Networking Working Group
Internet-Draft
Intended status: Standards Track
Expires: May 1, 2018

T. Przygienda
Juniper Networks
A. Sharma
Comcast
J. Drake
A. Atlas
Juniper Networks
October 28, 2017

RIFT: Routing in Fat Trees
draft-przygienda-rift-03

Abstract

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

Table of Contents

1. Introduction 3
1.1. Requirements Language 4
2. Reference Frame 4
2.1. Terminology 4
2.2. Topology 6
3. Requirement Considerations 8
4. RIFT: Routing in Fat Trees 10
4.1. Overview 10
4.2. Specification 10
4.2.1. Transport 10
4.2.2. Link (Neighbor) Discovery (LIE Exchange) 11
4.2.3. Topology Exchange (TIE Exchange) 11
4.2.3.1. Topology Information Elements 12
4.2.3.2. South- and Northbound Representation 12
4.2.3.3. Flooding 14
4.2.3.4. TIE Flooding Scopes 15
4.2.3.5. Initial and Periodic Database Synchronization . . 17
4.2.3.6. Purging 17
4.2.3.7. Southbound Default Route Origination 18
4.2.3.8. Optional Automatic Flooding Reduction and Partitioning 18
4.2.4. Policy-Guided Prefixes 19
4.2.4.1. Ingress Filtering 20
4.2.4.2. Applying Policy 21
4.2.4.3. Store Policy-Guided Prefix for Route Computation and Regeneration 22
4.2.4.4. Re-origination 22
4.2.4.5. Overlap with Disaggregated Prefixes 23
4.2.5. Reachability Computation 23
4.2.5.1. Northbound SPF 23
4.2.5.2. Southbound SPF 24

4.2.5.3. East-West Forwarding Within a Level 24
4.2.6. Attaching Prefixes 24
4.2.7. Attaching Policy-Guided Prefixes 26
4.2.8. Automatic Disaggregation on Link & Node Failures . . 27
4.2.9. Optional Autoconfiguration 30
4.3. Further Mechanisms 30
4.3.1. Overload Bit 30
4.3.2. Optimized Route Computation on Leafs 31
4.3.3. Key/Value Store 31
4.3.4. Interactions with BFD 31
4.3.5. Leaf to Leaf Procedures 32
4.3.6. Other End-to-End Services 32
4.3.7. Address Family and Multi Topology Considerations . . 33
4.3.8. Reachability of Internal Nodes in the Fabric 33
4.3.9. One-Hop Healing of Levels with East-West Links . . . 33
5. Examples 33
5.1. Normal Operation 33
5.2. Leaf Link Failure 35
5.3. Partitioned Fabric 36
5.4. Northbound Partitioned Router and Optional East-West Links 37
6. Implementation and Operation: Further Details 38
6.1. Considerations for Leaf-Only Implementation 39
6.2. Adaptations to Other Proposed Data Center Topologies . . 39
6.3. Originating Non-Default Route Southbound 40
7. Security Considerations 40
8. Information Elements Schema 40
8.1. common.thrift 41
8.2. encoding.thrift 44
9. IANA Considerations 49
10. Security Considerations 49
11. Acknowledgments 49
12. References 49
12.1. Normative References 49
12.2. Informative References 51
Authors' Addresses 52

1. Introduction

Clos [CLOS] and Fat-Tree [FATTREE] have gained prominence in today's networking, primarily as a result of a 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. The existing set of dynamic routing protocols was geared originally towards a network with an irregular topology and low degree of connectivity and consequently several attempts to adapt those have been made. Most successfully BGP [RFC4271] [RFC7938] has been extended to this purpose, not as much due to its inherent suitability

to solve the problem but rather because the perceived capability to modify it "quicker" and the immanent difficulties with link-state [DIJKSTRA] based protocols to fulfill certain of the resulting requirements.

In looking at the problem through the very lens of its requirements an optimal approach does not seem 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’. The balance of this document details the resulting protocol.

1.1. Requirements Language

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

2. Reference Frame

2.1. Terminology

This section presents the terminology used in this document. It is assumed that the reader is thoroughly familiar with the terms and concepts used in OSPF [RFC2328] and IS-IS [RFC1142], [ISO10589] as well as the according graph theoretical concepts of shortest path first (SPF) [DIJKSTRA] computation and directed acyclic graphs (DAG).

Level: Clos and Fat Tree networks are trees and ’level’ denotes the set of nodes at the same height in such a network, where the bottom level is level 0. A node has links to nodes one level down and/or one level up. Under some circumstances, a node may have links to nodes at the same level. As footnote: Clos terminology uses often the concept of "stage" but due to the folded nature of the Fat Tree we do not use it to prevent misunderstandings.

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

Point of Delivery (PoD): A self-contained vertical slice of a Clos or Fat Tree network containing normally only level 0 and level 1 nodes. It communicates with nodes in other PoDs via the spine.

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

Leaf: A node at level 0.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.2. Topology

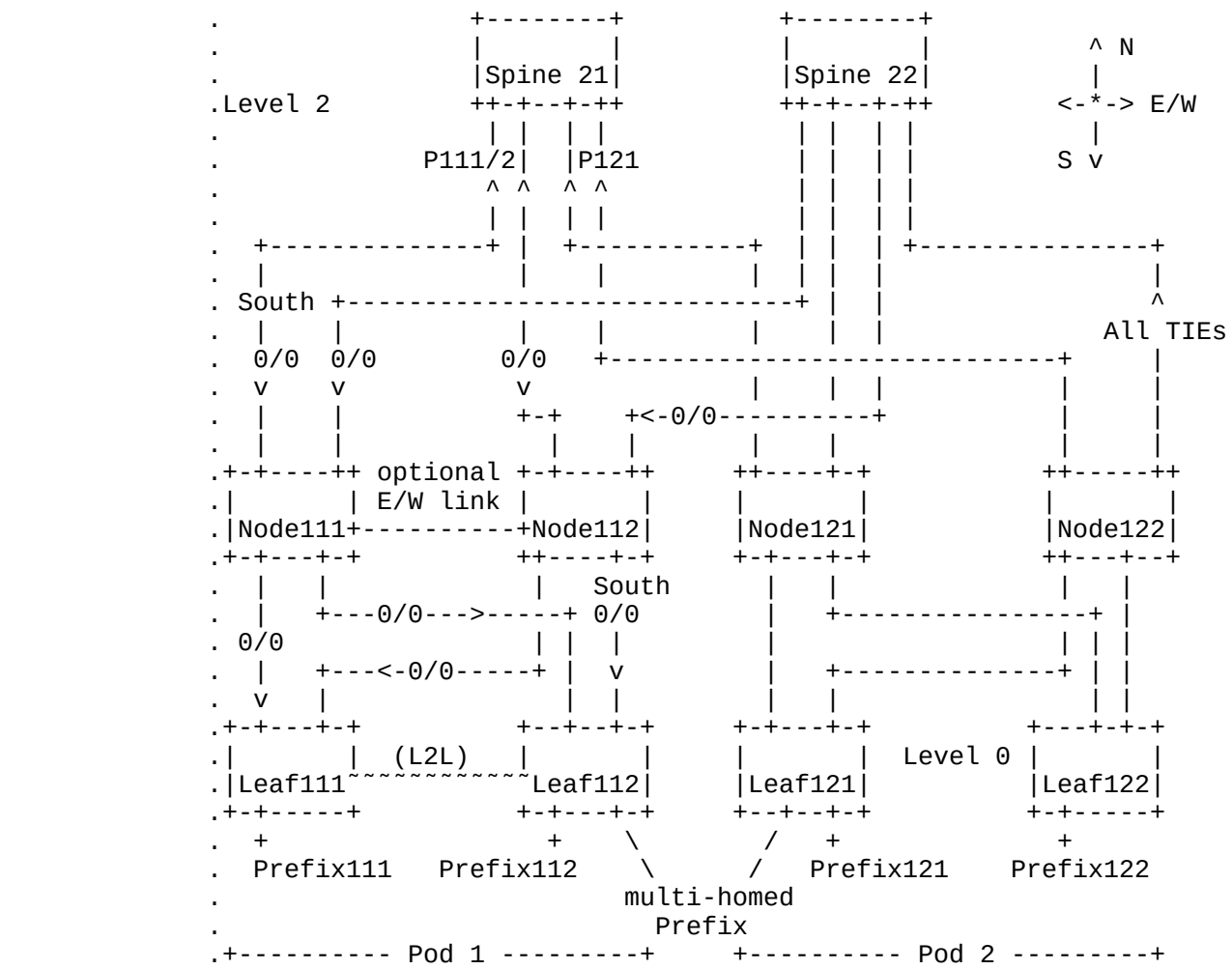

Figure 1: A two level spine-and-leaf topology

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

3. Requirement Considerations

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

REQ1: The control protocol should discover the physical links automatically and be able to detect cabling that violates fat-tree topology constraints. It must react accordingly to such mis-cabling attempts, at a minimum preventing adjacencies between nodes from being formed and traffic from being forwarded on those mis-cabled links. E.g. connecting a leaf to a spine at level 2 should be detected and ideally prevented.

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

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

REQ4: The solution should allow for minimum size routing information base and forwarding tables at leaf level for speed, cost and simplicity reasons. Holding excessive amount of information away from leaf nodes simplifies operation and lowers cost of the underlay.

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

REQ6: Traffic engineering should be allowed by modification of prefixes and/or their next-hops.

REQ7: 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].

REQ8: 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.

REQ9: 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.

REQ10: 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.

REQ11: Reducing the scope of communication needed throughout the network on link and state failure, as well as reducing advertisements of repeating, idiomatic or policy-guided information in stable state is highly desirable since it leads to better stability and faster convergence behavior.

REQ12: Once a packet traverses a link in a "southbound" direction, it must not take any further "northbound" steps along its path to delivery to its destination under normal conditions. Taking a path through the spine in cases where a shorter path is available is highly undesirable.

REQ13: Parallel links between same set of nodes must be distinguishable for SPF, failure and traffic engineering purposes.

REQ14: The protocol must not rely on interfaces having discernible unique addresses, i.e. it must operate in presence of unnumbered links (even parallel ones) or links of a single node having same addresses.

Following list represents possible requirements and requirements under discussion:

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

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

Finally, following are the non-requirements:

NONREQ1: Broadcast media support is unnecessary.

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

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

4. RIFT: Routing in Fat Trees

Derived from the above requirements we present a detailed outline of a protocol optimized for Routing in Fat Trees (RIFT) that in most abstract terms has many properties of a modified link-state protocol [RFC2328][RFC1142] when "pointing north" and path-vector [RFC4271] protocol when "pointing south". Albeit an unusual combination, it does quite naturally exhibit the desirable properties we seek.

4.1. Overview

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

4.2. Specification

4.2.1. Transport

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

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

4.2.2. Link (Neighbor) Discovery (LIE Exchange)

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

Each node is provisioned with the level at which it is operating and its PoD. A default level and PoD of zero are assumed, meaning that leafs do not need to be configured with a level (or even PoD). Nodes in the spine are configured with a PoD of zero. This information is propagated in the LIEs exchanged. Adjacencies are formed if and only if

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

2. the neighboring node is at most one level away AND

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

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

5. the advertised MTUs match on both sides.

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

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

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

4.2.3. Topology Exchange (TIE Exchange)

4.2.3.1. Topology Information Elements

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

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

TIEs contain sequence numbers, lifetimes and a type. Each type has a large identifying number space and information is spread across possibly many TIEs of a certain type by the means of a hash function that a node or deployment can individually determine. One extreme side of the scale is a prefix per TIE which leads to BGP-like behavior vs. dense packing into few TIEs leading to more traditional IGP trade-off with fewer TIEs. An implementation may even rehash at the cost of significant amount of re-advertisements of TIEs.

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

4.2.3.2. South- and Northbound Representation

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

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

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

TIE-Type	Content
node N-TIE	node properties 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 de-aggregated prefixes
PGP N-TIE	contains nodes north PGPs
PGP S-TIE	contains nodes south PGPs
KV N-TIE	contains nodes northbound KVs
KV S-TIE	contains nodes southbound KVs

Table 1: TIE Types

As an example illustrating a databases holding both representations, consider the topology in Figure 1 with the optional link between node 111 and node 112 (so that the flooding on an east-west link can be shown). This example assumes unnumbered interfaces. First, here are the TIEs generated by some nodes. For simplicity, the key value elements and the PGP elements which may be included in their S-TIEs or N-TIEs are not shown.

```
Spine21 S-TIEs:
Node S-TIE:
  NodeElement(layer=2, neighbors((Node111, layer 1, cost 1),
  (Node112, layer 1, cost 1), (Node121, layer 1, cost 1),
  (Node122, layer 1, cost 1)))
Prefix S-TIE:
  SouthPrefixesElement(prefixes(0/0, cost 1), (::/0, cost 1))

Node111 S-TIEs:
Node S-TIE:
  NodeElement(layer=1, neighbors((Spine21,layer 2,cost 1),
  (Spine22, layer 2, cost 1), (Node112, layer 1, cost 1),
  (Leaf111, layer 0, cost 1), (Leaf112, layer 0, cost 1)))
Prefix S-TIE:
  SouthPrefixesElement(prefixes(0/0, cost 1), (::/0, cost 1))

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

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

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

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

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

4.2.3.3. Flooding

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

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

4.2.3.4. TIE Flooding Scopes

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

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

A node’s node S-TIEs, consisting of all node’s adjacencies and prefix S-TIEs with default IP prefix and disaggregated prefixes, are flooded southbound in order to allow the nodes one level down to see connectivity of the higher level as well as reachability to the rest of the fabric. In order to allow a E-W disconnected node in a given level to receive the S-TIEs of other nodes at its level, every *NODE* S-TIE is "reflected" northbound to level from which it was received. It should be noted that east-west links are included in South TIE flooding; those TIEs need to be flooded to satisfy algorithms in Section 4.2.5. In that way nodes at same level can learn about each other without a more southern level, e.g. in case of leaf level. The precise flooding scopes are given in Table 2. Those rules govern in a symmetric fashion what SHOULD be included in TIDEs towards neighbors.

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

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

Table 2: Flooding Scopes

As an example to illustrate these rules, consider using the topology in Figure 1, with the optional link between node 111 and node 112, and the associated TIEs given in Figure 2. The flooding from particular nodes of the TIEs is given in Table 3.

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

Table 3: Flooding some TIEs from example topology

4.2.3.5. Initial and Periodic Database Synchronization

The initial exchange of RIFT is modeled after ISIS with TIDE being equivalent to CSNP and TIRE playing the role of PSNP. The content of TIDEs and TIREs is governed by Table 2.

4.2.3.6. Purging

RIFT does not purge information that has been distributed by the protocol. Purging mechanisms in other routing protocols have proven through many years of experience to be complex and fragile. 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 will timeout and clean up the according empty TIEs independently.

4.2.3.7. Southbound Default Route Origination

Under certain conditions nodes issue a default route in their South Prefix TIEs.

A node X that

1. is NOT overloaded AND

2. has southbound or east-west adjacencies

originates in its south prefix TIE such a default route IIF

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

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

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

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

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

4.2.3.8. Optional Automatic Flooding Reduction and Partitioning

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

Steps for flooding reduction and partitioning:

1. select all nodes in the same level for which node S-TIEs have been received and which have precisely the same non-empty sets of respectively north and south neighbor adjacencies and support flooding reduction (overload bits are ignored) and then

2. run on the chosen set a hash algorithm using nodes flood priorities and IDs to select flooding leader and backup per TIE originator ID, i.e. each node floods immediately through to all its necessary neighbors TIEs that it received with an originator ID that makes it the flooding leader or backup for this originator. The preference (higher is better) is computed as XOR(TIE-ORIGINATOR-ID<<1,~OWN-SYSTEM-ID)), whereas << is a non-circular shift and ~ is bit-wise NOT.

3. In the very unlikely case of hash collisions on either of the two nodes with highest values (i.e. either does NOT produce unique hashes as compared to all other hash values), the node running the election does not attempt to reduce flooding.

Additional rules for flooding reduction and partitioning:

1. A node always floods its own TIEs

2. A node generates TIDEs as usual but when receiving TIREs with requests for TIEs for a node for which it is not a flooding leader or backup it ignores such TIDEs on first request only. Normally, the flooding leader should satisfy the requestor and with that no further TIREs for such TIEs will be generated. Otherwise, the next set of TIDEs and TIREs will lead to flooding independent of the flooding leader status.

3. A node receiving a TIE originated by a node for which it is not a flooding leader floods such TIEs only when receiving an out-of-date TIDE for them, except for the first one.

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

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

4.2.4. Policy-Guided Prefixes

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

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

the south direction. If an PGP S-TIE is received on an east-west or northbound link, it must be discarded by ingress filtering.

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

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

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

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

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

4.2.4.1. Ingress Filtering

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

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

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

cost: This is initialized to the cost received

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

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

4.2.4.2. Applying Policy

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

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

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

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

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

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

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

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

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

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

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

There are three different policies that can be used:

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

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

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

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

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

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

4.2.4.4. Re-origination

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

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

4.2.4.5. Overlap with Disaggregated Prefixes

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

4.2.5. Reachability Computation

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

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

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

4.2.5.1. Northbound SPF

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

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

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

2. the adjacent node has one or more northbound adjacencies

This rule forms a "one-hop default route split-horizon" and prevents looping over default routes while allowing for "one-hop protection" of nodes that lost all northbound adjacencies.

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

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

2. the node does not originate a supersuming prefix itself.

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

A detailed example can be found in Section 5.4.

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

4.2.5.2. Southbound SPF

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

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

4.2.5.3. East-West Forwarding Within a Level

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

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

4.2.6. Attaching Prefixes

After the SPF is run, it is necessary to attach according prefixes. For S-SPF, prefixes from an N-TIE are attached to the originating node with that node’s next-hop set and a distance equal to the

prefix's cost plus the node's minimized path distance. The RIFT route database, a set of (prefix, type=spf, path_distance, next-hop set), accumulates these results. Obviously, the prefix retains its type which is used to tie-break between the same prefix advertised with different types.

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

Each prefix has its cost adjusted before being added into the RIFT route database. The cost of the prefix is set to the cost received plus the cost of the minimum cost next-hop to that neighbor. Then each prefix can be added into the RIFT route database with the next_hop_set; ties are broken based upon type first and then distance. RIFT route preferences are normalized by the according thrift model type.

An exemplary implementation for node X follows:

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

Figure 3: Adding Routes from S-TIE Prefixes

4.2.7. Attaching Policy-Guided Prefixes

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

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

Figure 4: Adding Routes from Policy-Guided Prefixes

4.2.8. Automatic Disaggregation on Link & Node Failures

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

We refer to the process of advertising additional prefixes as 'de-aggregation' or 'dis-aggregation'.

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

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

be |H(r). Observe that policy-guided prefixes are NOT affected since their scope is controlled by configuration.

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

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

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

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

A possible algorithm is described last:

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

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

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

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

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

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

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

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

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

Figure 5: Computation to Disaggregate Prefixes

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

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

1. all neighbor relationships MUST perform backlink checks.

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

3. all the lower level nodes are flooded the same disaggregated prefixes since we don't want to build an S-TIE per node and complicate things unnecessarily. The PoD containing the prefix will prefer southbound anyway.

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

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

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

4.2.9. Optional Autoconfiguration

RIFT nodes can operate in an optional mode where the levels in the fabric are being elected fully automatically. Future version of this document will render more detailed specification of the necessary protocol extensions.

4.3. Further Mechanisms

4.3.1. Overload Bit

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

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

4.3.2. Optimized Route Computation on Leafs

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

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

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

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

4.3.3. Key/Value Store

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 TIEs (which are always S-TIEs) can arrive from multiple nodes and need tie-breaking per key uses the following rules

1. Only KV TIEs originated by a node to which the receiver has an adjacency are considered.

2. Within all valid KV S-TIEs containing the key, the value of the S-TIE with the highest level and within the same level highest originator ID is preferred.

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 is not result of independent computation of every node but a diffused computation.

4.3.4. Interactions with BFD

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

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

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

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

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

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

4.3.5. Leaf to Leaf Procedures

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

Only nodes supporting Leaf to Leaf Procedures CAN advertise LIEs on E-W links at level 0 and MUST in such a case advertise the according flag in node capabilities as "true".

The overload bit MUST be set on all leaf’s node TIEs.

Only node’s own north and south TIEs are flooded over E-W leaf adjacency.

E-W leaf adjacency is always used in both north as well as south computation.

Any advertised aggregate in leaf’s south TIE MUST install a discard route.

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

4.3.6. Other End-to-End Services

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

4.3.7. Address Family and Multi Topology Considerations

Multi-Topology (MT)[RFC5120] and Multi-Instance (MI)[RFC6822] is used today in link-state routing protocols to support several domains on the same physical topology. RIFT supports this capability by carrying transport ports in the LIE protocol exchanges. Multiplexing of LIEs can be achieved by either choosing varying multicast addresses or ports on the same address.

BFD interactions in Section 4.3.4 are implementation dependent when multiple RIFT instances run on the same link.

4.3.8. Reachability of Internal Nodes in the Fabric

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

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

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

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

5. Examples

5.1. Normal Operation

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

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

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

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

3. Node 111 to Leaf 111, Leaf 112

4. Node 112 to Leaf 111, Leaf 112

5. Node 121 to Leaf 121, Leaf 122

6. Node 122 to Leaf 121, Leaf 122

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

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

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

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

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

Similarly, an S Tie with a default IP prefix would be originated by Node 121 and Node 122 and each would be sent to Leaf 121 and Leaf 122. Leaf 121 and Leaf 122 would each send the S-TIE from Node 121

to Node 122 and the S-TIE from Node 122 to Node 121. At this point IP connectivity with maximum possible ECMP has been established between the leafs while constraining the amount of information held by each node to the minimum necessary for normal operation and dealing with failures.

5.2. Leaf Link Failure

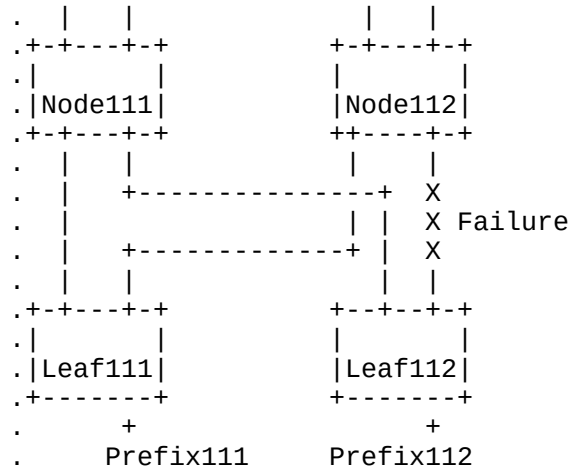

Figure 6: Single Leaf link failure

In case of a failing leaf link between node 112 and leaf 112 the link-state information will cause re-computation of the necessary SPF and the higher levels will stop forwarding towards prefix 112 through node 112. Only nodes 111 and 112, as well as both spines will see control traffic. Leaf 111 will receive a new S-TIE from node 112 and reflect back to node 111. Node 111 will de-aggregate prefix 111 and prefix 112 but we will not describe it further here since de-aggregation is emphasized in the next example. It is worth observing however in this example that if leaf 111 would keep on forwarding traffic towards prefix 112 using the advertised south-bound default of node 112 the traffic would end up on spine 21 and spine 22 and cross back into pod 1 using node 111. This is arguably not as bad as black-holing present in the next example but clearly undesirable. Fortunately, de-aggregation prevents this type of behavior except for a transitory period of time.

5.3. Partitioned Fabric

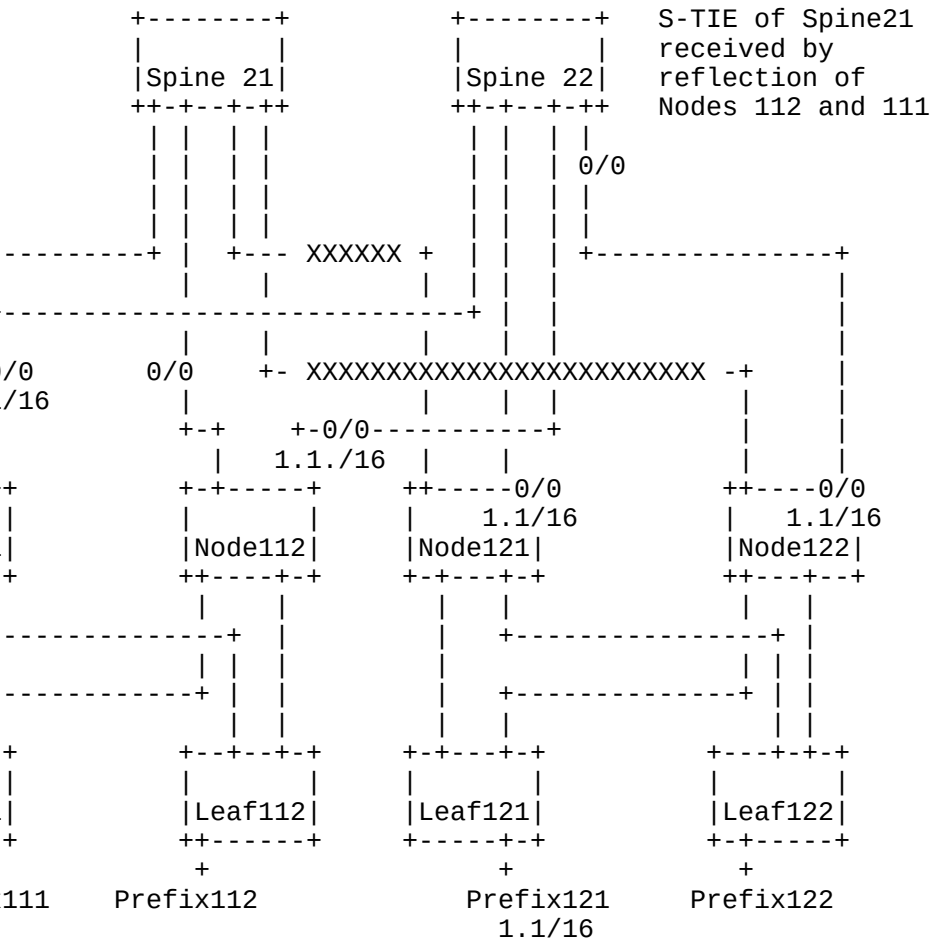

Figure 7: Fabric partition

Figure 7 shows the arguably most catastrophic but also the most interesting case. Spine 21 is completely severed from access to Prefix 121 (we use in the figure 1.1/16 as example) by double link failure. However unlikely, if left unresolved, forwarding from leaf 111 and leaf 112 to prefix 121 would suffer 50% black-holing based on pure default route advertisements by spine 21 and spine 22.

The mechanism used to resolve this scenario is hinging on the distribution of southbound representation by spine 21 that is reflected by node 111 and node 112 to spine 22. Spine 22, having

computed reachability to all prefixes in the network, advertises with the default route the ones that are reachable only via lower level neighbors that spine 21 does not show an adjacency to. That results in node 111 and node 112 obtaining a longest-prefix match to prefix 121 which leads through spine 22 and prevents black-holing through spine 21 still advertising the 0/0 aggregate only.

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

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

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

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

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

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

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

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

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

5.4. Northbound Partitioned Router and Optional East-West Links

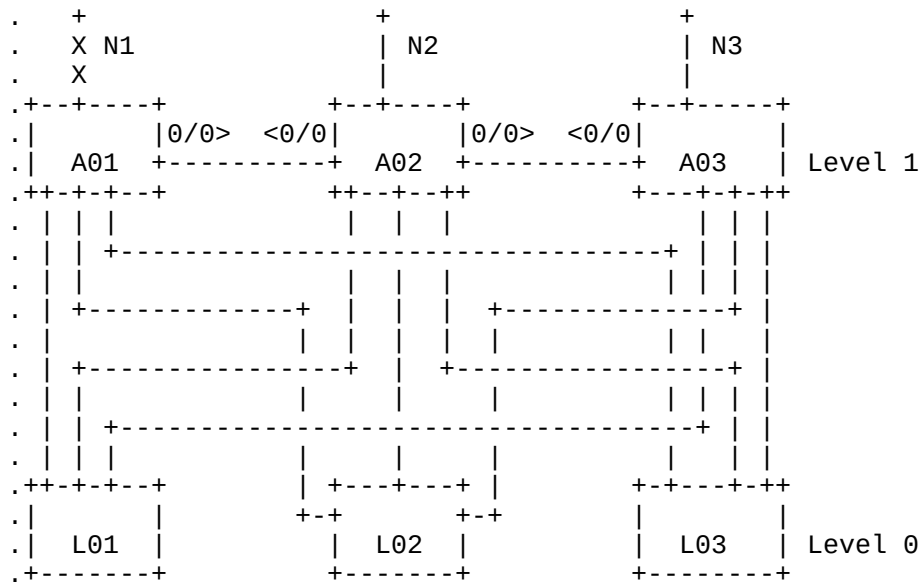

Figure 8: North Partitioned Router

Figure 8 shows a part of a fabric where level 1 is horizontally connected and A01 lost its only northbound adjacency. Based on N-SPF rules in Section 4.2.5.1 A01 will compute northbound reachability by using the link A01 to A02 (whereas A02 will NOT use this link during N-SPF). Hence A01 will still advertise the default towards level 0 and route unidirectionally using the horizontal link. Moreover, based on Section 4.3.8 it may advertise its loopback address as south PGP to remain reachable "from the south" for operational purposes. This is necessary since A02 will NOT route towards A01 using the E-W link (doing otherwise may form routing loops).

As further consideration, the moment A02 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 4.2.3.7. Moreover, A02 may now inject its loopback address as south PGP.

6. Implementation and Operation: Further Details

6.1. Considerations for Leaf-Only Implementation

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

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

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

3. Leaf node does not have to support flooding reduction and de-aggregation.

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

6.2. Adaptations to Other Proposed Data Center Topologies

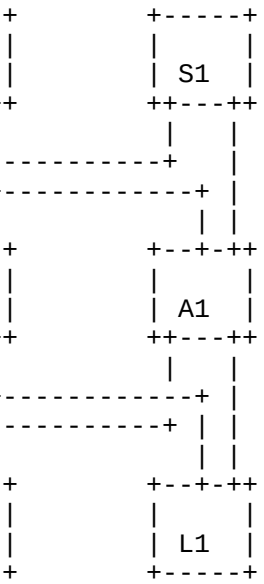

Figure 9: Level Shortcut

Strictly speaking, RIFT is not limited to Clos variations only. The protocol preconditions only a sense of 'compass rose direction' achieved by configuration of levels and other topologies are possible within this framework. So, conceptually, one could include leaf to leaf links and even shortcut between layers but certain requirements in Section 3 will not be met anymore. As an example, shortcutting levels illustrated in Figure 9 will lead either to suboptimal routing when L0 sends traffic to L1 (since using S0's default route will lead to the traffic being sent back to A0 or A1) or the leafs need each other's routes installed to understand that only A0 and A1 should be used to talk to each other.

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

6.3. Originating Non-Default Route Southbound

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

7. Security Considerations

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

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

8. Information Elements Schema

This section introduces the schema for information elements.

On schema changes that

1. change field numbers or

2. add new required fields or

3. change lists into sets, unions into structures or

4. change multiplicity of fields or

5. change datatypes of any field or

6. changes default value of any field

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

Thrift serializer/deserializer MUST not discard optional, unknown fields but preserve and serialize them again when re-flooding.

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

The schema is normative.

8.1. common.thrift

```
/**
    Thrift file for common definitions in RIFT
*/

namespace * models


typedef i64    SystemIDType
typedef i32    IPv4Address
/** this has to be of length long enough to accommodate prefix */
typedef binary IPv6Address
typedef i16    UDPPortType
typedef i32    TIENrType
typedef i32    MTUSizeType
typedef i32    SeqNrType
/** lifetime in seconds */
typedef i32    LifeTimeInSecType
typedef i16    LevelType
typedef i16    PodType
typedef i16    VersionType
typedef i32    MetricType
```

```
   typedef string KeyIDType
   /** node local, unique identification for a link (interface/tunnel
     * etc. Basically anything RIFT runs on). This is kept
     * at 32 bits so it aligns with BFD [RFC5880] discriminator size.
     */
   typedef i32    LinkIDType
   typedef string KeyNameType
   typedef i8     PrefixLenType
   /** timestamp in seconds since the epoch */
   typedef i64    TimestampInSecsType
   /** security nonce */
   typedef i64    NonceType
   /** adjacency holdtime */
   typedef i16    HoldTimeInSecType

   /** fixed leaf level that will not participate in election */
   const LevelType   leaf_level              = 0
   const LevelType   default_level           = 0
   const PodType     default_pod             = 0
   const LinkIDType  undefined_linkid        = 0
   const MetricType  default_distance        = 1
   /** any distance larger than this will be considered infinity */
   const MetricType  infinite_distance       = 0x70000000
   /** any element with 0 distance will be ignored,
    *  missing metrics will be replaced with default_distance
    */
   const MetricType  invalid_distance        = 0
   const bool overload_default               = false
   const bool flood_reduction_default        = true
   const bool leaf_2_leaf_procedures_default = false
   const HoldTimeInSecType default_holdtime  = 3
   /** 0 is illegal for SystemID */
   const SystemIDType IllegalSystemID        = 0

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

   enum AddressFamilyType {
      Illegal               = 0,
      AddressFamilyMinValue  = 1,
      IPv4     = 2,
      IPv6     = 3,
```

```
      AddressFamilyMaxValue  = 4,
   }

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

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

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

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

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

   /** @note: route types which MUST be ordered on preference
    *  PGP prefixes are most preferred attracting
    *  traffic north (towards spine)
    *  normal prefixes are attracting traffic south (towards leafs),
    *  i.e. prefix in NORTH PREFIX TIE is preferred
    */
   enum RouteType {
       Illegal           = 0,
       RouteTypeMinValue = 1,
       /** First legal value. */
       /** Discard routes are most preferred */
       Discard           = 2,

       /** Local prefixes are directly attached prefixes on the
```

```
         *  system such as e.g. interface routes.
         */
        LocalPrefix       = 3,
        /** advertised in S-TIEs */
        SouthPGPPrefix    = 4,
        /** advertised in N-TIEs */
        NorthPGPPrefix    = 5,
        /** advertised in N-TIEs */
        NorthPrefix       = 6,
        /** advertised in S-TIEs */
        SouthPrefix       = 7,
        RouteTypeMaxValue = 8
    }
```

8.2. encoding.thrift

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

include "common.thrift"

namespace * models

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

include "common.thrift"

namespace rs models
namespace py encoding

/** represents protocol major version */
const i32 protocol_major_version = 3
/** represents protocol minor version */
const i32 protocol_minor_version = 0

/** common RIFT packet header */
struct PacketHeader {
    1: required common.VersionType major_version = protocol_major_version;
    2: required common.VersionType minor_version = protocol_minor_version;
    /** this is the node sending the packet, in case of LIE/TIRE/TIDE
        also the originator of it */
    3: required common.SystemIDType  sender;
    /** level of the node sending the packet, required on anything but LIEs.
     *  Lack of presence on LIEs indicates auto-election of level.
```

```
     */
    4: optional common.LevelType     level;
}

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

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

/** Capabilities the node supports */
struct NodeCapabilities {
    /** can this node participate in flood reduction,
        only relevant at level > 0 */
    1: optional bool         flood_reduction = common.flood_reduction_default;
    /** does this node support leaf-2-leaf procedures,
        only relevant in case node has no southbound
        adjacencies, otherwise ignored */
    2: optional bool         leaf_2_leaf_procedures = common.leaf_2_leaf_procedu
res_default;
}

/** RIFT LIE packet

    @note this node's level is already included on the packet header */
struct LIEPacket {
    /** optional node or adjacency name */
    1: optional string                   name;
    /** local link ID */
    2: required common.LinkIDType        local_id;
    /** UDP port to which we can flood TIEs, same address
        as the hello TX this hello has been received on
     */
    3: required common.UDPPortType       flood_port;
    /** layer 3 MTU */
    4: required common.MTUSizeType       link_mtu_size;
    /** this will reflect the neighbor once received */
    5: optional Neighbor                 neighbor;
    6: optional common.PodType           pod = common.default_pod;
    /** optional nonce used for security computations */
    7: optional common.NonceType         nonce;
    /** optional node capabilities shown in the LIE. The capabilies
        MUST match the capabilities shown in the Node TIEs, otherwise
```

```
        the behavior is unspecified. A node detecting the mismatch
        SHOULD generate according error.
     */
    8: optional NodeCapabilities          capabilities;
    /** required holdtime of the adjacency, i.e. how much time
        MUST expire without LIE for the adjacency to drop
     */
    9: required common.HoldTimeInSecType  holdtime = common.default_holdtime;
}

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

/** ID of a TIE

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

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

/** A sorted TIDE packet, if unsorted, behavior is undefined */
struct TIDEPacket {
    /** all 00s marks starts */
    1: required TIEID           start_range;
    /** all FFs mark end */
    2: required TIEID           end_range;
```

```
    /** _sorted_ list of headers */
    3: required list<TIEHeader> headers;
}

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

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

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

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

/** Description of a node.

    It may occur multiple times in different TIEs but if either
        * capabilities values do not match or
        * flags values do not match
        * neighbors repeat with different values
    the behavior is undefined and a warning
    SHOULD be generated.

    @note: observe that absence of fields implies defined defaults
*/
struct NodeTIEElement {
    1: required common.LevelType                                level;
    2: optional NodeCapabilities                                capabilities;
    3: optional NodeFlags                                       flags;
    /** optional node name for easier operations */
    4: optional string                                          name;
    /** if neighbor systemID repeats in other node TIEs of same node
        the behavior is undefined */
```

```
    5: required map<common.SystemIDType,NodeNeighborsTIEElement> neighbors;
}

/** multiple prefixes */
struct PrefixTIEElement {
    /** prefixes with the associated cost.
        if the same prefix repeats in multiple TIEs of same node
        or with different metrics, behavior is unspecified */
    1: required map<common.IPPrefixType,common.MetricType> prefixes;
}

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

/** single element in a TIE. enum common.TIETypeType
    in TIEID indicates which elements MUST be present
    in the TIEElement. In case of mismatch the unexpected
    elements MUST be ignored.
 */
union TIEElement {
    /** in case of enum common.TIETypeType.NodeTIEType */
    1: optional NodeTIEElement            node;
    /** in case of enum common.TIETypeType.PrefixTIEType */
    2: optional PrefixTIEElement          prefixes;
    3: optional KeyValueTIEElement        keyvalues;
    /** @todo: policy guided prefixes */
}

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

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

/** protocol packet structure */
```

```
struct ProtocolPacket {
    1: required PacketHeader  header;
    2: required PacketContent content;
}
```

9. IANA Considerations

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

10. Security Considerations

Security mechanisms will be addressed in upcoming versions of this specification.

11. Acknowledgments

Many thanks to Naiming Shen for some of the early discussions around the topic of using IGPs for routing in topologies related to Clos. Adrian Farrel, Joel Halpern and Jeffrey Zhang provided thoughtful comments that improved the readability of the document and found good amount of corners where the light failed to shine. Kris Price was first to mention single router, single arm default considerations. Jeff Tantsura helped out with some initial thoughts on BFD interactions while Jeff Haas corrected several misconceptions about BFD’s finer points. Artur Makutunowicz pointed out many possible improvements and acted as sounding board in regard to modern protocol implementation techniques RIFT is exploring.

12. References

12.1. Normative References

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

[RFC1142] Oran, D., Ed., "OSI IS-IS Intra-domain Routing Protocol", RFC 1142, DOI 10.17487/RFC1142, February 1990, <https://www.rfc-editor.org/info/rfc1142>.

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

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

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

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

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

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

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

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

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

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

[RFC6234] Eastlake 3rd, D. and T. Hansen, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)", RFC 6234, DOI 10.17487/RFC6234, May 2011, <https://www.rfc-editor.org/info/rfc6234>.

[RFC6822] Previdi, S., Ed., Ginsberg, L., Shand, M., Roy, A., and D. Ward, "IS-IS Multi-Instance", RFC 6822, DOI 10.17487/RFC6822, December 2012, <https://www.rfc-editor.org/info/rfc6822>.

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

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

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

12.2. Informative References

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

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

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

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

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

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

[QUIC] Iyengar et al., J., "QUIC: A UDP-Based Multiplexed and Secure Transport", 2016.

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

Authors' Addresses

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

Email: prz@juniper.net

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

Email: Alankar_Sharma@comcast.com

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

Email: jdrake@juniper.net

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

Email: akatlas@juniper.net