Networking Working Group
Internet-Draft
Intended status: Standards Track
Expires: July 15, 2017

T. Przygienda
J. Drake
A. Atlas
Juniper Networks

January 11, 2017

RIFT: Routing in Fat Trees
draft-przygienda-rift-00

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 blackholing 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 http://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 July 15, 2017.

Copyright Notice

Copyright (c) 2017 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1. Introduction

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 succesfully 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] 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 not common in "fat-trees".

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 necesssary 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.

Deaggregation/Disaggregation Process in which a node decides to advertise certain prefixes it received in N-TIEs to prevent blackholing and suboptimal routing upon link failures.

LIE: This is an acronym for a "Link Information Element", largely equivalent to HELLOs in IGPs.

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

2.2. Topology

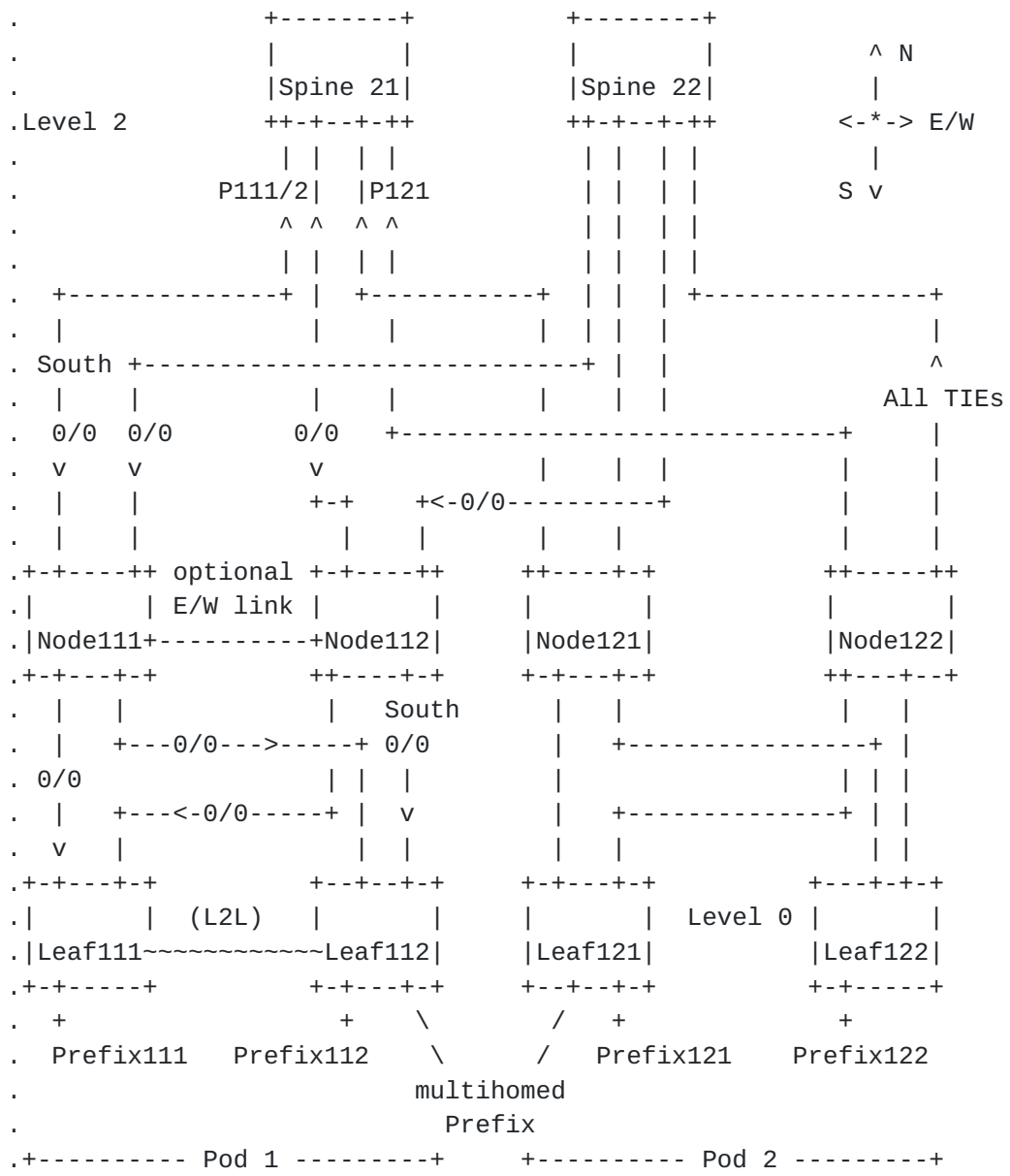

Figure 1: A two level spine-and-leaf topology

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 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 of the underlay when addresses are moving in the topology.

REQ2: High degree of ECMP (and ideally non equal cost) must be supported.

REQ3: Traffic engineering should be allowed by modification of prefixes and/or their next-hops.

REQ4: The control protocol must discover the physical links automatically and be able to detect cabling that violates fat-tree topology constraints. It must react accordingly to such miscabling attempts, at a minimum preventing adjacencies between nodes from being formed and traffic from being forwarded on those miscabled links. E.g. connecting a leaf to a spine at level 2 should be detected and ideally prevented.

REQ5: The solution should allow for access to link states of the whole topology to allow efficient support for modern control architectures like SPRING [RFC7855] or PCE [RFC4655].

REQ6: The solution should easily accomodate 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.

REQ7: 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 of necessary information should be as small as viable.

REQ8: The protocol should allow for maximum aggregation of carried routing information while at the same time automatically deaggregating the prefixes to prevent blackholing in case of failures. The deaggregation should support maximum possible ECMP/N-ECMP remaining after failure.

REQ9: 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.

REQ10: 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.

REQ11: 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. Taking a path through the spine in cases where a shorter path is available is highly undesirable.

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: We carrry parallel links with unique identifer carried in node TIEs. Link bundles (i.e. parallel links between same set of nodes) must be distinguishable for SPF and traffic engineering purposes. But further, do we rely on coalesced links from lower layers and BFD/m-BFD detection or hello all links ?

PEND3: BFD will obviously play a big role in fast detection of failures and the interactions will need to be worked out.

PEND4: What is the maximum scale of number leaf prefixes we need to carry. Is 0.5E6 enough ?

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. LIE exchange happens over well-known multicast address with a TTL of 1. TIE exchange mechanism uses address and port indicated by each node in the LIE exchange with TTL of 1 as well.

All packet formats are defined in Thrift or protobuf models.

4.2.2. Link (Neighbor) Discovery (LIE Exchange)

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

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

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

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

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

A node configure 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]. 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].

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. They 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 readvertisements of TIEs.

More information about the TIE structure can be found in the schema in Section 7.

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 south or to the north/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 4.2.4 and Section 4.2.5.

As an example to illustrate databases holding both representations, consider the topology in Figure 1 with the optional link between Node111 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 KeyValueElements and the PolicyGuidedPrefixesElements which may be included in an S-TIE or an N-TIE are not shown.

```
Spine21 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)))
SouthPrefixesElement(prefixes(0/0, cost 1), (::/0, cost 1))

Node111 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)))
SouthPrefixesElement(prefixes(0/0, cost 1), (::/0, cost 1))

Node111 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(...))))
NorthPrefixesElement(prefixes(Node111.loopback)

Node121 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)))
SouthPrefixesElement(prefixes(0/0, cost 1), (::/0, cost 1))

Node121 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(...))))
NorthPrefixesElement(prefixes(Node121.loopback)

Leaf112 N-TIE:
NodeLinkElement(layer=0,
neighbors((Node111, layer 1, cost 1, links(...)),
(Node112, layer 1, cost 1, links(...))))
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. TIEs themselves are transported over UDP with the ports indicates in the LIE exchanges.

Once QUIC [QUIC] achieves the desired stability in deployments it may prove a valuable candidate for TIE transport.

4.2.3.4. TIE Flooding Scopes

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.

It should be noted that east-west links are included in N-TIE flooding; they need to be flooded in case the level above the current level is disconnected from one or more nodes in the current level and southbound SPF desires to use those links as backup in case of some switches in the spine being partitioned in respect to some PoDs.

A node's S-TIEs, consisting of a node's adjacencies and a default IP prefix, 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 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. A node does not send an S-TIE northbound if it is from the same or lower level. No S-TIEs are propagated southbound.

Node S-TIE "reflection" allows to support disaggregation on failures describes in Section 4.2.4 and flooding reduction in Section 4.2.3.7.

Observe that a node does not reflood S-TIE received from the lower level towards other southbound nodes which has implications on the way TIDEs are generated in the southbound direction.

As an example to illustrate these rules, consider using the topology in Figure 1, with the optional link between Node111 and Node 112, and

the associated TIEs given in Figure 2. The flooding from particular nodes of the TIEs is given in Table 1.

Router floods to	Neighbor	TIEs
Leaf111	Node112	Leaf111 N-TIE, Node111 S-TIE
Leaf111	Node111	Leaf111 N-TIE, Node112 S-TIE
Node111	Leaf111	Node111 S-TIE
Node111	Leaf112	Node111 S-TIE
Node111	Node112	Node111 S-TIE, Node111 N-TIE, Leaf111 N-TIE, Leaf112 N-TIE, Spine21 S-TIE, Spine22 S-TIE
Node111	Spine21	Node111 N-TIE, Node112 N-TIE, Leaf111 N-TIE, Leaf112 N-TIE, Spine22 S-TIE
Node111	Spine22	Node111 N-TIE, Node112 N-TIE, Leaf111 N-TIE, Leaf112 N-TIE, Spine21 S-TIE
Node121	Leaf121	Node121 S-TIE
Node121	Leaf122	Node121 S-TIE
Node121	Spine21	Node121 N-TIE, Leaf121 N-TIE, Leaf122 N-TIE, Spine22 S-TIE
Node121	Spine22	Node121 N-TIE, Leaf121 N-TIE, Leaf122 N-TIE, Spine22 S-TIE
Spine21	Node111	Spine21 S-TIE
Spine21	Node112	Spine21 S-TIE
Spine21	Node121	Spine21 S-TIE
Spine21	Node122	Spine21 S-TIE
Spine22	Node111	Spine22 S-TIE
Spine22	Node112	Spine22 S-TIE
Spine22	Node121	Spine22 S-TIE
Spine22	Node122	Spine22 S-TIE

Table 1: Flooding some TIEs from example topology

4.2.3.5. Initial and Periodic Database Synchronization

The initial exchange of RIFT is modelled after ISIS with TIDE being equivalent to CSNP and TIRE playing the role of PSNP. The content of TIDEs in north and south direction will contain obviously just the according database variant and reflect the flooding scopes defined.

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 in its database until it restarts, even if the TIE looses all content. The re-advertisement of empty TIE fullfills 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. Optional Automatic Flooding Reduction and Partitioning

Two 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 set of north and south neighbor adjacencies and support flooding reduction 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)).

Additional rules for flooding reduction and partitioning:

a. A node always floods its own TIEs

b. 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.

c. 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 N-TIE.

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. Automatic Disaggregation on Link & Node Failures

Under normal circumstances, a node S-TIEs contain just its 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 blackholed. Even when not blackholing, 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'.

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

a. 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. Overload bits as introduced in Section 4.2.6.2.1 have to be respected during the computation.

b. 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).

c. 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.

d. 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 blackholing 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 via at least one 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, 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 readvertise 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
     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 3: 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:

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

b. 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.

c. disaggregated 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.

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

4.2.5. 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 northmost layer. It is also necessary to to have northbound policy-guided prefixes to guide traffic from the northmost 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.

The set of policy-guided prefixes received in a TIE is subject to ingress filtering and then regenerated to be sent out in the receiver's appropriate TIE. Both the ingress filtering and the regeneration 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.

4.2.5.1. Ingress Filtering

When a node X receives a PGP S-TIE or N-TIE that is originated from a node Y which does not have an adjacency with X, such a TIE MUST be discarded. Similarly, if node Y is at the same layer as node X, then X MUST discard PGP S- and N-TIEs.

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.

4.2.5.2. 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.

4.2.5.3. 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 tiebreaking 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 readvertisement of a new best PGP.

4.2.5.4. Regeneration

A node must regenerate 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 regenerate southbound policy-guided prefixes.

4.2.5.5. 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 infeasiblity 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 intermittant blackholes.

4.2.6. 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.

4.2.6.1. Specification

A node uses the N-TIEs to build a network graph with unidirectional links. As in IS-IS or OSPF, unidirectional links are associated together to confirm bidirectional connectivity. Because of the requirement that a packet traversing in a southbound direction must not go take any northbound links, a node has topological visibility only south of itself. There are no links at the computing node's level that go to a northbound level. Therefore, all paths computed must contain only east-west and southbound links. To enforce this, the network graph MUST have either its northbound unidirectional links removed or set to have a cost of COST_INFINITY.

A node runs a standard shortest path first (SPF) algorithm on this network graph. If a node is minimized to have a cost of COST_INFINITY, then it is not reachable.

4.2.6.1.1. Attaching Prefixes

After the SPF is run, it is necessary to attach prefixes. 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.

Prefixes from each S-TIE need to also be added to the RIFT route database. There is no SPF to be run. Instead, the computing node needs 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 distance and 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 prefered route so ignore
          end if
        end if
      end for
   end for
```

Figure 4: Adding Routes from S-TIE Prefixes

4.2.6.1.2. 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 4.2.5.3; 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 prefered route so ignore
            end if
          end if
      end for
```

Figure 5: Adding Routes from Policy-Guided Prefixes

4.2.6.2. Further Mechanisms

4.2.6.2.1. Overload Bit

The leaf node SHOULD set the 'overload' bit on its N-TIE, 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.

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

4.2.6.2.2. 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 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 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 4.2.6.1.1 as well as the policy-guided prefixes as in Section 4.2.6.1.2.

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

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

b. 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 prefered.

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 readvertised 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.

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):

- Spine 21 (PoD 0) to Node 111, Node 112, Node 121, and Node 122
- Spine 22 (PoD 0) to Node 111, Node 112, Node 121, and Node 122
- Node 111 to Leaf 111, Leaf 112
- Node 112 to Leaf 111, Leaf 112
- Node 121 to Leaf 121, Leaf 122
- 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 multihomed 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 multihomed 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

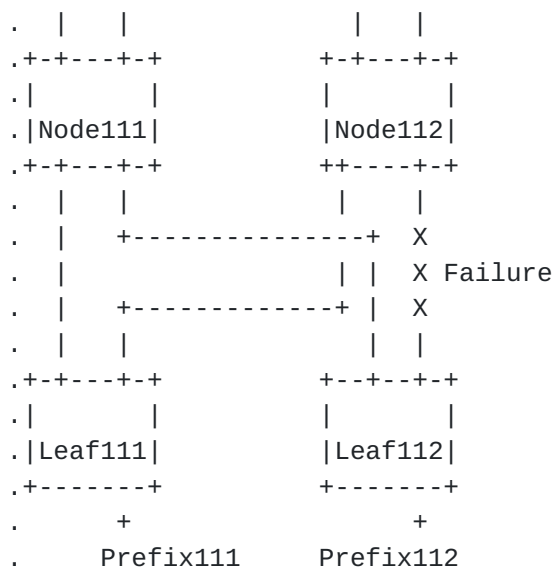

Figure 6: Single Leaf link failure

In case of a failing leaf link between node 112 and leaf 112 the link-state information will cause recomputation 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 deaggregate Prefix 111 and Prefix 112 but we will not describe it further here since deaggregation is emphasized in the next example. It is worth observing however in this example that if Leaf111 would keep on forwarding traffic towards Prefix112 using the advertised south-bound default of Node112 the traffic would end up on Spine21 and Spine22 and cross back into Pod1 using Node111. This is arguably not as bad as blackholing present in the next example but clearly undesirable. Fortunately, deaggregation prevents this type of behavior except for a transitory period of time.

5.3. Partitioned Fabric

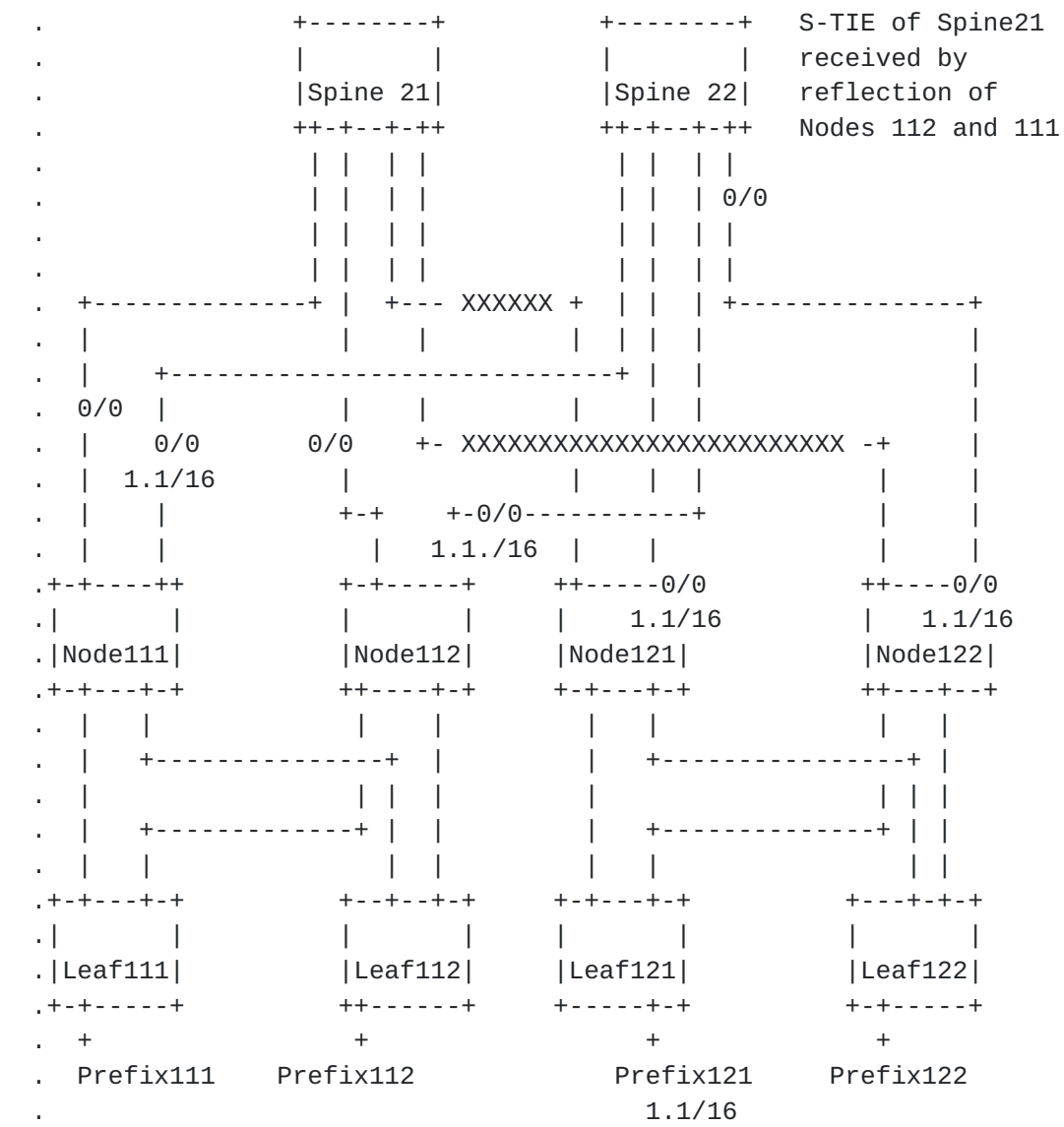

Figure 7: Fabric partition

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 P121 would suffer 50% blackholing 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 blackholing 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 the new S-TIEs containing 1.1/16 and reflection of those by node 111 and node 112 again. 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 4.2.4, 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.

6. Implementation and Operation: Further Details

6.1. Leaf to Leaf connection

[QUESTION] This would imply that the leaves have to understand the N-TIE format and pull out the prefixes to figure out the next-hop... Do we want this complexity?[/QUESTION]

6.2. 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.

6.3. Address Family and Topology

Multi-Topology (MT)[RFC5120] and Multi-Instance (MI)[RFC6822] is used today in link-state routing protocols to provide the option of several instances 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.

7. Information Elements Schema

This section introduces the schema for information elements.

On schema changes that

a. change field numbers or

b. add new required fields or

c. change lists into sets, unions into structures or

d. change multiplicity of fields or

e. change datatypes of any field or

f. 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.

```
//! Thrift file for RIFT, flooding for fat trees
//! @note: all numbers are implementation co'erced to unsigned versions using
the highest bit

/// represents protocol major version
const i32 CURRENT_MAJOR_VERSION = 1
const i32 CURRENT_MINOR_VERSION = 0

typedef i64    SystemID
typedef i32    IPv4Address
/// this has to be of length long enough to accomodate prefix
typedef binary IPv6Address
typedef i16    UDPPortType
```

```
typedef i16    TIENrType
typedef i16    MTUSizeType
typedef i32    SeqNrType
typedef i32    LifeTimeType
typedef i16    LevelType
typedef i16    PodType
typedef i16    VersionType
typedef i32    MetricType
typedef i64    KeyIDType
typedef i32    LinkIDType
typedef string KeyNameType
typedef bool   TieDirectionType

const LevelType   DEFAULT_LEVEL    = 0
const PodType     DEFAULT_POD      = 0
const LinkIDType  UNDEFINED_LINKID = 0

/// RIFT packet header
struct PacketHeader {
    1: required VersionType major_version = CURRENT_MAJOR_VERSION;
    2: required VersionType minor_version = CURRENT_MINOR_VERSION;
    3: required SystemID  sender;
    4: optional LevelType level = DEFAULT_LEVEL;
}

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

union Content {
    1: optional LIE          hello;
    2: optional TIDEPacket   tide;
    3: optional TIREPacket   tire;
    4: optional TIEPacket    tie;
}

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

// content per N-S direction
union TIEElement {
    1: optional NorthTIEElement north_element;
    2: optional SouthTIEElement south_element;
}
```

```
// @todo: flood header separately in UDP ?
// to allow caching to TIEs while changing lifetime?
struct TIEPacket {
    1: required TIEHeader  header;
    // North and South TIEs need the correct union
    // member to be sent, otherwise content is ignored
    2: required TIEElement element;
}

enum TIETypeType {
    Illegal              =   0,
    TIETypeMinValue      =   1,
    NodeTIEType          =   2,
    NorthPrefixTIEType   =   3,
    SouthPrefixTIEType   =   4,
    KeyValueTIEType      =   5,
    NorthPGPrefixTIEType =   6,
    SouthPGPrefixTIEType =   7,
    TIETypeMaxValue      =   8,
}

/// RIFT LIE packet
struct LIE {
    2: optional string          name;
    3: required SystemID        originator;
    // UDP port to which we can flood TIEs, same address
    // as the hello TX this hello has been received on
    4: required UDPPortType     flood_port;
    5: optional Neighbor        neighbor;
    6: optional PodType         pod = DEFAULT_POD;
    // level is already included on the packet header

}

struct LinkID {
    1: required LinkIDType      local_id;
    2: required LinkIDType      remote_id;
    // more properties of the link can go in here
}

struct Neighbor {
    1: required SystemID        originator;
    2: required UDPPortType     flood_port;
    // ignored on LIE
    // can carry description of multiple
    // parallel links in a TIE
    3: optional set<LinkID>     link_ids;
}
```

```
/// ID of a TIE
/// @note: TIEID space is a total order achieved by comparing the elements in
sequence defined
struct TIEID {
    /// indicates whether N or S-TIE, True > False
    1: required TieDirectionType    northbound;
    2: required SystemID            originator;
    3: required TIETypeType         tietype;
    4: required TIENrType           tie_nr;
}

struct TIEHeader {
    2: required TIEID        tieid;
    3: required SeqNrType    seq_nr;
    // in seconds
    4: required LifeTimeType lifetime;
}

// sorted, otherwise protocol doesn't work properly
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;
}

struct TIREPacket {
    1: required set<TIEHeader> headers;
}

struct NodeNeighborsTIEElement {
    /// if neighbor systemID repeats in set or TIEs
    /// the behavior is undefined
    1: required SystemID     neighbor;
    2: required LevelType    level;
    3: optional MetricType   cost = 1;
}

/// capabilities the node supports
struct NodeCapabilities {
    1: required bool         flood_reduction = true;
}

/// flags the node sets
struct NodeFlags {
    1: required bool         overflow = false;
```

```
}
```

```
struct NodeTIEElement {
    1: required LevelType                    level;
    2: optional NodeCapabilities             capabilities;
    3: optional NodeFlags                    flags;
    4: required set<NodeNeighborsTIEElement> neighbors;
}

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

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

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

struct PrefixWithMetric {
    1: required IPPrefixType   prefix;
    2: optional MetricType     cost = 1;
}

struct PrefixTIEElement {
    /// if the same prefix repeats in multiple TIEs
    /// or with different metrics, behavior is unspecified
    1: required set<PrefixWithMetric> prefixes;
}

struct KeyValue {
    1: required KeyIDType keyid;
    2: optional KeyNameType key;
    3: optional string value = "";
}

struct KeyValueTIEElement {
    1: required set<KeyValue>    keyvalues;
}

/// single element in a N-TIE
union NorthTIEElement {
    /// hinges of enum TIETypeType::NodeTIEType
    1: optional NodeTIEElement            node;
    /// hinges of enum TIETypeType::NorthPrefixTIEType
```

```
    2: optional PrefixTIEElement          prefixes;
    /// @todo: policy guided prefixes
}

union SouthTIEElement {
    /// hinges of enum TIETypeType::NodeTIEType
    1: optional NodeTIEElement            node;
    2: optional KeyValueTIEElement        keyvalues;
    /// hinges of enum TIETypeType::SouthPrefixTIEType
    3: optional PrefixTIEElement          prefixes;
    /// @todo: policy guided prefixes
}
```

8. IANA Considerations

9. Security Considerations

10. Acknowledgments

Many thanks to Naiming Shen for some of the early discussions around the topic of using IGPs for routing in topologies related to Clos. Adrian Farrel 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.

11. References

11.1. Normative References

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

[RFC1142] Oran, D., Ed., "OSI IS-IS Intra-domain Routing Protocol", RFC 1142, DOI 10.17487/RFC1142, February 1990, <http://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, <http://www.rfc-editor.org/info/rfc2119>.

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

[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, <http://www.rfc-editor.org/info/rfc4271>.

[RFC4655] Farrel, A., Vasseur, J., and J. Ash, "A Path Computation Element (PCE)-Based Architecture", RFC 4655, DOI 10.17487/RFC4655, August 2006, <http://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, <http://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, <http://www.rfc-editor.org/info/rfc5303>.

[RFC5309] Shen, N., Ed. and A. Zinin, Ed., "Point-to-Point Operation over LAN in Link State Routing Protocols", RFC 5309, DOI 10.17487/RFC5309, October 2008, <http://www.rfc-editor.org/info/rfc5309>.

[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, <http://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, <http://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, <http://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, <http://www.rfc-editor.org/info/rfc7938>.

11.2. Informative References

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

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

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

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

[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

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