      SeqNrType
/** @note MUST be interpreted in implementation as unsigned */
typedef i32      LifeTimeInSecType
/** @note MUST be interpreted in implementation as unsigned */
typedef i8       LevelType
/** optional, recommended monotonically increasing number _per packet type per
adjacency_
    that can be used to detect losses/misordering/restarts.
    This will be moved into envelope in the future.
    @note MUST be interpreted in implementation as unsigned rollling over
number */
typedef i16      PacketNumberType
/** @note MUST be interpreted in implementation as unsigned */
typedef i32      PodType
/** @note MUST be interpreted in implementation as unsigned. This is carried in
the
          security envelope and MUST fit into 8 bits. */
typedef i8       VersionType
/** @note MUST be interpreted in implementation as unsigned */
typedef i16      MinorVersionType
/** @note MUST be interpreted in implementation as unsigned */
typedef i32      MetricType
/** @note MUST be interpreted in implementation as unsigned and unstructured */
typedef i64      RouteTagType
/** @note MUST be interpreted in implementation as unstructured label value */
typedef i32      LabelType
/** @note MUST be interpreted in implementation as unsigned */
typedef i32      BandwithInMegaBitsType
/** @note Key Value key ID type */
typedef string   KeyIDType
/** node local, unique identification for a link (interface/tunnel
  * etc. Basically anything RIFT runs on). This is kept
  * at 32 bits so it aligns with BFD [RFC5880] discriminator size.
  */
typedef i32    LinkIDType
typedef string KeyNameType
typedef i8     PrefixLenType
/** timestamp in seconds since the epoch */
typedef i64    TimestampInSecsType
/** security nonce.
 *  @note MUST be interpreted in implementation as rolling over unsigned value
*/
typedef i16    NonceType
/** LIE FSM holdtime type */
```

```
typedef i16    TimeIntervalInSecType
/** Transaction ID type for prefix mobility as specified by RFC6550,  value
    MUST be interpreted in implementation as unsigned  */
typedef i8     PrefixTransactionIDType
/** timestamp per IEEE 802.1AS, 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 nodes behavior in case of ZTP
 */
enum HierarchyIndications {
    /** forces level to `leaf_level` and enables according procedures */
    leaf_only                           = 0,
    /** forces level to `leaf_level` and enables according procedures */
    leaf_only_and_leaf_2_leaf_procedures = 1,
    /** forces level to `top_of_fabric` and enables according procedures */
    top_of_fabric                       = 2,
}

const PacketNumberType  undefined_packet_number    = 0
/** This MUST be used when node is configured as top of fabric in ZTP.
    This is kept reasonably low to alow for fast ZTP convergence on
    failures. */
const LevelType   top_of_fabric_level              = 24
/** default bandwidth on a link */
const BandwithInMegaBitsType  default_bandwidth    = 100
/** fixed leaf level when ZTP is not used */
const LevelType   leaf_level                   = 0
const LevelType   default_level                = leaf_level
const PodType     default_pod                  = 0
const LinkIDType  undefined_linkid             = 0

/** default distance used */
const MetricType  default_distance         = 1
/** any distance larger than this will be considered infinity */
const MetricType  infinite_distance        = 0x7FFFFFFF
/** represents invalid distance */
const MetricType  invalid_distance         = 0
const bool overload_default                = false
const bool flood_reduction_default         = true
/** default LIE FSM holddown time */
const TimeIntervalInSecType   default_lie_holdtime  = 3
/** default ZTP FSM holddown time */
const TimeIntervalInSecType   default_ztp_holdtime  = 1
/** by default LIE levels are ZTP offers */
```

```
const bool default_not_a_ztp_offer        = false
```

```
/** by default e'one is repeating flooding */
const bool default_you_are_flood_repeater = true
/** 0 is illegal for SystemID */
const SystemIDType IllegalSystemID        = 0
/** empty set of nodes */
const set<SystemIDType> empty_set_of_nodeids = {}
/** default lifetime of TIE is one week */
const LifeTimeInSecType default_lifetime      = 604800
/** default lifetime when TIEs are purged is 5 minutes */
const LifeTimeInSecType purge_lifetime        = 300
/** round down interval when TIEs are sent with security hashes
    to prevent excessive computation. **/
const LifeTimeInSecType rounddown_lifetime_interval = 60
/** any `TieHeader` that has a smaller lifetime difference
    than this constant is equal (if other fields equal). This
    constant MUST be larger than `purge_lifetime` to avoid
    retransmissions */
const LifeTimeInSecType lifetime_diff2ignore  = 400

/** default UDP port to run LIEs on */
const UDPPortType     default_lie_udp_port       =  914
/** default UDP port to receive TIEs on, that can be peer specific */
const UDPPortType     default_tie_udp_flood_port =  915

/** default MTU link size to use */
const MTUSizeType     default_mtu_size           = 1400
/** default link being BFD capable */
const bool            bfd_default                = true

/** undefined nonce, equivalent to missing nonce */
const NonceType       undefined_nonce            = 0;
/** outer security key id, MUST be interpreted as in implementation as unsigned
*/
typedef i8            OuterSecurityKeyID
/** security key id, MUST be interpreted as in implementation as unsigned */
typedef i32           TIESecurityKeyID
/** undefined key */
const TIESecurityKeyID undefined_securitykey_id   = 0;
/** Maximum delta (negative or positive) that a mirrored nonce can
    deviate from local value to be considered valid. If nonces are
    changed every minute on both sides this opens statistically
    a `maximum_valid_nonce_delta` minutes window of identical LIEs,
    TIE, TI(x)E replays.
    The interval cannot be too small since LIE FSM may change
    states fairly quickly during ZTP without sending LIEs*/
const i16             maximum_valid_nonce_delta  = 5;

/** direction of tie */
```

```
enum TieDirectionType {
```

```
    Illegal           = 0,
    South             = 1,
    North             = 2,
    DirectionMaxValue = 3,
}

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

/** IP v4 prefix type */
struct IPv4PrefixType {
    1: required IPv4Address    address;
    2: required PrefixLenType  prefixlen;
}

/** IP v6 prefix type */
struct IPv6PrefixType {
    1: required IPv6Address    address;
    2: required PrefixLenType  prefixlen;
}

/** IP address type */
union IPAddressType {
    1: optional IPv4Address   ipv4address;
    2: optional IPv6Address   ipv6address;
}

/** prefix representing reachablity.

    @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 when it moves
 */
```

```
struct PrefixSequenceType {
    1: required IEEE802_1ASTimeStampType  timestamp;
    /** transaction ID set by client in e.g. in 6LoWPAN */
    2: optional PrefixTransactionIDType   transactionid;
}

/** type of TIE.

    This enum indicates what TIE type the TIE is carrying.
    In case the value is not known to the receiver,
    re-flooded the same way as prefix TIEs. This allows for
    future extensions of the protocol within the same schema major
    with types opaque to some nodes unless the flooding scope is not
    the same as prefix TIE, then a major version revision MUST
    be performed.
*/
enum TIETypeType {
    Illegal                                  = 0,
    TIETypeMinValue                          = 1,
    /** first legal value */
    NodeTIEType                              = 2,
    PrefixTIEType                            = 3,
    PositiveDisaggregationPrefixTIEType      = 4,
    NegativeDisaggregationPrefixTIEType      = 5,
    PGPrefixTIEType                          = 6,
    KeyValueTIEType                          = 7,
    ExternalPrefixTIEType                    = 8,
    PositiveExternalDisaggregationPrefixTIEType = 9,
    TIETypeMaxValue                          = 10,
}

/** RIFT route types.

    @note: route types which MUST be ordered on their preference
            PGP prefixes are most preferred attracting
            traffic north (towards spine) and then south
            normal prefixes are attracting traffic south (towards leafs),
            i.e. prefix in NORTH PREFIX TIE is preferred over SOUTH PREFIX TIE.

    @note: The only purpose of those values is to introduce an
           ordering whereas an implementation can choose internally
           any other values as long the ordering is preserved
 */
enum RouteType {
    Illegal              =  0,
    RouteTypeMinValue    =  1,
    /** first legal value. */
    /** discard routes are most prefered */
```

```
    Discard              =  2,

    /** local prefixes are directly attached prefixes on the
     *  system such as e.g. interface routes.
     */
    LocalPrefix          =  3,
    /** advertised in S-TIEs */
    SouthPGPPrefix       =  4,
    /** advertised in N-TIEs */
    NorthPGPPrefix       =  5,
    /** advertised in N-TIEs */
    NorthPrefix          =  6,
    /** externally imported north */
    NorthExternalPrefix  =  7,
    /** advertised in S-TIEs, either normal prefix or positive disaggregation
*/
    SouthPrefix          =  8,
    /** externally imported south */
    SouthExternalPrefix  =  9,
    /** negative, transitive prefixes are least preferred */
    NegativeSouthPrefix  = 10,
    RouteTypeMaxValue    = 11,
}
```

B.2. encoding.thrift

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

include "common.thrift"

/** Represents protocol encoding schema major version */
const common.VersionType protocol_major_version = 2
/** Represents protocol encoding schema minor version */
const common.MinorVersionType protocol_minor_version =  0

/** common RIFT packet header */
struct PacketHeader {
    /** major version type of protocol */
    1: required common.VersionType major_version = protocol_major_version;
    /** minor version type of protocol */
    2: required common.VersionType 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;
}

/** community */
struct Community {
    1: required i32          top;
    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. The schema may add to this
    field future capabilities to indicate whether it will support
    interpretation of future schema extensions on the same major
    revision. Such fields MUST be optional and have an implicit or
    explicit false default value. If a future capability changes route
    selection or generates blackholes if some nodes are not supporting
    it then a major version increment is unavoidable.
*/
struct NodeCapabilities {
    /** must advertise supported minor version dialect that way */
    1: required common.MinorVersionType        protocol_minor_version =
            protocol_minor_version;
    /** can this node participate in flood reduction */
    2: optional bool                           flood_reduction =
            common.flood_reduction_default;
    /** does this node restrict itself to be top-of-fabric or
        leaf only (in ZTP) and does it support leaf-2-leaf procedures */
    3: optional common.HierarchyIndications    hierarchy_indications;
}

/** link capabilities */
struct LinkCapabilities {
    /** indicates that the link is supporting BFD */
    1: optional bool                           bfd =
            common.bfd_default;
    /** indicates whether the interface will support v4 forwarding. This MUST
      * be set to true when LIEs from a v4 address are sent and MAY be set
```

```
      * to true in LIEs on v6 address. If v4 and v6 LIEs indicate contradicting
      * information the behavior is unspecified. */
    2: optional bool                          v4_forwarding_capable =
           true;
}

/** RIFT LIE packet

    @note this node's level is already included on the packet header */
struct LIEPacket {
    /** node or adjacency name */
    1: optional string                        name;
    /** local link ID */
    2: required common.LinkIDType             local_id;
    /** UDP port to which we can receive flooded TIEs */
    3: required common.UDPPortType            flood_port =
           common.default_tie_udp_flood_port;
    /** layer 3 MTU, used to discover to mismatch. */
    4: optional common.MTUSizeType            link_mtu_size =
           common.default_mtu_size;
    /** local link bandwidth on the interface */
    5: optional common.BandwithInMegaBitsType link_bandwidth =
           common.default_bandwidth;
    /** reflects the neighbor once received to provide
        3-way connectivity */
    6: optional Neighbor                      neighbor;
    /** node's PoD */
    7: optional common.PodType                pod =
           common.default_pod;
    /** node capabilities shown in the LIE. The capabilies
        MUST match the capabilities shown in the Node TIEs, otherwise
        the behavior is unspecified. A node detecting the mismatch
        SHOULD generate according error */
   10: required NodeCapabilities              node_capabilities;
   /** capabilities of this link */
   11: optional LinkCapabilities              link_capabilities;
   /** required holdtime of the adjacency, i.e. how much time
       MUST expire without LIE for the adjacency to drop */
   12: required common.TimeIntervalInSecType  holdtime =
           common.default_lie_holdtime;
   /** unsolicited, downstream assigned locally significant label
       value for the adjacency */
   13: optional common.LabelType              label;
    /** indicates that the level on the LIE MUST NOT be used
        to derive a ZTP level by the receiving node */
   21: optional bool                          not_a_ztp_offer =
           common.default_not_a_ztp_offer;
   /** indicates to northbound neighbor that it should
```

```
       be reflooding this node's N-TIEs to achieve flood reduction and
       balancing for northbound flooding. To be ignored if received from a
       northbound adjacency */
   22: optional bool                          you_are_flood_repeater =
            common.default_you_are_flood_repeater;
   /** can be optionally set to indicate to neighbor that packet losses are
seen on
       reception based on packet numbers or the rate is too high. The receiver
SHOULD
       temporarily slow down flooding rates
    */
   23: optional bool                          you_are_sending_too_quickly =
            false;
   /** instance name in case multiple RIFT instances running on same interface
*/
   24: optional string                        instance_name;
}

/** LinkID pair describes one of parallel links between two nodes */
struct LinkIDPair {
    /** node-wide unique value for the local link */
    1: required common.LinkIDType      local_id;
    /** received remote link ID for this link */
    2: required common.LinkIDType      remote_id;

    /** describes the local interface index of the link */
   10: optional common.PlatformInterfaceIndex       platform_interface_index;
   /** describes the local interface name */
   11: optional string                              platform_interface_name;
   /** indication whether the link is secured, i.e. protected by outer key,
absence
       of this element means no indication, undefined outer key means not
secured */
   12: optional common.OuterSecurityKeyID           trusted_outer_security_key;
   /** indication whether the link is protected by established BFD session */
   13: optional bool                                bfd_up;
}

/** ID of a TIE

    @note: TIEID space is a total order achieved by comparing the elements
           in sequence defined and comparing each value as an
           unsigned integer of according length.
*/
struct TIEID {
    /** direction of TIE */
    1: required common.TieDirectionType    direction;
    /** indicates originator of the TIE */
```

```
    2: required common.SystemIDType        originator;
    /** type of the tie */
    3: required common.TIETypeType         tietype;
    /** number of the tie */
    4: required common.TIENrType           tie_nr;
```

```
}

/** Header of a TIE.

   @note: TIEID space is a total order achieved by comparing the elements
              in sequence defined and comparing each value as an
              unsigned integer of according length.

   @note: After sequence number the lifetime received on the envelope
              must be used for comparison before further fields.

   @note: `origination_time` and `origination_lifetime` are disregarded
              for comparison purposes and carried purely for debugging/security
              purposes if present.
*/
struct TIEHeader {
    /** ID of the tie */
    2: required TIEID                           tieid;
    /** sequence number of the tie */
    3: required common.SeqNrType                seq_nr;

    /** absolute timestamp when the TIE
        was generated. This can be used on fabrics with
        synchronized clock to prevent lifetime modification attacks. */
   10: optional common.IEEE802_1ASTimeStampType   origination_time;
   /** original lifetime when the TIE
       was generated. This can be used on fabrics with
       synchronized clock to prevent lifetime modification attacks. */
   12: optional common.LifeTimeInSecType          origination_lifetime;
}

/** Header of a TIE as described in TIRE/TIDE.
*/
struct TIEHeaderWithLifeTime {
    1: required     TIEHeader                           header;
    /** remaining lifetime that expires down to 0 just like in ISIS.
        TIEs with lifetimes differing by less than `lifetime_diff2ignore` MUST
        be considered EQUAL. */
    2: required     common.LifeTimeInSecType          remaining_lifetime;
}

/** TIDE with sorted TIE headers, if headers are unsorted, behavior is
undefined */
struct TIDEPacket {
    /** first TIE header in the tide packet */
    1: required TIEID                       start_range;
    /** last TIE header in the tide packet */
    2: required TIEID                       end_range;
```

```
    /** _sorted_ list of headers */
```

```
    3: required list<TIEHeaderWithLifeTime> headers;
}

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

/** neighbor of a node */
struct NodeNeighborsTIEElement {
    /** level of neighbor */
    1: required common.LevelType                level;
    /**  Cost to neighbor.

         @note: All parallel links to same node
         incur same cost, in case the neighbor has multiple
         parallel links at different cost, the largest distance
         (highest numerical value) MUST be advertised
         @note: any neighbor with cost <= 0 MUST be ignored in computations */
    3: optional common.MetricType               cost = common.default_distance;
    /** can carry description of multiple parallel links in a TIE */
    4: optional set<LinkIDPair>                 link_ids;

    /** total bandwith to neighbor, this will be normally sum of the
        bandwidths of all the parallel links. */
    5: optional common.BandwithInMegaBitsType   bandwidth =
            common.default_bandwidth;
}

/** Flags the node sets */
struct NodeFlags {
    /** indicates that node is in overload, do not transit traffic through it
*/
    1: optional bool         overload = common.overload_default;
}

/** Description of a node.

    It may occur multiple times in different TIEs but if either
        * capabilities values do not match or
        * flags values do not match or
        * neighbors repeat with different values

    the behavior is undefined and a warning SHOULD be generated.
    Neighbors can be distributed across multiple TIEs however if
    the sets are disjoint. Miscablings SHOULD be repeated in every
    node TIE, otherwise the behavior is undefined.

    @note: observe that absence of fields implies defined defaults
```

```
*/
struct NodeTIEElement {
    /** level of the node */
    1: required common.LevelType            level;
    /** node's neighbors. If neighbor systemID repeats in other node TIEs of
        same node the behavior is undefined */
    2: required map<common.SystemIDType,
                NodeNeighborsTIEElement>    neighbors;
    /** capabilities of the node */
    3: required NodeCapabilities            capabilities;
    /** flags of the node */
    4: optional NodeFlags                   flags;
    /** optional node name for easier operations */
    5: optional string                      name;
    /** PoD to which the node belongs */
    6: optional common.PodType              pod;

    /** if any local links are miscabled, the indication is flooded */
   10: optional set<common.LinkIDType>      miscabled_links;

}

struct PrefixAttributes {
    /** distance of the prefix */
    2: required common.MetricType            metric = common.default_distance;
    /** generic unordered set of route tags, can be redistributed to other
protocols or use
        within the context of real time analytics */
    3: optional set<common.RouteTagType>     tags;
    /** monotonic clock for mobile addresses */
    4: optional common.PrefixSequenceType    monotonic_clock;
    /** indicates if the interface is a node loopback */
    6: optional bool                         loopback = false;
    /** indicates that the prefix is directly attached, i.e. should be routed
to even if
        the node is in overload. **/
    7: optional bool                         directly_attached = true;

    /** in case of locally originated prefixes, i.e. interface addresses this
can
        describe which link the address belongs to. */
   10: optional common.LinkIDType            from_link;
}

/** TIE carrying prefixes */
struct PrefixTIEElement {
    /** prefixes with the associated attributes.
        if the same prefix repeats in multiple TIEs of same node
```

```
        behavior is unspecified */
    1: required map<common.IPPrefixType, PrefixAttributes> prefixes;
}
```

```
/** Generic key value pairs */
struct KeyValueTIEElement {
    /** if the same key repeats in multiple TIEs of same node
        or with different values, behavior is unspecified */
    1: required map<common.KeyIDType,string>    keyvalues;
}

/** single element in a TIE. enum `common.TIETypeType`
    in TIEID indicates which elements MUST be present
    in the TIEElement. In case of mismatch the unexpected
    elements MUST be ignored. In case of lack of expected
    element the TIE an error MUST be reported and the TIE
    MUST be ignored.

    This type can be extended with new optional elements
    for new `common.TIETypeType` values without breaking
    the major but if it is necessary to understand whether
    all nodes support the new type a node capability must
    be added as well.
 */
union TIEElement {
    /** used in case of enum common.TIETypeType.NodeTIEType */
    1: optional NodeTIEElement            node;
    /** used in case of enum common.TIETypeType.PrefixTIEType */
    2: optional PrefixTIEElement          prefixes;
    /** positive prefixes (always southbound)
        It MUST NOT be advertised within a North TIE and ignored otherwise
    */
    3: optional PrefixTIEElement          positive_disaggregation_prefixes;
    /** transitive, negative prefixes (always southbound) which
        MUST be aggregated and propagated
        according to the specification
        southwards towards lower levels to heal
        pathological upper level partitioning, otherwise
        blackholes may occur in multiplane fabrics.
        It MUST NOT be advertised within a North TIE.
    */
    5: optional PrefixTIEElement          negative_disaggregation_prefixes;
    /** externally reimported prefixes */
    6: optional PrefixTIEElement          external_prefixes;
    /** positive external disaggregated prefixes (always southbound).
        It MUST NOT be advertised within a North TIE and ignored otherwise
    */
    7: optional PrefixTIEElement
positive_external_disaggregation_prefixes;
    /** Key-Value store elements */
    9: optional KeyValueTIEElement        keyvalues;
}
```

```
/** TIE packet */
struct TIEPacket {
    1: required TIEHeader  header;
    2: required TIEElement element;
}

/** content of a RIFT packet */
union PacketContent {
    1: optional LIEPacket     lie;
    2: optional TIDEPacket    tide;
    3: optional TIREPacket    tire;
    4: optional TIEPacket     tie;
}

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

Appendix C. Finite State Machines and Precise Operational Specifications

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

On Entry action is performed every time and right before the according state is entered, i.e. after any transitions from previous state.

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

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

The FSMs and procedures are NOT normative in the sense that an implementation MUST implement them literally (which would be overspecification) but an implementation MUST exhibit externally observable behavior that is identical to the execution of the specified FSMs.

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

C.1. LIE FSM

Initial state is `OneWay`.

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

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

```
digraph Ga556dde74c30450aae125eaebc33bd57 {
    Nd16ab5092c6b421c88da482eb4ae36b6[label="ThreeWay"][shape="oval"];
    N54edd2b9de7641688608f44fca346303[label="OneWay"][shape="oval"];
    Nfeef2e6859ae4567bd7613a32cc28c0e[label="TwoWay"][shape="oval"];
    N7f2bb2e04270458cb5c9bb56c4b96e23[label="Enter"][style="invis"]
[shape="plain"];
    N292744a4097f492f8605c926b924616b[label="Enter"][style="dashed"]
[shape="plain"];
    Nc48847ba98e348efb45f5b78f4a5c987[label="Exit"][style="invis"]
[shape="plain"];
    Nd16ab5092c6b421c88da482eb4ae36b6 -> N54edd2b9de7641688608f44fca346303
    [label="|NeighborChangedLevel|\n|NeighborChangedAddress|\n|
UnacceptableHeader|\n|MTUMismatch|\n|PODMismatch|\n|HoldtimeExpired|\n|
MultipleNeighbors|"]
    [color="black"][arrowhead="normal" dir="both" arrowtail="none"];
    Nd16ab5092c6b421c88da482eb4ae36b6 -> Nd16ab5092c6b421c88da482eb4ae36b6
    [label="|TimerTick|\n|LieRcvd|\n|SendLie|"][color="black"]
    [arrowhead="normal" dir="both" arrowtail="none"];
    Nfeef2e6859ae4567bd7613a32cc28c0e -> Nfeef2e6859ae4567bd7613a32cc28c0e
    [label="|TimerTick|\n|LieRcvd|\n|SendLie|"][color="black"]
    [arrowhead="normal" dir="both" arrowtail="none"];
    N54edd2b9de7641688608f44fca346303 -> Nd16ab5092c6b421c88da482eb4ae36b6
    [label="|ValidReflection|"][color="red"][arrowhead="normal" dir="both"
arrowtail="none"];
    Nd16ab5092c6b421c88da482eb4ae36b6 -> Nd16ab5092c6b421c88da482eb4ae36b6
    [label="|HALChanged|\n|HATChanged|\n|HALSChanged|\n|UpdateZTPOffer|"]
[color="blue"]
    [arrowhead="normal" dir="both" arrowtail="none"];
    Nd16ab5092c6b421c88da482eb4ae36b6 -> Nd16ab5092c6b421c88da482eb4ae36b6
    [label="|ValidReflection|"][color="red"][arrowhead="normal" dir="both"
arrowtail="none"];
    Nfeef2e6859ae4567bd7613a32cc28c0e -> N54edd2b9de7641688608f44fca346303
    [label="|LevelChanged|"][color="blue"][arrowhead="normal" dir="both"
arrowtail="none"];
    Nfeef2e6859ae4567bd7613a32cc28c0e -> N54edd2b9de7641688608f44fca346303
```

```
    [label="|NeighborChangedLevel|\n|NeighborChangedAddress|\n|
UnacceptableHeader|\n|MTUMismatch|\n|PODMismatch|\n|HoldtimeExpired|\n|
MultipleNeighbors|"]
    [color="black"][arrowhead="normal" dir="both" arrowtail="none"];
    Nfeef2e6859ae4567bd7613a32cc28c0e -> Nd16ab5092c6b421c88da482eb4ae36b6
    [label="|ValidReflection|"][color="red"][arrowhead="normal" dir="both"
arrowtail="none"];
    N54edd2b9de7641688608f44fca346303 -> N54edd2b9de7641688608f44fca346303
    [label="|TimerTick|\n|LieRcvd|\n|NeighborChangedLevel|\n|
NeighborChangedAddress|\n|UnacceptableHeader|\n|MTUMismatch|\n|PODMismatch|\n|
HoldtimeExpired|\n|SendLie|"]
    [color="black"][arrowhead="normal" dir="both" arrowtail="none"];
    N292744a4097f492f8605c926b924616b -> N54edd2b9de7641688608f44fca346303
    [label=""][color="black"][arrowhead="normal" dir="both" arrowtail="none"];
```

```
    Nd16ab5092c6b421c88da482eb4ae36b6 -> N54edd2b9de7641688608f44fca346303
    [label="|LevelChanged|"][color="blue"][arrowhead="normal" dir="both"
arrowtail="none"];
    N54edd2b9de7641688608f44fca346303 -> Nfeef2e6859ae4567bd7613a32cc28c0e
    [label="|NewNeighbor|"][color="black"][arrowhead="normal" dir="both"
arrowtail="none"];
    N54edd2b9de7641688608f44fca346303 -> N54edd2b9de7641688608f44fca346303
    [label="|LevelChanged|\n|HALChanged|\n|HATChanged|\n|HALSChanged|\n|
UpdateZTPOffer|"]
    [color="blue"][arrowhead="normal" dir="both" arrowtail="none"];
    Nfeef2e6859ae4567bd7613a32cc28c0e -> Nfeef2e6859ae4567bd7613a32cc28c0e
    [label="|HALChanged|\n|HATChanged|\n|HALSChanged|\n|UpdateZTPOffer|"]
    [color="blue"][arrowhead="normal" dir="both" arrowtail="none"];
    Nd16ab5092c6b421c88da482eb4ae36b6 -> Nfeef2e6859ae4567bd7613a32cc28c0e
    [label="|NeighborDroppedReflection|"]
    [color="red"][arrowhead="normal" dir="both" arrowtail="none"];
    N54edd2b9de7641688608f44fca346303 -> N54edd2b9de7641688608f44fca346303
    [label="|NeighborDroppedReflection|"][color="red"]
    [arrowhead="normal" dir="both" arrowtail="none"];
}
```

LIE FSM DOT

.. To be updated ..

LIE FSM Figure

Events

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

- NeighborDroppedReflection: lost previous own reflection from neighbor

- NeighborChangedLevel: neighbor changed advertised level
- NeighborChangedAddress: neighbor changed IP address
- UnacceptableHeader: unacceptable header seen
- MTUMismatch: MTU mismatched
- PODMismatch: Unacceptable PoD seen
- HoldtimeExpired: adjacency hold down expired
- MultipleNeighbors: more than one neighbor seen on interface
- SendLie: send a LIE out
- UpdateZTPOffer: update this node's ZTP offer

Actions

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

on HALChanged in TwoWay finishes in TwoWay: store new HAL

on MTUMismatch in ThreeWay finishes in OneWay: no action

on HALChanged in ThreeWay finishes in ThreeWay: store new HAL

on ValidReflection in TwoWay finishes in ThreeWay: no action

on ValidReflection in OneWay finishes in ThreeWay: no action

on NeighborDroppedReflection in ThreeWay finishes in TwoWay: no action

on LieRcvd in ThreeWay finishes in ThreeWay: PROCESS_LIE

on MultipleNeighbors in TwoWay finishes in OneWay: no action

on UnacceptableHeader in ThreeWay finishes in OneWay: no action

on MTUMismatch in TwoWay finishes in OneWay: no action

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

on UnacceptableHeader in TwoWay finishes in OneWay: no action

on HALSChanged in TwoWay finishes in TwoWay: store HALS

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

on NeighborChangedLevel in TwoWay finishes in OneWay: no action

on NewNeighbor in OneWay finishes in TwoWay: PUSH SendLie event

on NeighborChangedAddress in ThreeWay finishes in OneWay: no action

on HALChanged in OneWay finishes in OneWay: store new HAL

on NeighborChangedLevel in OneWay finishes in OneWay: no action

on HoldtimeExpired in TwoWay finishes in OneWay: no action

on SendLie in TwoWay finishes in TwoWay: SEND_LIE

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

on NeighborChangedAddress in OneWay finishes in OneWay: no action

on HATChanged in TwoWay finishes in TwoWay: store HAT

on LieRcvd in TwoWay finishes in TwoWay: PROCESS_LIE

on MultipleNeighbors in ThreeWay finishes in OneWay: no action

on MTUMismatch in OneWay finishes in OneWay: no action

on SendLie in OneWay finishes in OneWay: SEND_LIE

on LieRcvd in OneWay finishes in OneWay: PROCESS_LIE

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

on TimerTick in OneWay finishes in OneWay: PUSH SendLie event

on PODMismatch in ThreeWay finishes in OneWay: no action

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

on NeighborChangedLevel in ThreeWay finishes in OneWay: no action

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

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

on HATChanged in OneWay finishes in OneWay: store HAT

on HATChanged in ThreeWay finishes in ThreeWay: store HAT

on HoldtimeExpired in OneWay finishes in OneWay: no action

on UnacceptableHeader in OneWay finishes in OneWay: no action

on PODMismatch in OneWay finishes in OneWay: no action

on SendLie in ThreeWay finishes in ThreeWay: SEND_LIE

on NeighborChangedAddress in TwoWay finishes in OneWay: no action

on ValidReflection in ThreeWay finishes in ThreeWay: no action

on HALSChanged in OneWay finishes in OneWay: store HALS

on HoldtimeExpired in ThreeWay finishes in OneWay: no action

on HALSChanged in ThreeWay finishes in ThreeWay: store HALS

on NeighborDroppedReflection in OneWay finishes in OneWay: no action

on PODMismatch in TwoWay finishes in OneWay: no action

on Entry into OneWay: CLEANUP

Following words are used for well known procedures:

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

2. CLEANUP: neighbor MUST be reset to unknown

3. SEND_LIE: create a new LIE packet

    1. reflecting the neighbor if known and valid and

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

3. setting `you_are_not_flood_repeater` to computed value

4. PROCESS_LIE:

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

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

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

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

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

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

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

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

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

        5. CHECK_THREE_WAY

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

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

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

C.2. ZTP FSM

Initial state is ComputeBestOffer.

```
digraph Gd436cc3ced8c471eb30bd4f3ac946261 {
    N06108ba9ac894d988b3e4e8ea5ace007
[label="Enter"]
[style="invis"]
[shape="plain"];
    Na47ff5eac9aa4b2eaf12839af68aab1f
[label="MultipleNeighborsWait"]
[shape="oval"];
    N57a829be68e2489d8dc6b84e10597d0b
[label="OneWay"]
[shape="oval"];
    Na641d400819a468d987e31182cdb013e
[label="ThreeWay"]
[shape="oval"];
    Necfbfc2d8e5b482682ee66e604450c7b
[label="Enter"]
[style="dashed"]
[shape="plain"];
    N16db54bf2c5d48f093ad6c18e70081ee
[label="TwoWay"]
[shape="oval"];
    N1b89016876b44cc1b9c1e4a735769560
[label="Exit"]
[style="invis"]
[shape="plain"];
    N16db54bf2c5d48f093ad6c18e70081ee -> N57a829be68e2489d8dc6b84e10597d0b
[label="|NeighborChangedLevel|\n|NeighborChangedAddress|\n|UnacceptableHeader|
\n|MTUMismatch|\n|PODMismatch|\n|HoldtimeExpired|"]
[color="black"]
[arrowhead="normal" dir="both" arrowtail="none"];
    N57a829be68e2489d8dc6b84e10597d0b -> N57a829be68e2489d8dc6b84e10597d0b
[label="|NeighborDroppedReflection|"]
[color="red"]
[arrowhead="normal" dir="both" arrowtail="none"];
    N57a829be68e2489d8dc6b84e10597d0b -> Na47ff5eac9aa4b2eaf12839af68aab1f
[label="|MultipleNeighbors|"]
[color="black"]
[arrowhead="normal" dir="both" arrowtail="none"];
    Necfbfc2d8e5b482682ee66e604450c7b -> N57a829be68e2489d8dc6b84e10597d0b
[label=""]
[color="black"]
[arrowhead="normal" dir="both" arrowtail="none"];
    N57a829be68e2489d8dc6b84e10597d0b -> N16db54bf2c5d48f093ad6c18e70081ee
```

```
[label="|NewNeighbor|"]
```

```
[color="black"]
[arrowhead="normal" dir="both" arrowtail="none"];
    Na641d400819a468d987e31182cdb013e -> Na47ff5eac9aa4b2eaf12839af68aab1f
[label="|MultipleNeighbors|"]
[color="black"]
[arrowhead="normal" dir="both" arrowtail="none"];
    N16db54bf2c5d48f093ad6c18e70081ee -> N16db54bf2c5d48f093ad6c18e70081ee
[label="|HALChanged|\n|HATChanged|\n|HALSChanged|\n|UpdateZTPOffer|"]
[color="blue"]
[arrowhead="normal" dir="both" arrowtail="none"];
    Na641d400819a468d987e31182cdb013e -> N16db54bf2c5d48f093ad6c18e70081ee
[label="|NeighborDroppedReflection|"]
[color="red"]
[arrowhead="normal" dir="both" arrowtail="none"];
    Na47ff5eac9aa4b2eaf12839af68aab1f -> Na47ff5eac9aa4b2eaf12839af68aab1f
[label="|TimerTick|\n|MultipleNeighbors|"]
[color="black"]
[arrowhead="normal" dir="both" arrowtail="none"];
    N57a829be68e2489d8dc6b84e10597d0b -> N57a829be68e2489d8dc6b84e10597d0b
[label="|LevelChanged|\n|HALChanged|\n|HATChanged|\n|HALSChanged|\n|
UpdateZTPOffer|"]
[color="blue"]
[arrowhead="normal" dir="both" arrowtail="none"];
    Na641d400819a468d987e31182cdb013e -> Na641d400819a468d987e31182cdb013e
[label="|HALChanged|\n|HATChanged|\n|HALSChanged|\n|UpdateZTPOffer|"]
[color="blue"]
[arrowhead="normal" dir="both" arrowtail="none"];
    Na641d400819a468d987e31182cdb013e -> N57a829be68e2489d8dc6b84e10597d0b
[label="|NeighborChangedLevel|\n|NeighborChangedAddress|\n|UnacceptableHeader|
\n|MTUMismatch|\n|PODMismatch|\n|HoldtimeExpired|"]
[color="black"]
[arrowhead="normal" dir="both" arrowtail="none"];
    Na47ff5eac9aa4b2eaf12839af68aab1f -> Na47ff5eac9aa4b2eaf12839af68aab1f
[label="|HALChanged|\n|HATChanged|\n|HALSChanged|\n|UpdateZTPOffer|"]
[color="blue"]
[arrowhead="normal" dir="both" arrowtail="none"];
    N16db54bf2c5d48f093ad6c18e70081ee -> N57a829be68e2489d8dc6b84e10597d0b
[label="|LevelChanged|"]
[color="blue"]
[arrowhead="normal" dir="both" arrowtail="none"];
    Na641d400819a468d987e31182cdb013e -> N57a829be68e2489d8dc6b84e10597d0b
[label="|LevelChanged|"]
[color="blue"]
[arrowhead="normal" dir="both" arrowtail="none"];
    N16db54bf2c5d48f093ad6c18e70081ee -> Na47ff5eac9aa4b2eaf12839af68aab1f
[label="|MultipleNeighbors|"]
[color="black"]
[arrowhead="normal" dir="both" arrowtail="none"];
```

```
    Na47ff5eac9aa4b2eaf12839af68aab1f -> N57a829be68e2489d8dc6b84e10597d0b
[label="|MultipleNeighborsDone|"]
```

```
[color="black"]
[arrowhead="normal" dir="both" arrowtail="none"];
    N16db54bf2c5d48f093ad6c18e70081ee -> Na641d400819a468d987e31182cdb013e
[label="|ValidReflection|"]
[color="red"]
[arrowhead="normal" dir="both" arrowtail="none"];
    Na47ff5eac9aa4b2eaf12839af68aab1f -> N57a829be68e2489d8dc6b84e10597d0b
[label="|LevelChanged|"]
[color="blue"]
[arrowhead="normal" dir="both" arrowtail="none"];
    Na641d400819a468d987e31182cdb013e -> Na641d400819a468d987e31182cdb013e
[label="|TimerTick|\n|LieRcvd|\n|SendLie|"]
[color="black"]
[arrowhead="normal" dir="both" arrowtail="none"];
    N57a829be68e2489d8dc6b84e10597d0b -> N57a829be68e2489d8dc6b84e10597d0b
[label="|TimerTick|\n|LieRcvd|\n|NeighborChangedLevel|\n|
NeighborChangedAddress|\n|NeighborAddressAdded|\n|UnacceptableHeader|\n|
MTUMismatch|\n|PODMismatch|\n|HoldtimeExpired|\n|SendLie|"]
[color="black"]
[arrowhead="normal" dir="both" arrowtail="none"];
    N57a829be68e2489d8dc6b84e10597d0b -> Na641d400819a468d987e31182cdb013e
[label="|ValidReflection|"]
[color="red"]
[arrowhead="normal" dir="both" arrowtail="none"];
    N16db54bf2c5d48f093ad6c18e70081ee -> N16db54bf2c5d48f093ad6c18e70081ee
[label="|TimerTick|\n|LieRcvd|\n|SendLie|"]
[color="black"]
[arrowhead="normal" dir="both" arrowtail="none"];
    Na641d400819a468d987e31182cdb013e -> Na641d400819a468d987e31182cdb013e
[label="|ValidReflection|"]
[color="red"]
[arrowhead="normal" dir="both" arrowtail="none"];
}
```

ZTP FSM DOT

Events

- TimerTick: one second timer tic
- LevelChanged: node's level has been changed by ZTP or configuration
- HALChanged: best HAL computed by ZTP has changed
- HATChanged: HAT computed by ZTP has changed

- HALSChanged: set of HAL offering systems computed by ZTP has changed
- LieRcvd: received LIE
- NewNeighbor: new neighbor parsed
- ValidReflection: received own reflection from neighbor
- NeighborDroppedReflection: lost previous own reflection from neighbor
- NeighborChangedLevel: neighbor changed advertised level
- NeighborChangedAddress: neighbor changed IP address
- UnacceptableHeader: unacceptable header seen
- MTUMismatch: MTU mismatched
- PODMismatch: Unacceptable PoD seen
- HoldtimeExpired: adjacency hold down expired
- MultipleNeighbors: more than one neighbor seen on interface
- MultipleNeighborsDone: cooldown for multiple neighbors expired
- SendLie: send a LIE out
- UpdateZTPOffer: update this node's ZTP offer

Actions

on MTUMismatch in OneWay finishes in OneWay: no action

on HoldtimeExpired in OneWay finishes in OneWay: no action

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

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

on HALChanged in MultipleNeighborsWait finishes in MultipleNeighborsWait: store new HAL

on NeighborChangedAddress in ThreeWay finishes in OneWay: no action

on ValidReflection in OneWay finishes in ThreeWay: no action

on MTUMismatch in TwoWay finishes in OneWay: no action

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

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

on HATChanged in ThreeWay finishes in ThreeWay: store HAT

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

on HALSChanged in TwoWay finishes in TwoWay: store HALS

on PODMismatch in TwoWay finishes in OneWay: no action

on LieRcvd in TwoWay finishes in TwoWay: PROCESS_LIE

on PODMismatch in ThreeWay finishes in OneWay: no action

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

on SendLie in TwoWay finishes in TwoWay: SEND_LIE

on SendLie in OneWay finishes in OneWay: SEND_LIE

on TimerTick in OneWay finishes in OneWay: PUSH SendLie event

on HALChanged in OneWay finishes in OneWay: store new HAL

on HALSChanged in ThreeWay finishes in ThreeWay: store HALS

on NeighborChangedLevel in TwoWay finishes in OneWay: no action

on PODMismatch in OneWay finishes in OneWay: no action

on HoldtimeExpired in TwoWay finishes in OneWay: no action

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

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

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

on LieRcvd in OneWay finishes in OneWay: PROCESS_LIE

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

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

on HALChanged in TwoWay finishes in TwoWay: store new HAL

on UnacceptableHeader in OneWay finishes in OneWay: no action

on HALSChanged in OneWay finishes in OneWay: store HALS

on HALSChanged in MultipleNeighborsWait finishes in MultipleNeighborsWait: store HALS

on SendLie in ThreeWay finishes in ThreeWay: SEND_LIE

on MTUMismatch in ThreeWay finishes in OneWay: no action

on HATChanged in MultipleNeighborsWait finishes in MultipleNeighborsWait: store HAT

on NeighborChangedAddress in OneWay finishes in OneWay: no action

on ValidReflection in TwoWay finishes in ThreeWay: no action

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

on NeighborChangedLevel in OneWay finishes in OneWay: no action

on HATChanged in OneWay finishes in OneWay: store HAT

on NeighborDroppedReflection in OneWay finishes in OneWay: no action

on HALChanged in ThreeWay finishes in ThreeWay: store new HAL

on NeighborAddressAdded in OneWay finishes in OneWay: no action

on NeighborChangedAddress in TwoWay finishes in OneWay: no action

on LieRcvd in ThreeWay finishes in ThreeWay: PROCESS_LIE

on UnacceptableHeader in TwoWay finishes in OneWay: no action

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

on HATChanged in TwoWay finishes in TwoWay: store HAT

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

on ValidReflection in ThreeWay finishes in ThreeWay: no action

on UnacceptableHeader in ThreeWay finishes in OneWay: no action

on HoldtimeExpired in ThreeWay finishes in OneWay: no action

on NeighborChangedLevel in ThreeWay finishes in OneWay: no action

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

on NewNeighbor in OneWay finishes in TwoWay: PUSH SendLie event

on NeighborDroppedReflection in ThreeWay finishes in TwoWay: no action

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

on Entry into OneWay: CLEANUP

Following words are used for well known procedures:

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

2. CLEANUP: neighbor MUST be reset to unknown

3. SEND_LIE: create a new LIE packet

    1. reflecting the neighbor if known and valid and

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

3. setting `you_are_not_flood_repeater` to computed value

4. PROCESS_LIE:

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

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

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

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

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

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

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

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

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

        5. CHECK_THREE_WAY

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

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

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

C.3. Flooding Procedures

Flooding Procedures are described in terms of a flooding state of an adjacency and resulting operations on it driven by packet arrivals. The FSM has basically a single state and is not well suited to represent the behavior.

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

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

C.3.1. FloodState Structure per Adjacency

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

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

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

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

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

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

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

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

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

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

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

try_to_transmit_tie(TIE):

A. if not is_flood_filtered(TIE) then

1. remove TIE from TIES_RTX if present

2. if TIE" with same key on TIES_ACK then

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

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

3. else insert TIE into TIES_TX

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

tie_been_acked(TIE): remove TIE from all collections.

remove_from_all_queues(TIE): same as `tie_been_acked`.

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

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

clear_requests(TIEs): remove all TIEs from TIES_REQ.

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

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

Elements on TIES_ACK should be processed with highest priority

TIES_TX

TIES_REQ and TIES_RTX

C.3.2. TIDEs

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

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

C.3.2.1. TIDE Generation

As given by timer constant, periodically generate TIDEs by:

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

TIDE_START: Begin of TIDE packet range.

a. NEXT_TIDE_ID = MIN_TIEID

b. while NEXT_TIDE_ID not equal to MAX_TIEID do

1. TIDE_START = NEXT_TIDE_ID

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

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

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

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

6. NEXT_TIDE_ID = END

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

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

C.3.2.2. TIDE Processing

On reception of TIDEs the following processing is performed:

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

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

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

LASTPROCESSED: Last processed TIEID in TIDE

DBTIE: TIE in the LSDB if found

a. LASTPROCESSED = TIDE.start_range

b. for every HEADER in TIDE do

1. DBTIE = find HEADER in current LSDB

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

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

4. LASTPROCESSED = HEADER

5. if DBTIE not found then

I) if originator is this node then bump_own_tie

II) else put HEADER into REQKEYS

6. if DBTIE.HEADER < HEADER then

I) if originator is this node then bump_own_tie else

i. if this is a N-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)

C.3.3. TIREs

C.3.3.1. TIRE Generation

There is not much to say here. Elements from both TIES_REQ and TIES_ACK MUST be collected and sent out as fast as feasible as TIREs. When sending TIREs with elements from TIES_REQ the `lifetime` field MUST be set to 0 to force reflooding from the neighbor even if the TIEs seem to be same.

C.3.3.2. 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)

C.3.4. 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)

C.3.5. TIEs Processing When LSDB Received Newer Version on Other Adjacencies

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

C.3.6. Sending TIEs

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

Appendix D. Constants

D.1. Configurable Protocol Constants

This section gathers constants that are provided in the schema files and in the document.

	Type	Value
LIE IPv4 Multicast Address	Default Value, Configurable	224.0.0.120 or all-rift-routers to be assigned in IPv4 Multicast Address Space Registry in Local Network Control Block
LIE IPv6 Multicast Address	Default Value, Configurable	FF02::A1F7 or all-rift-routers to be assigned in IPv6 Multicast Address Assignments
LIE Destination Port	Default Value, Configurable	914
Level value for TOP_OF_FABRIC flag	Constant	24
Default LIE Holdtime	Default Value, Configurable	3 seconds
TIE Retransmission Interval	Default Value	1 second
TIDE Generation Interval	Default Value, Configurable	5 seconds
MIN_TIEID signifies start of TIDEs	Constant	TIE Key with minimal values: TIEID(originator=0, tietype=TIETypeMinValue, tie_nr=0, direction=South)
MAX_TIEID signifies end of TIDEs	Constant	TIE Key with maximal values: TIEID(originator=MAX_UINT64, tietype=TIETypeMaxValue, tie_nr=MAX_UINT64, direction=North)

Table 6: all_constants

Authors' Addresses

Tony Przygienda (editor)
Juniper
1137 Innovation Way
Sunnyvale, CA
USA

Email: prz@juniper.net

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

Email: Alankar_Sharma@comcast.com

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

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

Bruno Rijsman
Individual

Email: fl0w@yandex-team.ru

Dmitry Afanasiev
Yandex

Email: fl0w@yandex-team.ru