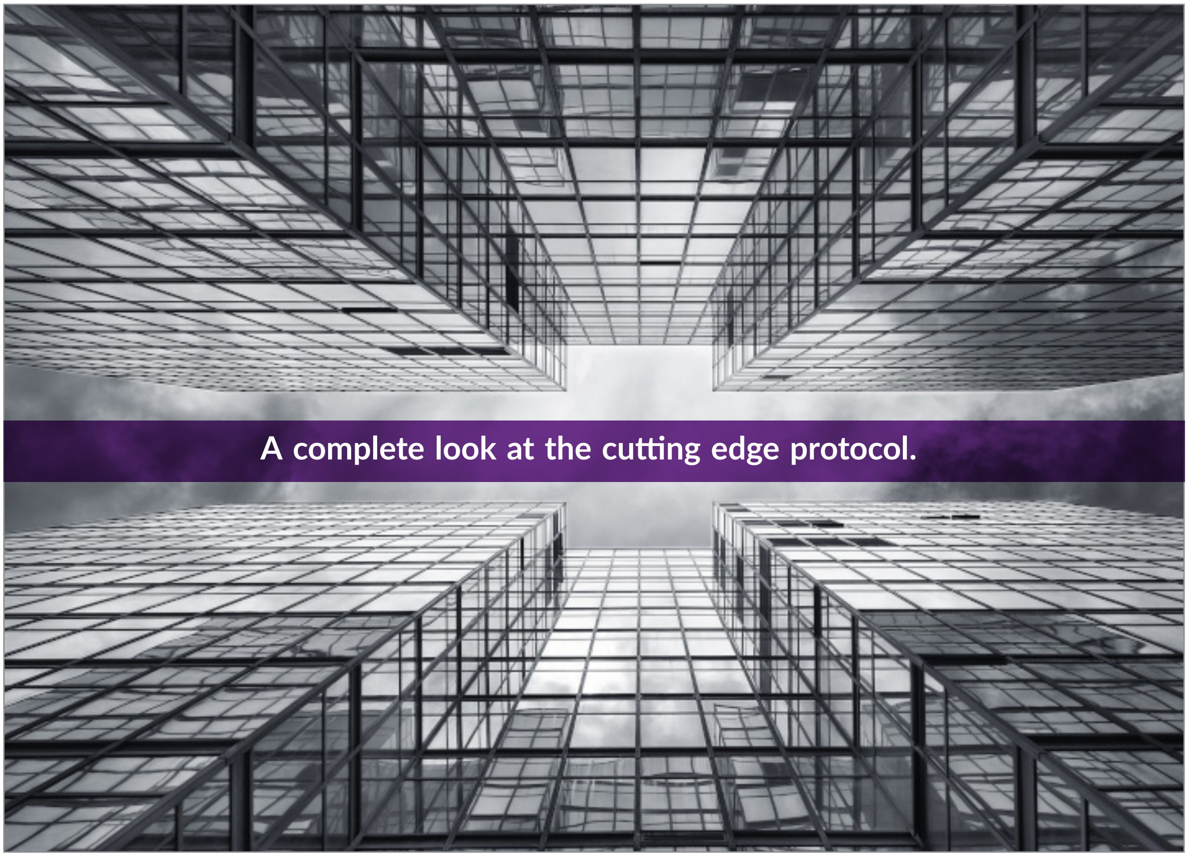
DAY ONE: ROUTING IN FAT TREES (RIFT)



A complete look at the cutting edge protocol.

By Melchior Aelmans, Olivier Vandezande, Bruno Rijsman,
Jordan Head, Christian Graf, Leonardo Alberro,
Hitesh Mali, Oliver Steudler

DAY ONE: ROUTING IN FAT TREES (RIFT)

The need for a radical new approach to data center IP fabric routing resulted in a 'from the ground up' designed protocol that leverages the best of existing protocols and solutions to challenges that couldn't be solved. RIFT is ready to take on the new challenges that will come to the data center: it's scalable, it's an open standard, and it's designed for a variety of new use cases. This book will help you understand RIFT and provide design, configuration, monitoring, and troubleshooting guidance.

"A new routing protocol from the IETF is always an event. There are so few of them; and this one is loaded with innovations that make it especially suitable for the anisotropic nature of modern data center networks. I wish RIFT all the success of its predecessors, and hope to see the day when all major vendors and open source distributions provide their own interoperable implementation. Working with extraordinary people like Bruno and Tony was an incredibly enriching experience, and a lot of fun. Many thanks, guys!"

Pascal Thubert, Cisco Systems

"RIFT is specifically engineered for data center fabric underlay routing. While RIFT is a modern routing protocol, it's carefully built from well-established concepts such as distance vector and link state protocols. It's built with zero touch provisioning (day 1), zero touch maintenance (day 2), and security, all integral to the protocol. Authors have done a fabulous job covering implementation of the protocol at different levels of the network, all the way up to hosts. This book will show you how to get your fabric networking to a better place with a transparent, optimized underlay."

Arun Viswanathan, VP Engineering Routing, Juniper Networks

"RIFT is designed from the ground up as a means to redefine data center routing. It includes advanced features, self-optimizations, and it fills the voids left by other routing protocols. RIFT could be an autonomous all-in-one routing solution for enterprise to hyper-scale data centers. Thanks to the standards community for recognizing and contributing towards RIFT as a standard." *Alankar Sharma, Comcast*

IT'S DAY ONE AND YOU HAVE A JOB TO DO, SO LEARN HOW TO:

- Get to know all the ins and outs of the new RIFT protocol.
- Install and configure RIFT on various platforms.
- Understand both Juniper and open source RIFT implementations.
- Review key design considerations using RIFT from key professionals.

ISBN 978-1-7363160-0-9
54000>
9 781736 316009

JUNIPER
NETWORKS

Day One: Routing in Fat Trees

by Melchior Aelmans, Olivier Vandezande, Bruno Rijsman, Jordan Head, Christian Graf, Hitesh Mali, Leonardo Alberro, Oliver Steudler

Published by Juniper Networks Books
Authors: Melchior Aelmans, Olivier Vandezande, Bruno Rijsman, Jordan Head, Christian Graf, Hitesh Mali, Leonardo Alberro, Oliver Steudler
Technical Reviewers: Russ White, Tony Przygienda, Matthew Jones
Editor in Chief: Patrick Ames
Illustrator: Valentijn Flik

ISBN 978-1-7363160-0-9 (Paperback)
Printed in the USA by Vervante Corporation.

ISBN 978-1-7363160-1-6 (ePub)

Version History: v1, December. 2020
2 3 4 5 6 7 8 9 10
Comments, errata: dayone@juniper.net

About the Authors

Melchior Aelmans is a Consulting Engineer at Juniper Networks, where he has been working with many operators on the design and evolution of their networks. He has over 15 years of experience in various operations, engineering, and systems engineering positions with Cloud Providers, Data Centers and Service Providers. Melchior enjoys evangelizing and discussing routing protocols, routing security, internet routing and peering and data center routing. He participates in IETF and RIPE and is a regular attendee and presenter at other conferences and meetings, and is a board member at the NLNOG foundation.

Olivier Vandezande is a Resident Engineer working for Juniper Advanced Services in Switzerland. He has more than 20 years of consulting with hands-on experience as a consultant for Professional Services with a track record of delivering projects from consulting, design to execution helping Enterprise and Service Provider customers understand, design, configure, test, automate and troubleshoot a wide range of network related technologies. His diverse practical experience is underpinned by top industry awarded certifications. Olivier initially started his career as a software developer on OpenVMS platform.

Bruno Rijsman is a network and software architect with over 25 years of experience in technical and leadership roles at network equipment vendors, most recently Juniper Networks. He has worked on the implementation of a broad range of networking software, including several routing protocols, MPLS, software defined networking, broadband subscriber management, video streaming, and others. He currently spends his time on open source software development (including the open source implementation of RIFT described in this book) and quantum networking (https://www.scientificamerican.com/article/the-quantum-internet-is-emerging-one-experiment-at-a-time/).

Jordan Head is a Senior Resident Engineer working for Juniper Advanced Services designing, supporting, and automating service provider networks in the United States. His career started at Amazon Web Services in 2008, working for teams that designed, deployed, and automated both the data center and network infrastructure that underpin existing cloud services. Jordan went on to other cloud and service providers to help solve challenges that are presented by operating and automating networks at such a scale.

Christian Graf is Senior Consulting Engineer at Juniper Networks. He started 25 years ago as Project-Manager covering Planning and Installation of Network-Infrastructure and moving towards the Enterprise as Systems Engineer. Since many years Christian is assigned to Tier-1 and Tier-2 Service-Provider Core and Edge networks looking into all kinds of MPLS-VPNs, fast convergence, High-Availability, Segment-Routing, IP-Fabrics and virtualization.

Hitesh Mali is a Consulting Engineer at Juniper who has a love for network designing and enjoys experimenting with various emerging networking technologies. Hitesh has spent the last decade helping Cable MSO (Multiple System Operators) in building network solutions.

Leonardo Alberro is a Professor and Master Student at Universidad de la República in Montevideo, Uruguay, where he has been working on research and development projects in the networking area. After finishing his engineering degree in the networking and security area, he started working in data center networking. His master thesis is focused on routing scalability in massive data centers. Leonardo enjoys teaching the basic concepts of computer networking to the future computer engineers at the Universidad de la República.

Oliver Steudler is a Consulting Engineer at Juniper Networks responsible for Service Providers in Switzerland and Austria. As a seasoned professional with 20+ years of experience in networking he has co-authored two books on network security published by Syngress (now Elsevier).

Author Acknowledgments

The authors would like to thank, in random order: Patrick Ames, Russ White, Tony Przygienda, and Matthew Jones (Juniper Networks), Pascal Thubert (Cisco) and Jeff Tantsura (Apstra), and the many others who helped in some form with their valuable contributions, content, input, software, comments, and mental support. Many others supported us in various forms who may not be mentioned but to whom we are all very grateful!

Melchior Aelmans would especially like to thank Bruno Rijsman for contributing to this book and writing a chapter on the open source RIFT implementation.

Bruno Rijsman would like to thank Tony Przygienda and Pascal Thubert for the numerous discussions and email exchanges on RIFT; most of what I know about RIFT is thanks to them. And I would like to thank Mariano Scazzariello and Tommaso Caiazzi for implementing negative disaggregation in RIFT-Python and doing scaling testing of RIFT-Python using Kathará (https://www.kathara.org/).

Leonardo Alberro would like to thank Maximiliano Lucero and Agustina Parnizari for their excellent work on the RIFT dissector. Also I would like to thank Eduardo Grampín and Alberto Castro for their remarkable support and contribution.

Reviewers: Russ White, Tony Przygienda, and Matthew Jones. Thank you very much for all your time and effort!

Welcome to Day One

This book is part of the *Day One* library, produced and published by Juniper Networks Books. *Day One* books cover the Junos OS and Juniper Networks network administration with straightforward explanations, step-by-step instructions, and practical examples that are easy to follow.

- Download a free PDF edition at https://www.juniper.net/dayone
- PDF books are available on the Juniper app: Junos Genius
- Purchase the paper edition at Vervante Corporation (www.vervante.com).

Key RIFT Resources

While writing this book the authors were greatly influenced by the existing work previously done by other authors. We'd like to thank and recognize the authors of the following works and encourage readers to explore them:

- https://datatracker.ietf.org/doc/draft-ietf-rift-rift/
- https://datatracker.ietf.org/doc/draft-ietf-rift-applicability/
- https://www.juniper.net/documentation/en_US/junos/topics/topic-map/rift-in-junos.html
- https://github.com/brunorijsman/rift-python
- https://www.juniper.net/documentation/en_US/junos/topics/topic-map/sdn-vxlan.html
- https://ipj.dreamhosters.com/wp-content/uploads/2020/09/232-ipj-2.pdf

A What You Need to Know Before Reading This Book

- You need a basic understanding of datacenter underlay routing protocols.
- This book assumes that you have some hands-on experience with IP fabrics.
- You need a basic understanding of Juniper routers and switches and some Junos OS knowledge.
- You need to be open to radically new ideas and approaches to data center underlay architectures.

After Reading This Book You Will

- Get to know all the ins and outs of the new RIFT protocol.
- Install and configure RIFT on various platforms.
- Understand both Juniper and open source RIFT implementations.
- Review key design considerations using RIFT from key professionals.

How This Book Is Set Up

This book is organized in six chapters and an Appendix.

Chapter 1 provides an introduction to data center routing and IP fabrics.

Chapter 2 introduces the Routing in Fat Trees protocol.

Chapter 3 introduces the Juniper RIFT implementation and deployment.

Chapter 4 discusses Junos RIFT monitoring and troubleshooting.

Chapter 5 explains the Wireshark RIFT Dissector.

Chapter 6 details the open source RIFT implementation.

Appendix A elaborates the RIFT definitions.

Appendix B covers REDIS.

NOTE Some device output and Junos configurations have a reduced font size in order to fit its width on the page.

Foreword

The famous quote attributed to Henry Ford: "*If I had asked people what they wanted, they would have said faster horses,*" instead, they got what they really needed, faster and better transportation.

Invention and evolution of RIFT follows the similar pattern – the ability to think out of the box, vision, perseverance, and incredibly deep domain knowledge to build the right solution for the problem.

Carefully done trade-off analysis with an open-minded approach - let's invent but also reuse the best parts of existing technologies that work well, lead to an elegant combination of distance-vector and link-state routing characteristics where they are most needed: minimum state at the bottom of the fabric, maximum at the top. Add to that ability to conditionally disaggregate on an event and complete ZTP, and there's very little left to wish for. RIFT has come very close to a state of perfection within the topological constraints it has been designed to address.

It has been a great pleasure to chair the RIFT working group and work with incredibly talented and innovative individuals that ultimately made RIFT possible!

This is just the beginning of the journey. Many other exciting technologies will be developed and further enhanced on top of RIFT. Think: Multicast, Segment Routing, just to name just a few.

Excited? Ready to start building tomorrow's solution to today's problems? Go and install RIFT on your laptop/server/switch, it's available, open, and ready!

Jeff Tantsura, IETF Routing in Fat Trees Workgroup Chair
December 2020

Preface

Over the years and especially with the advent of cloud and multicloud applications, increasing demand is causing data center requirements to change faster than ever. Workloads need to be able to move between servers both within a single data center or between multiple data centers, regardless of whether they are public or private. Also the majority of traffic nowadays is intra-datacenter resulting in a much higher demand for East-West bandwidth capacity. And with edge computing, 5G, ML/AI, and the ever increasing rise of over the top services the end isn't even in sight yet.

In this new cloud computing era existing protocols as we know them simply are not up for the job anymore as they don't scale to fit the increasing demand. In 1953 Charles Clos already thought about scaling multi-stage non-blocking circuit-switching networks for telecom usage. These same architectures are still in use in today's data centers successfully connecting the world's most critical infrastructure.

Besides the physical design, several routing protocols are used to arrange transportation of the actual IP traffic from one point in the data center to another or to the edge of the network in order to forward it to an external (for example, internet) destination. Some of today's protocols find their roots in the pre-IP era and have evolved in order to try to meet current needs. Tailoring a protocol to the needs for (hyper) scale cloud computing is only feasible up to a certain point. After that one needs to start over, rethink, and redesign it from scratch in order to meet the new demands and requirements.

Several attempts to scale networks out vertically with bigger chassis, higher port count per box, and modifying existing (older) protocols have resulted in mixed results that only cover parts of the issues, solve some point problems, have seen limited implementations but still weren't able to bring the needed scale.

Technical arguments aside, it goes without saying that data center and network operators will have to comply with the continued pressure to bring down capital and operational spending. Especially those offering services to third-parties or attaching a cost to their services that need to 'deliver more for less'.

Both operators and vendors concluded there is a need for a radically new approach to the datacenter IP fabric routing challenges. This resulted in a 'from the ground up' designed protocol that leverages the best of existing protocols and brings maybe even more solutions to challenges that couldn't be solved. Routing in Fat Trees (RIFT) is ready to take on the new challenges that will come to the datacenter: it is a scalable, open standard that's been designed for a variety of new use cases.

The authors of this book hope that these pages will help you to understand RIFT and provide guidance for building the protocol's business case, design, configuration, monitoring and troubleshooting.

Melchior Aelmans, et. al.
December 2020

Chapter 1

Data Center Routing and IP Fabrics

Why is routing in IP fabrics a special problem?

Since the initial publication of the drafts resulting in RFC7938 "*Use of BGP for Routing in Large-Scale Data Centers*", BGP has been the default option for (large) data center fabrics, assumed by most controllers, intent-based systems, training courses in data center (DC) fabrics, and implementers.

But recent activity in both the IETF and implementers suggests using link-state protocols in DC fabrics instead of BGP for various reasons, such as speed of convergence and telemetry data. The main challenge using a link-state protocol in a very large IP fabric is flooding of routing information, both for reachability and topology. In particular, link-state protocols are (too) good at flooding and so can be very chatty. Some initiatives to address the problem are work in progress and focus on distributed (or localized) optimized flooding in IS-IS and centralized calculation of optimal flooding trees.

MORE? Examples of this work in IETF can be found here:

- https://datatracker.ietf.org/doc/draft-white-distoptflood/
- https://datatracker.ietf.org/doc/draft-lsr-isis-flood-reflection/
- https://datatracker.ietf.org/doc/draft-ietf-lsr-dynamic-flooding/

Distance-vector routing and link-state routing protocols together only partially address the challenges IP fabric operators face. To resolve a wider range of problems, a new routing protocol, Routing in Fat Trees (RIFT), was invented. One could say RIFT brings the best of both and then adds some solutions to address very specific IP fabric challenges.

After reading this introduction readers might conclude the authors do not believe BGP or any link-state protocol should be used as the routing protocol for a DC fabric underlay—this is not the case. Ultimately, it is up to the individual operator to decide which protocol is "the best" for their application, a decision which should be based on business and operational needs.

1.1 BGP in the Underlay

While the reasons for using BGP in the underlay have been outlined in several places through the years, including RFC7938, let's quickly recap and explore some of these reasons as background.

1.1.1. Advantages of using BGP in Data Center Underlay

First, BGP is widely implemented. Virtually every routing vendor and every open-source routing stack (FRRouting, BIRD, OpenBGPD) has a fairly complete and well-tested BGP implementation. Operators can be confident that no matter whose hardware and software they choose, BGP will be supported—and the implementation is likely to be mature, interoperable with other implementations, and running in many production networks.

Second, BGP was—at least at one time—conceived of as one of the most straightforward routing protocols to understand and implement. The logic of path-vector is reasonably easy to understand and implement correctly, and the underlying transport mechanism, TCP, was already well understood.

Third, BGP is widely deployed, and hence well understood by operators. Operators consider it easier to hire someone who knows BGP than any other protocol, and it is easy to find tooling for operating BGP in the open-source community. There is a bit of irony in this point as ten years ago it was almost impossible to find engineers with solid BGP experience; the advent of BGP on large-scale data center fabrics has become something of a "self-fulfilling prophecy" in this regard.

Fourth, where scale is an issue, the perception is BGP outshines every other protocol. After all, "BGP runs the Internet", and you cannot ask for a better proof point of scalability than that. The initial implementations of BGP on large-scale DC fabrics originally tried various IGPs and found they could not scale to the size required.

Fifth, BGP has extensive prefix filtering, route tagging, and traffic engineering capabilities. No other protocol, other than perhaps EIGRP, can match BGP's very rich set of options for filtering, changing route parameters, and ability to control route flow.

Sixth, BGP can be used for both the underlay and the overlay in a single network. In theory, this makes the configuration simpler. The normal configuration when using BGP for both is to configure the underlay using eBGP peering and the overlay as iBGP peering.

With all these advantages, why would an operator decide to move away from using BGP in both the underlay and overlay?

1.1.2. Challenges with BGP in the Data Center Underlay

There are counterpoints to many of the advantages of using BGP as an underlay protocol. Beginning with the second—BGP is one of the simplest routing stacks to implement. With the advent of multiple address families, RPKI, EVPN, VPLS, MPLS traffic engineering, BGP Link-State (BGP-LS), and the many other features which have been "piled into" BGP over the last twenty years, BGP implementations have exploded in complexity. By now BGP may be the most complex protocol to implement and maintain among all the routed control planes today.

BGP is also widely used for carrying external routes and policy between autonomous systems and through Internet Exchange Points (IXPs). As, for example, it does not auto-discover new peers, BGP defaults for operation in these situations. It's defaults for operating in a data center fabric, however, are the opposite of those desired for inter-AS peering.

It is easy enough to create a single knob that turns on a group of features at once. It is not so easy to hide the increased complexity—and the higher chance of a defect in the code or a misconfiguration of some kind—resulting from these kinds of changes. BGP is strongly automatable—but it will take massive work to make it autonomic (https://en.wikipedia.org/wiki/Autonomic_computing). The protocol originally wasn't designed with today's demanding tools and complexity in mind. One also needs to keep in mind that making large changes to the BGP protocol to make it more autonomic in data center deployments potentially destabilizing BGP in the internet.

At some point, the "routing community" needs to decide if it is wise to make one protocol the "protocol to end all protocols." Is a single solution the right answer for all problems? Or is it better to move back towards developing multiple parallel protocols to support different use cases? This criticism may not apply to operators building their private implementation of BGP for use on their DC fabrics—but these kinds of implementations are few and far between.

A second issue in the same neighborhood is the amount of specialized configuration required to allow BGP to converge quickly on the kinds of dense topologies used for DC fabrics that Figure 1.1 illustrates.

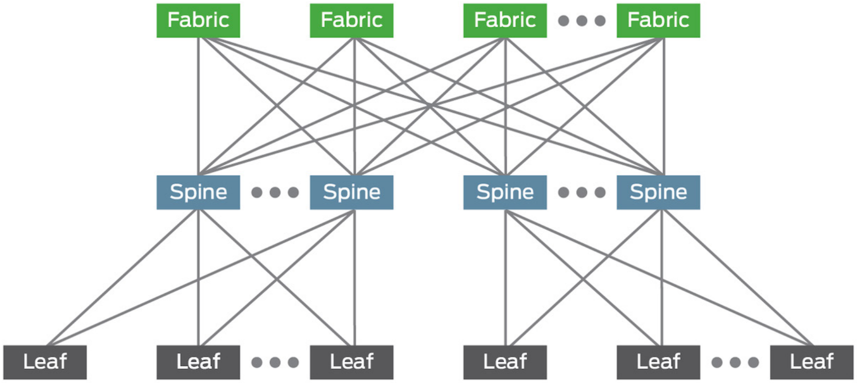


Figure 1.1 A Small IP Fabric / Clos

Note that in Figure 1.1, leafs are Top of Rack (ToR) switches or compute nodes partaking in the fabric. The fabric is the Top of Fabric (ToF). Two spine switches and three leafs form a pod.

How BGP converges depends on the kind of topology change. In the case of a single router or link failure, BGP can converge almost as quickly as an IGP, given the failure timers are tuned correctly and if BFD and other underlying mechanisms are in use, etc. The case of a withdraw from the edge of the network, however, is much different.

In the case of a withdraw, BGP converges by hunting across available paths, starting from the shortest and ending in the longest. This hunt happens because of the timing of processing and forwarding updates. To prevent loops, a BGP speaker must process an update locally, modifying the routing table before it can forward the update to its peers. Longer paths just take longer for withdrawals to traverse than shorter ones. This withdraw behavior can be a problem in at least two situations: when a workload is moved from one location on the fabric to another, and when an anycast address representing a service instance is removed from the fabric. In these cases, the slow convergence time of BGP can impact applications running on the fabric.

Reducing the impact of the hunt is fairly easy—the key is to reduce the length of the paths through which BGP must "search." The easiest way to do this is to block the reflection of updates and withdraws through the network; for instance, leaf switches in Figure 1.1 should not reflect any withdraws or updates upwards to any of its peers from the spine layer. On the leaf nodes filters should only permit BGP updates to the spines with an empty AS path (^$). There are many ways to accomplish this, but a common method is to create filters on each layer. As you can imagine this means lots of additional (potentially manual) configuration.

With these changes, BGP is essentially converted into “fancy RIP,” and the time required to withdraw a route (or move it from one place to another in the fabric) can be reduced to about one minute in large scale fabrics. It is possible to modify BGP to converge more quickly, but this returns the discussion to the first argument—is creating a single protocol to solve all problems really the right answer? When is the complexity of the BGP code “complex enough” to start considering other options?

There are two other points to consider before moving on to Link-State protocols in DC fabric underlays. One of the advantages listed for BGP is its many different policy options, such as route filtering and tagging. If the underlay is really designed to provide undifferentiated IP connectivity, these policies do not seem like much of a real advantage. Policy, such as route tagging and filtering, should be moved to the overlay—which is most likely going to be BGP anyway.

A final point is this—providers and data center operators split infrastructure and customer routes to separate these two kinds of information into different failure domains. One misunderstanding about failure domains is they must be “absolute” and “complete,” where the two failure domains are completely decoupled at every point, if they are effective. This is not, however, always the case as it is likely impossible to build networks out of completely decoupled failure domains. Instead, this is a matter of tradeoffs; how much gain is there in separating these two kinds of information in this way, versus how difficult is it to separate these two kinds of information, and how much deoptimization is likely to occur?

In a data center fabric, it is possible to separate underlay (infrastructure) routes from overlay (customer) routes to form different failure and maintenance domains potentially separating the protocols as well. This also creates another administrative domain, opening the possibility of allowing customers to control some aspects of the reachability information in the overlay without posing a risk to the underlay.

1.2. Link-state in the Underlay

Link-state protocols, like IS-IS and OSPF, are also widely implemented and understood. Every commercial routing stack and many open-source routing stacks include an implementation of OSPF or IS-IS, and these implementations are mature, well tested, and widely deployed—however, most of these implementations, like BGP, are not optimized for use on DC fabrics. This section considers the positive aspects of using a link-state protocol in a DC fabric, some of the challenges operators face when deploying standard link-state protocols on DC fabrics, and realistic expectations for scale when using these unmodified implementations. The following sections will consider modified link-state protocols currently being designed and implemented, and the probable scaling characteristics of these implementations.

The first advantage link-state protocols have over BGP in DC fabrics is convergence speed—but the irony is link-state protocols are at their fastest where BGP is at its slowest, and vice versa. Link-state protocols are most challenged at scale during initial convergence because of the density of the topology through which flooding must take place. Consider the network in Figure 1.1 when a leaf node originates a new link-state Update – it will send the update to every router in the spine row. Every spine, in turn, will send the update to every leaf router, which will then, in turn, send the LSU to every spine router. The number of copies each fabric device receives depends primarily on timing, but in topologies including 2000 plus fabric devices, each one was observed receiving more than forty copies of each LSU. Nonetheless, unmodified link-state protocols converge at their worst as fast or faster than BGP up to some scale, where scale includes both the number of devices (nodes in the Shortest Path Tree, or SPT) and the number of reachable destinations. To what scale? The number varies but based on prior large-scale deployments, thousands (or more) devices in a fabric with hundred-thousand routes is not unreasonable within a single flooding domain. Optimizations will increase the scaling numbers somewhat—though to what degree will depend on many factors.

Where link-state protocols converge much faster than BGP is when a reachable destination either moves from one place on the fabric to another or is disconnected from the fabric entirely. From the perspective of IS-IS, any reachable destination changes are just change in leaf connectivity, which means the destination can just be removed from the SPT without running Shortest Path First (SPF). This is called a *partial SPF*; it is extremely fast and requires minimal processing on each of the fabric devices.

The second advantage link-state protocols have over BGP in DC fabrics is topology visibility. Link-state protocols require each device to maintain a full view of the topology, which must be synchronized with every other router in the network (or rather flooding domain); this is called the Link-State Database (LSDB).

When a controller is used, in order to obtain a copy of the LSDB, it only needs to connect to one router (or two routers, for resilience) in the fabric. This kind of information is useful for traffic engineering and traffic steering. Further, periodic snapshots of the network topology from the perspective of the control plane can be a useful mine of telemetry information.

The first challenge for link-state protocols in the DC fabric is scaling, mainly related to flooding. Link-state protocols are, by design, very chatty and will cause heavy flooding in large scale fabrics. Several ways to reduce the number of LSUs each device receives are work in progress in IETF but are out of scope for this book for now.

Another problem often cited in this area is the impression that link-state protocols can drop or fail to deliver LSUs—that flooding is periodic, rather than reliable, and the period is long enough to allow significant problems to develop. All link-state protocols, however, include reliability mechanisms to deliver flooded packets. For instance, IS-IS tracks whether each neighbor has received an LSU through acknowledgments and will retransmit LSUs until they are acknowledged. IS-IS can also send a description of the entire database periodically to ensure a neighbor's LSDB is correctly synchronized. OSPF has similar mechanisms.

Two other challenges link-state protocols face are scaling the number of reachable destinations and the time required to run the SPF algorithm used to calculate the set of loop-free paths. Faster processors combined with well-designed and tested implementations of SPF, along with optimizations such as partial SPF, have largely mitigated these concerns up to much larger scales than many engineers realize. Link-state protocols will never scale to the same levels as BGP but they will scale enough to support a large proportion of the DC fabrics operators will build.

1.3 Routing in Fat Trees in the Underlay

Routing in Fat Trees (RIFT) is a recent addition to the list of underlay protocol options, combining link-state and distance-vector concepts. Link-state-like operation is retained as information is transmitted up the fabric towards the Top of Fabric (ToF), while distance-vector-like operation carries reachability and topology information towards the edges of the fabric, the leaves.

RIFT has been designed from the ground up to tailor specifically to spine-and-leaf fabrics needs. It is also designed to keep in mind multi-threaded implementations as CPUs are not getting faster anymore but they do include more cores.

1.4 Underlay Conclusion

BGP has been and will continue to be an important option for DC fabric underlays for many years to come. BGP may eventually offer some of the interesting features link-state protocols already offer, such as faster convergence and closer-to-autonomic deployment.

On the other hand, some features of a link-state protocol, such as the ability to grab a complete view of the entire topology from a single place – pulling a copy of the LSDB – are going to be very difficult to replicate in BGP, and BGP's convergence speed is always likely to lag behind a link-state protocol. Table 1.1 summarizes some of the differences between the options for operators.

Table 1.1 Feature Comparison

Feature	BGP tailored for DC fabrics using policy and filters	IS-IS (modified for DC fabrics)	RIFT
Peer Discovery	Partial	Yes	Yes
Automatic Tier Calculation	No	Potentially	Yes
Mis-cabling Detection	No	Capability in progress	Yes
Required configuration	Loopback address, peering; can be reduced with protocol modifications; can be automated	System ID, loopback address; can be automated or locally calculated	ToF state and others; can be automated
Aggregation; Default only on ToR and Below	Manually configured	No	Yes
Scales to underlay routing on host	Yes	Depends on fabric size and implementation	Yes
High ECMP Fanout Support	Yes	Yes	Yes
Unequal Cost Load Sharing	Yes (in some implementations)	No	Yes
Full View of Topology	No (partly with BGP-LS)	Yes	Yes
Carry Opaque Configuration Data	No (can carry opaque information through communities)	No (can carry opaque information through tags)	Yes
Drain Node without Disruption	Yes	Yes	Yes
Automatic Disaggregation	No	No	Yes
Fast Convergence Speed	Partial (Depends on event type)	Yes	Yes
Security	Origin Validation based on prefix/ASN	No	Yes
Overlay Support	Assumes single protocol (eBGP underlay, iBGP/ EVPN overlay)	Assumes EVPN overlay	Supports EVPN overlay, can operate pure L3 fabric with no overlay to the workload
Support for general topologies (not just DC fabrics)	Yes	Yes	No, but RIFT fabrics could be used in many different scenarios

The remainder of this book will focus purely on RIFT. The authors however would like to offer some pointers to further reading on the IETF work addressing IS-IS flooding, BGP auto-discovery, etc.

MORE? This work can be found here (a non-exclusive list):

- https://datatracker.ietf.org/doc/draft-white-distoptflood/
- https://datatracker.ietf.org/doc/draft-ietf-lsr-dynamic-flooding/
- https://datatracker.ietf.org/doc/draft-decraene-lsr-isis-flooding-speed/
- https://datatracker.ietf.org/doc/draft-xu-idr-neighbor-autodiscovery/

1.5 Business Advantages and Drivers

From a business decision point of view the single most important question with regards to adoption of a new technology is *What will it bring me and how will the business benefit?*

In particular those responsible for the budget motivations are not easily convinced by the “technical shiny shiny” (thanks James Bensley, https://twitter.com/jwbensley, for the quote).

A one-line answer to this could be, “RIFT provides more useful features and capabilities, with less operational overhead.” As the inventor of RIFT, Tony Przygienda would say: “It just works.”

This could potentially be a huge cost-saver as not having to configure each node individually is a timesaver. Time that could be used to deploy additional, revenue generating, new services. To throw in another quote, Bruno Rijsman, “You should not have to worry anymore about your data center underlay than you do about the plumbing in your building.”

Reducing operational overhead reduces human errors. RIFT has built-in true Zero Touch Provisioning, which auto detects the level/topology, identifies mis-cabling, and implements North-and-Southbound routing policies. RIFT’s default routing policies put very low requirements on the leaf/ToR switches because of sparse routing.

BGP has no understanding of the underlying interface speed. BGP can nicely perform ECMP, but honoring correctly the underlying interface bandwidth is non-trivial. RIFT has true ECMP capabilities and loadshares by default across any interface available in weighted fashion.

Many underlay-designs rely on EBGP, as it removes the need for an IGP. The cost of EBGP are policies per node and each device needs its own private AS, which again is an operational burden. RIFT shares the scaling-advantage of BGP, however there is no need to provision a per-device unique AS with its policies.

RIFT has been also designed specifically to allow "true IP routing" all the way down to (multihomed) servers while respecting the fact that server NIC silicon should not be required under normal conditions to store all the routes on the fabric.

So in short, RIFT scales as nicely as other protocols, is easier to operate, and does so with less operational overhead and at lower cost.

1.5.1 Physical Infrastructure and Cabling

Since physical interaction is required to cable devices together in the correct manner, human error will always be a factor. However, in densely connected IP fabrics, identifying when something is mis-cabled is challenging, tedious, and time consuming, requiring engineering staff to validate the interface and protocol operation. The validation itself is also error prone and can cause collateral damage when examining optical cabling and transceivers.

If the issue persists after initial troubleshooting, on-site staff must be engaged again at which point the process repeats until everything is fixed. Optical cabling and transceivers are rather sensitive as well, so with increased interaction the chances of degradation or failure increases. Risk of collateral damage to other optical infrastructure in the vicinity also increases.

RIFT incorporates built-in mis-cabling detection into its ZTP functionality that eliminates the challenge of identifying mis-cabled links. Additionally, it carries lots of information that seems trivial but simplifies operational troubleshooting such as node, interface, and instance names that are readily visible on both sides of the link.

1.5.2 Network Provisioning

RIFT requires *very* minimal configuration in that you only need to configure which devices are considered Top-of-Fabric. If the devices downstream of the Top-of-Fabric nodes are cabled correctly they will automatically provision. Adjacencies won't form on mis-cabled or mis-configured links. This will be discussed later on.

In contrast, traditional underlay protocols require a fair amount of initial configuration in order to deploy. A number of well-known identifiers and variables, such as the router ID and AS number, must be correctly assigned, deployed, and maintained on a per device basis. Automation does help to make this dramatically easier, but reliable automation can be expensive to develop and maintain, especially as scale requirements increase. And automation works best when nodes are already reachable, which requires a working IGP.

Consider augmenting the fabric scale or replacing a device. At the very least, these situations require the addition of configuration on all devices involved, including making certain these identifiers are properly accounted for to prevent duplication and facilitate policies and troubleshooting. RIFT's ZTP functionality does not require any configuration in such cases but does allow for manual configuration and even for a mix of ZTP and manually configured nodes in a fabric without restrictions.

Whether you're deploying a new underlay, scaling an existing one, or fixing a broken one, all that's required is proper cabling when the device is connected.

1.5.3 Multihoming, Load Balancing and Routing on the Host

Service outages or degradation can easily cause revenue loss. As such, a significant investment is normally committed to ensure that services remain highly available. That availability could be incorporated into the network, the applications, or infrastructure subsystems such as power and cooling plants. The degree of availability ("how many nines are desired") also adds complexity and further increases cost.

Most services and applications hold state to perform their work, state that cannot be moved or accessed when the network has failed. Real-time replication of application state across the network in real time or near-real time is available for some classes of services like databases, but it is generally difficult to implement and affects application performance. Building truly stateless services unaffected by infrastructure failures is incredibly difficult and ultimately shifts the problem to messaging systems holding service state. These messaging systems become very fragile if the delay and loss characteristics of the underlying network are not nearly perfect.

Many operators, as the first step, focus on increasing resilience in the physical infrastructure, such as power and cooling systems. This is obviously never a bad idea, but is incredibly expensive and still leaves ToR switches as a single point of failure and the weakest link in fabric resiliency. The failure or upgrade of a ToR switch to which a server is single-homed results in the application losing state.

It is possible to multihome servers to different ToRs to mitigate that problem but unfortunately doing so requires the use of fragile or proprietary protocols such as STP variants or MC-LAG. Running these protocols means that software and hardware requirements for the ToRs go up, which could mean increased hardware and licensing cost. This is especially true if Active-Active load balancing is required.

Even if those solutions are deployed, they force the use of Level 2 homing with Level 3 starting at the ToR level leading to a security 'air-gap' and to stacked tunneling architectures if the server is originating native Level 3 services, which becomes more pronounced with virtualization technologies and micro-segmentation.

Deploying native IP routing on the hosts would solve these and many other problems such as choosing the correct outgoing link to route around failures. Further, it would introduce advantages like load balancing traffic depending on available bandwidth, complete ZTP, and visibility of servers being introduced into the fabric in the routing databases. Lastly, it would also extend a Level 3 trust model all the way to the leaf where the overlay can originate encrypted tunnels utilizing the capabilities of smart NICs.

As good as the idea sounds, the introduction of servers into Level 3 routing multiplies the necessary scale of the routing architecture by a very significant multiplier roughly equivalent to the number of servers on a ToR. This is not an insignificant number and pretty quickly puts traditional IGPs out the picture and making deployment of complicated, band-aided BGP even more torturous to address.

Since RIFT has been designed from day one with the vision of supporting servers being part of the underlay or “Routing on the Host” (ROTH), deploying it on servers is possible and addresses all the concerns above.

Servers as well as any other RIFT nodes would also benefit from other RIFT-native benefits. To start with, RIFT introduces bidirectional equal and weighted unequal cost load balancing. This means that once failures occur RIFT will weight flows across the remaining links to account for the reduced bandwidth. OSPF and IS-IS are simply not capable of doing this and while BGP can to an extent, it is far from easy.

Further, RIFT is designed to run on hosts without any configuration altogether which is not only convenient but goes a long way to minimize the attack surface in security terms. RIFT even allows you to roll-over a whole fabric to a new key and prevent connection of compromised, pre-factored servers. Lastly, in operation terms, it is a distinct advantage to see all the servers attached to the fabric including, for example, their IP/MAC bindings, without use of out-of-band tools.

1.5.4 Convergence, Failures, and Troubleshooting

RIFT has a particularly efficient approach to routing and associated flooding: it combines the best of both link-state and distance-vector concepts. Routing information that is flooded from leaf to ToF nodes (north) behaves like a link-state protocol. As information propagates northbound, each subsequently higher level in the fabric contains a full topological view of all southbound nodes. Routing information that is flooded from ToF to leaf nodes (south) behaves as a distance-vector protocol by only advertising a default route unless link failures have to be addressed.

In short, RIFT maintains only the absolute minimum routing information that is required to establish reachability in the fabric. This may allow a greater reduction in hardware costs and better ROI the further south you go in the network, compounded by the fact that leaf nodes are the most ubiquitous.

As mentioned, RIFT behaves both as a link-state and distance-vector protocol, except it doesn't suffer from any of the major pitfalls. Failures in a link-state protocol propagate to the entire network, while this does mean that convergence happens quite quickly, flooding becomes a serious problem. RIFT only floods routing information to the devices that absolutely need it, keeping blast radius to an absolute minimum.

BGP on the other hand suffers from slower convergence behavior known as *path hunting*. This is where, as a failure propagates through a network the path will become longer and longer until traffic is ultimately blackholed. This is made much worse in the case of a more specific prefix failing where a larger aggregate exists because typical longest match routing will cause instability for the larger aggregate until the path hunting behavior is over and the fabric is converged. Basically, impact to a smaller part of the network, can impact a much larger part that is otherwise unaffected.

RIFT eliminates this problem with its disaggregation mechanism, since a prefix is only disaggregated when failures occur, troubleshooting becomes much easier compared to OSPF, IS-IS, or BGP.

The rest of this book will give you facts, examples, and proof of the welcomed appearance of RIFT in the fabric.

Chapter 2

Routing in Fat Trees Protocol

Work on the RIFT protocol officially started in IETF when the RIFT working group charter was approved in February 2018. The charter states:

"The Routing in Fat Trees (RIFT) protocol addresses the demands of routing in Clos and fat-tree networks via a mixture of both link-state and distance-vector techniques colloquially described as link-state towards the spine and Distance-Vector towards the leaves. RIFT uses this hybrid approach to focus on networks with regular topologies with a high degree of connectivity, a defined directionality, and large scale."

The working group was chartered to create a protocol that would:

- Deal with automatic construction of fat-tree topologies based on detection of links.
- Minimize the amount of routing state held at each topology level.
- Automatically prune topology distribution exchanges to a sufficient subset of links.
- Support automatic disaggregation of prefixes on link and node failures to prevent black-holing and suboptimal routing.
- Allow traffic steering and re-routing policies.
- Provide mechanisms to synchronize a limited key-value data-store that can be used after protocol convergence.

According to the charter:

"It is important that nodes participating in the protocol should need only very light configuration and should be able to join a network as leaf nodes simply by connecting to the network using the default configuration. The protocol must support IPv6 and should also support IPv4."

As briefly described earlier, RIFT combines concepts from both link-state and distance-vector protocols, but this one liner description is way too short to cover all the unique aspects of RIFT and to describe all of its (unique) features. A deep dive into all these details is to be found in this chapter. Subsequent chapters will focus on both the open source and Juniper Networks implementation specifics.

2.1 Topology Considerations

One of the reasons Clos and fat-tree topologies have become the de facto standard for IP fabrics is the advent of a significant increase in traffic between servers *within* the data center (East/West) as opposed traffic *leaving or entering* the data center (North/South). Spine/leaf variants make it possible for each service's traffic flows to traverse the shortest path, meet capacity needs, and remain highly resilient to failures.

In the context of reachability information, despite the East/West nomenclature, the traffic is actually moving North/South. Northbound from leaf nodes at the bottom of the fabric to the nodes at the top of the fabric and then southbound in the reverse direction. If you consider the required reachability information to facilitate this server-to-server traffic pattern, it's quite minimal. For example, in a 3-stage Clos, leaf nodes don't require more than a default route to reach spine nodes and spine nodes don't need the *entire* routing table to reach leaf nodes, just the information describing nodes one level south. This is why RIFT behaves as a link-state protocol northbound and a distance-vector protocol southbound.

While data centers were the first modern IP networks to reap the benefits of a Clos or fat-tree based IP fabric, that does not mean a Clos or fat-tree topology is required to reap the benefits of RIFT. As long as the topology has a sense of top and bottom (north and south), RIFT should be considered for other types of network deployments, such as metro, enterprise, or telco cloud.

This sense of directionality is required so that the topology can be sorted into different levels, that is to say that nodes at the top of the fabric remain at the highest level and nodes at the bottom (leaf nodes) stay at the lowest level. This facilitates RIFT's routing model as well as other features like mis-cabling detection through ZTP.

Furthermore, even though a Clos or a fat-tree topology naturally fits into this category, implementations are not required to be a perfect representation of either topology. RIFT's specification can be relaxed to support horizontal or vertical shortcuts between different levels in the fabric.

2.2 Fundamental Operations

The following subchapters describe the operation of the RIFT protocol. These basics and concepts are fundamental in order to understand the working of the protocol. While reading you might encounter new abbreviations which are explained in the text itself or defined in the Appendix.

2.2.1 Neighbor Discovery

RIFT automatically discovers neighbors, negotiates Zero Touch Provisioning (ZTP), and detects any mis-cablings through the exchanging of Link Information Elements (LIE).

LIE messages are encoded in an "envelope" that supports authentication and provides increased security.

Table 2.1 Default RIFT Transport

Address Family	Default Multicast Address	Destination Port
IPv4	224.0.0.120	UDP/914
IPv6	FF01::A1F7	UDP/914

LIEs are always sent with a TTL (IPv4) value or HL (IPv6) value 1 in order to prevent reaching beyond a single layer 3 hop.

As a precursor to further detail, a brief overview of some of the critical fields in the LIE header is beneficial as listed in the next few tables.

Table 2.2 LIE Header Overview

Field	Use
Local ID	Local ID of the link
MTU	Layer 3 MTU of the local link, which is used to discover MTU mismatches
PoD	Local node's PoD value
Neighbor	Used to "reflect" a neighboring nodes system ID and link ID

Now let's take a look at the adjacency formation process itself.

Table 2.3 *Adjacency States*

State	Meaning
OneWay	Initial state.
TwoWay	Local node has received a valid LIE from the remote node.
ThreeWay	Local node sees its own System ID in the LIE from the remote node. In cases where parallel links are used, link IDs must also match.
MultipleNeighborsWait	Local node sees multiple neighbors on a single link and initiates a hold down timer before processing LIEs.

RIFT nodes will only attempt to form adjacencies in the following circumstances:

Table 2.4 *Adjacency Formation Requirements*

Value	Requirement(s)
PoD	Both node's PoD values must match, OR Either node must advertise a value of 0 (undefined/any).
Major Version	Both node's Major Version field must match.
Existing ThreeWay Adjacencies	Local node cannot have any existing northbound ThreeWay adjacencies with a PoD value that differs from the remote node.
System ID	Both nodes must advertise valid System ID values that must not match.
MTU	Both node's corresponding interfaces must have the same MTU.
Level	Both nodes must advertise level values.
Level Restrictions	Adherence to at least one of the following additional level restrictions is required to ensure the nodes operate at the correct level in the topology. If the local node is a leaf (level 0), it must not have any existing ThreeWay adjacencies to nodes at the Highest Adjacency ThreeWay (HAT) that differs from the remote node. If the local node is not a leaf, the remote node must be a leaf. If both nodes are leaves, they must support leaf-2-leaf functionality. If neither node is a leaf, they are no more than 1 level away from each other.

Figure 2.1 illustrates a simplified scenario where two nodes establish a ThreeWay adjacency.

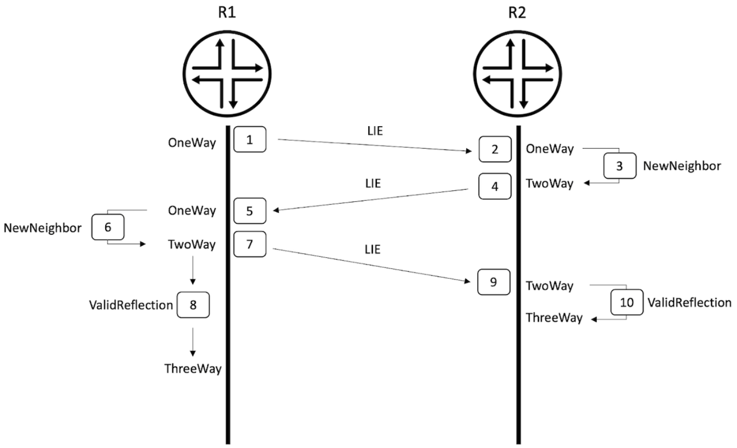


Figure 2.1 ThreeWay Adjacency

NOTE The example LIE messages only show relevant headers.

1. R1 begins in the OneWay state and advertises a LIE message to R2.

LIE Message from R1	
System ID	222000
Level	0
Local ID	1
MTU	1400

2. R2 receives the LIE message from R1 and processes it.

3. R2 has no neighbors on this interface, so a NewNeighbor event is triggered.

4. R2 enters the TwoWay state and advertises a LIE message to R1, this time it contains neighbor information indicating it has a neighbor.

LIE Message from R2	
System ID	111000
Level	1
Local ID	2
MTU	1400
Neighbor Field	
Originator	222000
Remote ID	1

5. R1 receives the LIE message from R2 and processes it.

6. R1 has no neighbors on this interface, so a NewNeighbor event is triggered.

7. R1 enters the TwoWay state and advertises a LIE message to R2, this time it contains neighbor information indicating it has a neighbor.

LIE Message from R1	
System ID	222000
Level	0
Local ID	1
MTU	1400
Neighbor Field	
Originator	111000
Remote ID	2

8. R1 has now seen its own system and link information reflected in the Neighbor field of the LIE message from R2, causing it to trigger a ValidReflectionEvent and finally enter the ThreeWay state. R1 can now begin flooding TIEs.

9. R2 receives the LIE message from R1 and processes it.

10. R2 has now seen its own system and link information reflected in the Neighbor field of the LIE message from R1, causing it to trigger a ValidReflectionEvent and finally enter the ThreeWay state. R2 can now begin flooding TIEs.

2.2.2 Topology Information Elements (TIEs)

Topology Information Elements (TIEs) are used to convey a node's connected adjacencies, prefix information, and capabilities (e.g. flood reduction). Remember that RIFT is based on the concept of directionality (north and south) and depending on a node's location in the fabric, will represent itself differently when advertising TIEs in a given direction. Similarly, TIEs are also referred to by the direction in which they are advertised, specifically, North TIEs (N-TIE) or South TIEs (S-TIE).

Generally, the various types of TIEs carry similar information regardless of the direction they are advertised, with some exceptions.

As with LIE messages, the TIEs also support authentication.

Table 2.5 TIE Types

Type	North TIEs	South TIEs
Node TIE	Node adjacencies and capabilities	Node adjacencies and capabilities
Prefix TIE	Directly reachable prefixes	Originated default prefix Directly reachable prefixes
Positive Disaggregation TIE	N/A	Positively disaggregated prefixes
Negative Disaggregation TIE	N/A	Negatively disaggregated prefixes
External Prefix TIE	External/redistributed prefixes	External/redistributed prefixes
Key-Value TIE	Northbound KVs.	Southbound KVs

2.2.2.1 TIE Exchange

TIEs are exchanged (flooded) over UDP to a destination port learned during LIE negotiation. Flooding takes place within certain scopes of the topology depending upon the direction and type being advertised. All N-TIEs are always flooded northbound in order to present the higher level with the full topological view of the network south of it. This ensures that traffic received on nodes at or below a particular level, will always take the most specific route toward the advertising node. All Node S-TIEs are flooded southbound whereas all *non-node* S-TIEs are only flooded southbound if the local node originated that TIE. This allows for nodes one level down to have the required reachability to the higher advertising level and the rest of the fabric.

It is important to note that RIFT uses N-TIEs and S-TIEs, even on East-West links (i.e. there are no east TIEs or west TIEs) and that they have their own flooding scopes. East-West links will be discussed in a later chapter.

Table 2.6 TIE Flooding Scopes

Type	South	North	East-West
All N-TIE	Never flood	Always flood	Flood only if local node is a ToF node
Node S-TIE	Flood if the originator and the local node's level are the same	Flood if the originator is at a higher level than the local node	Flood only if the local node is not a ToF node
Non-Node S-TIE	Flood only if the local node is the originator	Flood only if the neighbor is the originator	Flood only if the local node is the originator and not a ToF node

2.2.2.2 Link-state Database Synchronization

TIEs alone are not enough to efficiently keep the Link-State Database (LSDB) up to date. RIFT employs somewhat similar mechanisms as IS-IS in order to keep local and remote routing information current, but also incorporates improvements to boost efficiency.

Topology Information Description Elements (TIDEs) are used to advertise a complete directory of TIEs for the given direction, similar to that of an IS-IS Complete Sequence Number PDU (CSNP). Topology Information Request Elements (TIREs) are used to either request missing TIEs or acknowledge received TIEs, similar to that of an IS-IS Partial Sequence Number PDU (PSNP). Unlike TIEs, the originator of a TIDE or TIRE may only flood to a directly connected node and will therefore never be re-flooded. TIDEs and TIREs contain different information based upon the direction in which they are flooded, with Table 2.7 outlining the specific information contained in both.

Table 2.7 TIDE and TIRE Contents

Type	North	South	East-West
TIDE	All N-TIE headers not originated by the local node All S-TIE headers originated by the local node All Node S-TIEs of all nodes at the same level	All N-TIEs All Node S-TIEs All S-TIEs originated by the neighboring node	If the local node is a ToF node, include all N-TIEs If the local node is not a ToF node include only self-originated TIEs
TIRE (request)	Request all N-TIEs Request all of the neighboring node's self-originated TIEs Request all Node S-TIEs	Request all S-TIEs	If the local node is a ToF node, northbound flooding rules apply If the local node is not a ToF node, southbound flooding rules apply
TIRE (acknowledgement)	Acknowledge all received TIEs	Acknowledge all received TIEs	Acknowledge all received TIEs

To further understand the mechanisms that keeps the LSDB current, we'll introduce some of the important fields in the TIE header, in more detail:

- TIE ID: A unique number identifying the TIE.
- Sequence Number: A sequence number describing the current version of the TIE.
- Remaining Lifetime: A number describing the remaining lifetime of the TIE.

As each node generates TIEs, it should maintain a list of the corresponding TIE IDs (including empty TIEs), even when the protocol restarts. This mechanism improves convergence times. Consider a TIE that loses all content, the flooding of the empty TIE will allow adjacent nodes to accelerate the cleanup of stale entries. This occurs because TIE lifetimes are kept quite long to prevent periodic re-origination of TIEs, especially in larger fabrics. This increased lifetime value means that as convergence events happen, stale entries may exist for quite some time until the lifetime value expires at 0.

In other IGPs, it is typical to explicitly signal to other nodes that specific routing information be removed from the LSDB. OSPF flooding the LSA with MaxAge at the maximum value or IS-IS flooding the LSP with the lifetime set to 0 are both examples of this. These designs have proven fragile in deployment and are challenging to implement, so RIFT takes a different approach in that it floods an empty TIE with a *shorter* lifetime value, allowing the LSDB to naturally age it out.

Failures such as a node rebooting and rejoining the fabric make it necessary for RIFT to handle received TIEs with its own System ID. In such a case it's the local nodes responsibility to originate a more current empty TIE with a higher sequence number value in order to update the fabric. Now, sequence numbers in RIFT are a bit different from the other IGPs as well. Typical IGPs will perform checksums to verify that routing information is unique as well as leveraging the sequence numbers to supersede older routing updates. The checksum calculation presents an inefficiency due to the fact that it is CPU intensive and also requires rewriting the packet to insert the calculated value prior to being placed on the wire. In RIFT, *both* functions are handled by the sequence number alone. Like OSPF and IS-IS, the sequence number will continue to be used to supersede older advertisements as it increases, but the uniqueness of the TIEs is handled by a simple random 64-bit number (0 - 1,073,741,823) that is generated as new TIEs are originated. The large number makes collisions even more unlikely than a checksum calculation and makes rewriting the packet checksum unnecessary.

2.2.3 SPF Computation

Like other areas of RIFT, SPF computation is also based upon direction (N-SPF and S-SPF). This provides the distinction that allows RIFT to behave as a link-state protocol northbound where nodes at progressively *higher levels* have a full link-state view of the topology south of it and where nodes at *lower levels* have a default toward the north. Neither computation can generate looped paths, which enables RIFT to support optional features like bandwidth-based unequal cost load balancing.

2.2.3.1 North SPF

When calculating a local node's N-TIEs, N-SPF will only use northbound and east-west adjacencies. System ID and level values from the higher level S-TIEs are used to ensure proper bidirectional connectivity.

East-West links require additional consideration for default routes and more specific routes:

- Default routes may only be considered for N-SPF computation if the local node has no northbound adjacencies *and* the adjacent node has at least one. These two requirements ensure that loops are prevented over default routes and provide redundancy for nodes that lose all northbound adjacencies, except in the case of ToF nodes.
- More specific routes may be considered for N-SPF computation if no northbound neighbors are advertising the same or less specific non-default prefix *and* the local node is also not originating a less specific non-default prefix.

Furthermore, East-West links at the ToF also follow these rules, but are strictly used for N-TIE control plane flooding (Figure 2.2) between different planes and should never be used for forwarding. This facilitates route disaggregation in case of link failures in multiplane designs.

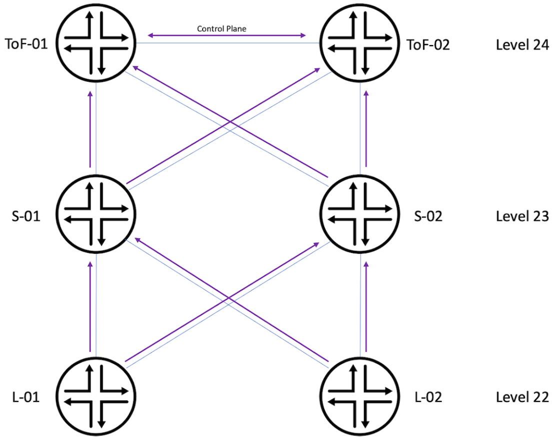


Figure 2.2 *North Flooding Scope*

2.2.3.2 South SPF

S-SPF (Figure 2.3) computes S-TIEs with only southbound adjacencies, while using N-TIEs from the lower level node's S-TIE to facilitate similar bidirectional checks to that of N-SPF. East-West links are never considered under any circumstance so that packets moving southbound never change direction.

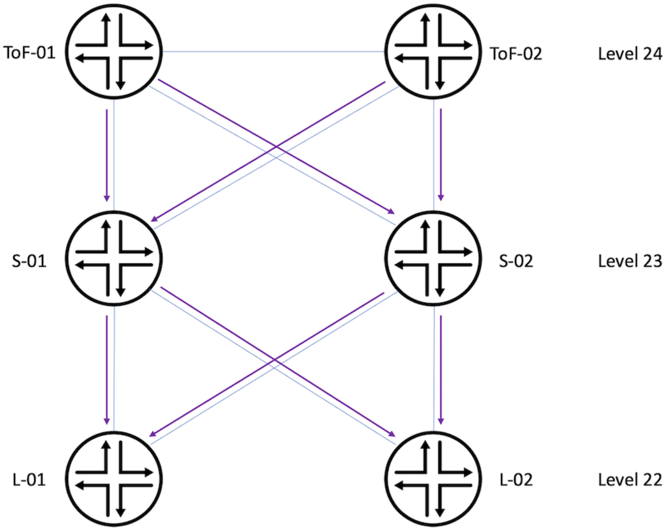


Figure 2.3 *South Flooding Scope*

2.2.3.3 Load Balancing

Load balancing in IP fabrics is quite a prevalent problem. It is either difficult to implement (i.e. via BGP) or doesn't utilize all of the available paths (OSPF and IS-IS) and is generally limited to equal cost paths only. RIFT however, also calculates available bandwidth that will continue to utilize all available shortest paths automatically. Traffic does not need to bow-tie within the fabric to reach its destination. Generally, load balancing is done only for the default routes heading north but could also be implemented for disaggregated prefixes and southbound routes.

Figure 2.4 depicts an IP fabric experiencing multiple failures. Each leaf node 20G of uplink capacity to each spine and each spine node has 200G of uplink capacity to its northbound node.

Under normal conditions, each prefix carries an associated distance value, this can simply be thought of as a typical metric value (lower values are preferred). As failures occur, SPF computation must factor in the now unavailable bandwidth and calculate a Bandwidth Adjusted Distance, or BAD. The BAD value will be used instead of the original distance to weight traffic across available links. The RIFT specification provides an example algorithm but given RIFT is loop-free, each node is free to implement a different algorithm if so desired.

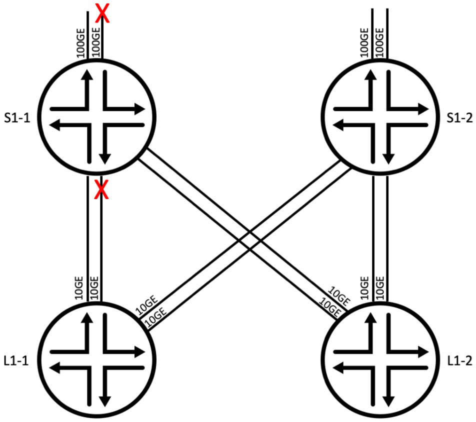


Figure 2.4 Weighted Unequal Cost Load Balancing

Table 2.8 lists the resulting BAD values calculated for default routes toward the spine layer.

Table 2.8 BAD Values

Node	Transit Path	BAD
L1-1	S1-1	2
L1-1	S1-2	1
L1-2	S1-1	2
L1-2	S1-2	1

2.2.4 South Reflection

South reflection is a mechanism where only Node S-TIEs are reflected back up one level north, allowing all nodes within the same level to be aware of each other.

Figure 2.5 shows a basic example of how south reflection is triggered and propagated.

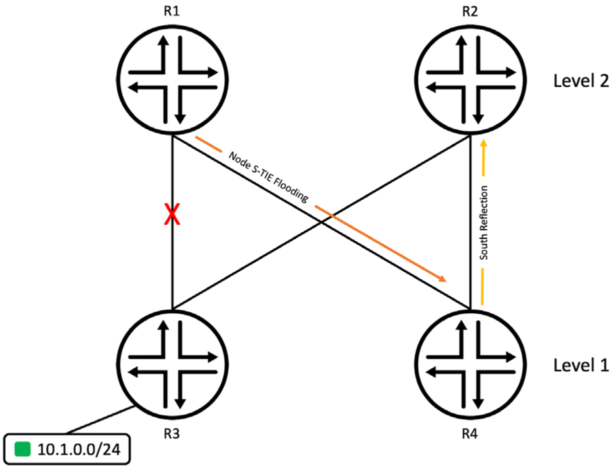


Figure 2.5 South Reflection

The adjacency between R1 and R3 fails, triggering a new Node S-TIE to be advertised to R4.

R4 will then reflect that Node S-TIE to R2, thereby informing R2 of the lost adjacency. Which also means that the prefix 10.1.0.0/24 has lost a degree of connectivity through R1.

2.2.5 Route Disaggregation

Route disaggregation is a procedure where RIFT advertises a more specific route in addition to the default route in order to mitigate blackholes.

Thus far, we have only mentioned a default route in the traditional sense (i.e. 0.0.0.0/0), but RIFT is capable of advertising different prefixes that could be used as a fabric-specific default. This factor is important because disaggregated prefixes must always be more specific than whatever is used as the fabric default. For example, if we were advertising 10.0.0.0/16 as our default fabric route, 10.50.1.210/32 could be disaggregated, but 192.168.1.59/32 could not.

There are two types of disaggregation, positive and negative. A node advertises positive routes to signal that it can reach a prefix and negative routes when it cannot. In either case, disaggregated routes are always advertised as prefix or external S-TIEs and are never reflected, other nodes don’t need to be aware of which nodes are advertising disaggregated routes.

2.2.5.1 Positive Disaggregation

Positive disaggregation is simple in that it is just an additional route advertisement that the southern node can route toward based on typical longest match routing. Effectively, RIFT punches a hole in the default route for prefixes that are *partially* connected.

It is also non-transitive in nature as prefixes are only advertised as far south as absolutely required to not burden nodes with routing information that would not be beneficial. For non-disaggregated prefixes, the default route still provides the necessary reachability.

Let's expand on the previous topology and show how south reflection triggers positive route disaggregation on R2 and how R3 and R4 will interpret the advertisement.

With R2 being aware (Figure 2.6) that R1 cannot reach 10.1.0.0/24 it will originate positively disaggregated S-TIEs to R3 and R4 for 10.1.0.0/24 (green) and the default route (purple).

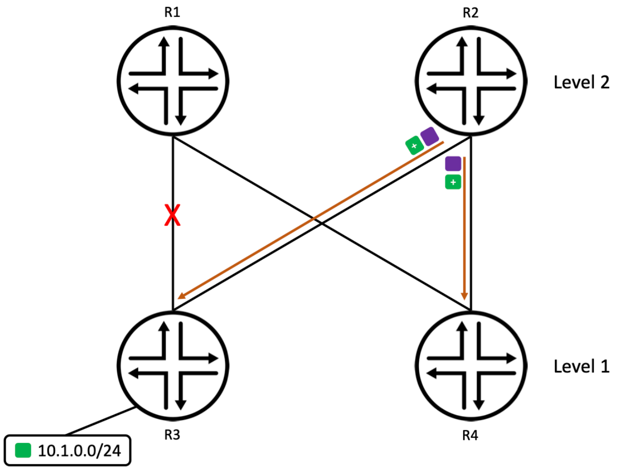


Figure 2.6 *Positive Disaggregation (Control Plane)*

This causes R3 and R4 to install the more specific prefix (Figure 2.7) and forward traffic toward R2, allowing R2 to load balance return traffic appropriately.

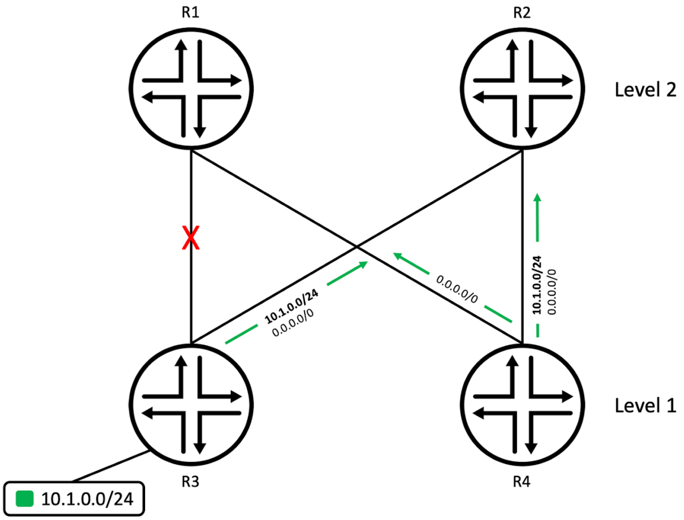


Figure 2.7 *Positive Disaggregation (Forwarding Plane)*

This also means that for prefixes that are not disaggregated, R4 may also load balance over the default routes toward R1 and R2.

Consider not having route disaggregation and R4 still tries to use the default route toward R1; any traffic toward destinations in 10.1.0.0/24 would be blackholed, or if a higher level is available, possibly bow-tie.

2.2.5.2 Negative Disaggregation

Negative disaggregation is more complex and we'll cover it in its entirety in upcoming sections, but for now let's just provide an overview.

Negative disaggregation is *required* when the fabric contains multiple planes. Similarly, ToF nodes are also required in multiplane fabrics to ensure flooding rules work as expected; without them negative disaggregation may not function in all cases. As a reminder, multiplane fabrics are where the ToF nodes cannot or do not connect to all nodes one level lower. This could be by design, due to scale limitations, or through a cascading series of failures.

Negative disaggregation is triggered when a node loses reachability to a prefix through all nodes one level higher that are part of a plane. That is to say, the prefix is reachable through at least one but not all planes. With the ToF nodes not connecting to every single node south of it and therefore not having the same routing information, they have no way of knowing that negative disaggregation is required when that loss of reachability occurs, or was never available in first place. Thus, it is mandatory to propagate that northbound information to the other plane's ToF nodes. One possibility is to interconnect them through a series of East-West rings. These are only required for the exchange N-TIEs so that all ToF nodes have the same LSDB, and not for forwarding, so two revenue ports per ToF should be sufficient. Figure 2.8 provides a simple example of this connectivity.

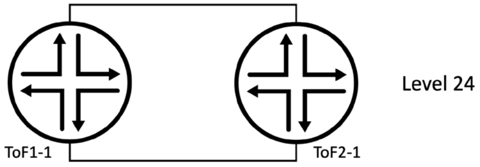


Figure 2.8 *Top-of-Fabric Interplane Ring*

Unlike positive routes, negative routes are considered transitive in nature. That means that negative routes may continue to propagate south until the blackhole is mitigated.

Because negatively disaggregated routes communicate a node's *inability* to reach a prefix, the RIB and FIB operations are quite different from that of the simpler positive route. Nevertheless, after necessary computations, the routing table only contains the usual "positive" entries. Let's look at an example of negative disaggregation. First, we'll focus on how nodes interpret those routes starting with Figure 2.9.

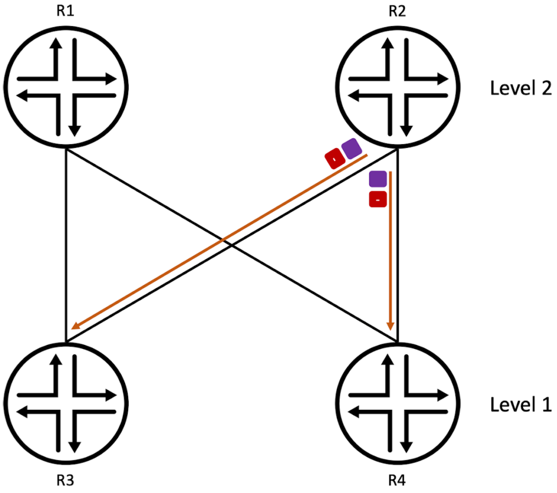


Figure 2.9 *Negative Disaggregation (Control Plane)*

Though it is not shown, you can discern that R2 has lost all connectivity for prefix 10.1.0.0/24 (red) in its respective plane and is advertising negative routes informing R3 and R4 that it is unable to reach it. Note that the default route (purple) is still advertised in the normal manner.

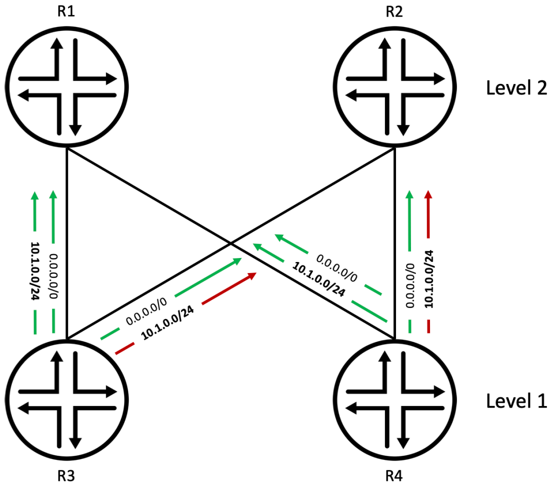


Figure 2.10 Negative Disaggregation (Forwarding Plane)

R3 and R4 receive the negative prefixes and install routes for R2 with a R1 as a next-hop in the RIB.

R3 and R4 will then perform additional route lookups to determine paths where negative routes *were not* received, and install corresponding next-hops for those paths into the FIB. This results in 10.1.0.0/24 only being reachable via R1.

Again, if you consider this example without disaggregation, traffic from either R3 or R4 that sends traffic destined to 10.1.0.0/24 via R2's default route would ultimately be blackholed.

2.2.6 Zero Touch Provisioning

By design any RIFT nodes in a fabric can operate in Zero Touch Provisioning (ZTP) mode, in other words it can be connected to the fabric with no initial configuration. ToF nodes are an exception, they must have a defined level value (usually 24) in order to serve as the "seed" for the rest of the fabric, allowing nodes to fully configure themselves. Configured nodes and nodes operating in ZTP mode can be mixed and will form a valid topology if achievable.

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

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

Before looking at an example, let's review a couple of important terms:

- Valid Offered Level (VOL): The level value advertised in a LIE that has passed all adjacency formation checks (other than level constraints).
- Highest Available Level (HAL): The highest level value among all received VOLs by a node.

Figure 2.11 shows a simple example of how S1-1 and L1-1 might use ZTP to derive their level values.

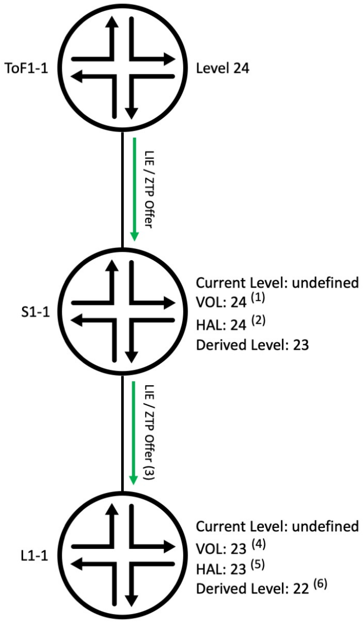


Figure 2.11 ZTP Level Derivation

Note that in Figure 2.11, ToF1-1 already has a defined level value of 24. Value 24 has been chosen for no particularly specific reason other than to allow formation of fabrics high enough for any practical purpose while not leading to excessively long "counting down to undefined" when a fabric loses all ToFs. The process is:

1. S1-1 receives the LIE from ToF1-1 and will see level 24, this value is considered a VOL.

2. S1-1 then calculates a HAL value of 24.

3. S1-1 derives its own level as 23 (HAL - 1) and will begin advertising that value in its LIEs.

4. L1-1 receives LIEs from S1-1 with a level value of 23, this value is considered a VOL.

5. L1-1 then calculates a HAL value of 23.

6. L1-1 derives its own level as 22.

You already know that it is required for ToF nodes to have a defined level value. It is also possible to do this for leaf nodes as well with the `LEAF_ONLY` flag, which effectively sets the level value to 0 and ensures that they remain at the bottom of the fabric. Nodes will not consider received LIEs with a level value of 0 in ZTP computation.

2.2.7 Failure Scenarios

2.2.7.1 Baseline

As discussed so far, RIFT has many advantages over other traditional link-state or distance vector protocols. Figure 2.12 shows a common single plane fat-tree topology. We'll use this to highlight some of RIFT's core concepts but also how those concepts are advantageous by exploring various failure scenarios step-by-step. We'll also talk about some design considerations as well.

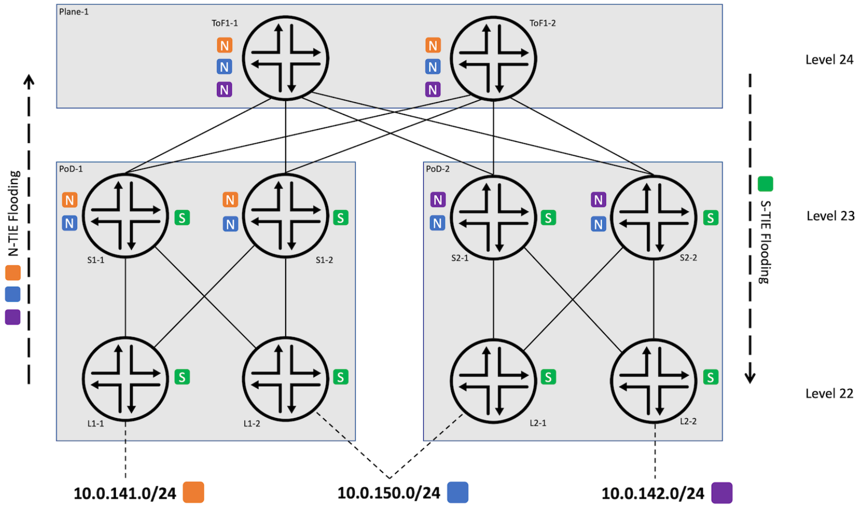


Figure 2.12 *Single Plane Fat-tree Topology*

2.2.7.2 Leaf Link Failure

Figure 2.13 illustrates the scenario of a link failure between S2-2 and L2-2, severing access to 10.0.142.0/24 via S2-2.

S

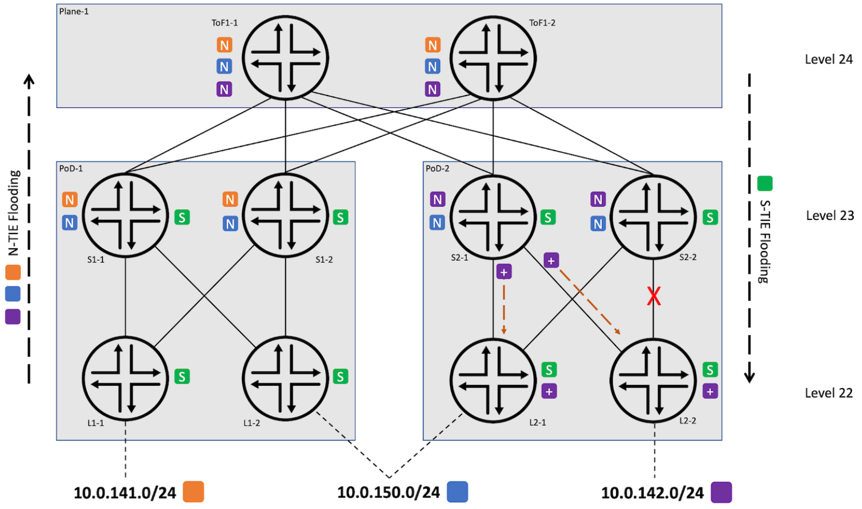


Figure 2.13 *Single Plane Leaf Link Failure*

You can see in Figure 2.13 that:

1. S2-2 and L2-2 will be aware of the failure and originate new Node TIEs indicating the lost adjacency:

- L2-2 advertises Node N-TIEs to S2-1
- S2-2 advertises Node N-TIEs to ToF1-1 and ToF1-2
- S2-2 also advertises Node S-TIEs to L2-1

At this point, S2-1 does not have the information required to discern that S2-2 has lost its adjacency to L2-2 because they are not directly connected.

2. L2-1 will then reflect the S-TIEs received from S2-2 to S2-1.

3. S2-1 now knows about the failure on S2-2 and will generate positively disaggregated prefix S-TIEs for prefix 10.0.142.0/24 to L2-1 and L2-2. S2-1 will still advertise its default route for any other prefixes that do not require disaggregation.

4. L2-1 and L2-2 will then prefer the more specific route for prefix 10.0.142.0/24 with equal cost paths to S2-1.

5. Finally, the tables below show each leaf's Northbound RIB before and after the failure.

It is possible that before the positively disaggregated routes are installed in L2-1's FIB, traffic from L2-1 toward 10.0.142.0/24 may traverse S2-2 via the default route. This would cause traffic to be routed to the ToF1-1 and ToF1-2 before returning to S2-1 for final delivery at L2-2. Even though this is suboptimal and may increase latency, it is only temporary and is preferred to blackholing the traffic.

Table 2.9 Results of North RIB after a Leaf Link Failure

Pre-Failure: L2-1 Northbound		Post-Failure: L2-1 Northbound	
0.0.0.0/0	via S2-1	10.0.142.0/24	via S2-1
	via S2-2	0.0.0.0/0	via S2-1
			via S2-2

Pre-Failure: L2-2 Northbound		Post-Failure: L2-2 Northbound	
0.0.0.0/0	via S2-1	10.0.142.0/24	Direct
	via S2-2	0.0.0.0/0	via S2-1

As you can see, RIFT contains the failure to the smallest possible footprint in the fabric, in this case the devices in PoD-2. There is no need for propagation to any other nodes as they would not benefit from any additional routing information.

Conversely, without enhancement, both link-state and distance vector protocols would require the failure to propagate to the rest of the fabric in some way.

2.2.7.3 Node Failure

Figure 2.14 illustrates a total spine failure on S1-1.

You can see in Figure 2.14 that:

1. All adjacencies associated with S1-1 are torn down:

- S1-1 - L1-1
- S1-1 - L1-2
- S1-1 - ToF1-1
- S1-1 - ToF1-2

2. This results in new Node TIEs being generated by the previously adjacent nodes:

- ToF1-1 and ToF1-2 advertise Node S-TIEs to S1-2, S2-1, and S2-2
- L1-1 and L1-2 advertise Node N-TIEs to S1-2.

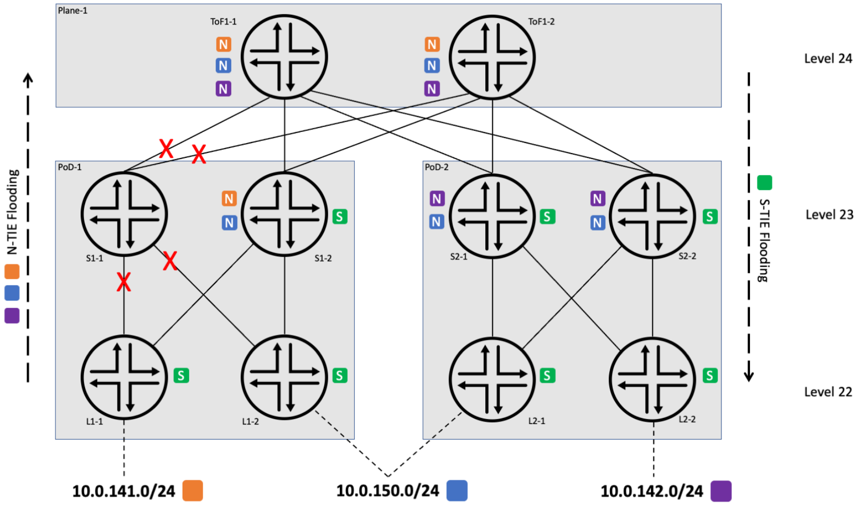


Figure 2.14 Node Failure

3. South reflection still occurs but does not yield any need for disaggregation because no blackhole would exist. This is because of the fact that all nodes affected by the failure are aware due to being directly connected. Remember, disaggregation is only required when a prefix is *partially* reachable.

4. The end result is simply a reduction in equal cost paths for the affected nodes. Table 2.10 below show the state of L1-1's northbound RIB and ToF1-1's southbound RIB before and after the failure (L1-2 and ToF1-2 would be identical to their counterparts, so they are not shown.)

Table 2.10 Results of North and South RIBs after a Node Failure

Pre-Failure: ToF1-1 Southbound		Post Failure: ToF1-1 Southbound	
10.0.141.0/24	via S1-1	10.0.141.0/24	via S1-2
	via S1-2		
10.0.142.0/24	via S2-1	10.0.142.0/24	via S2-1
	via S2-2		via S2-2
10.0.150.0/24	via S1-1	10.0.150.0/24	via S1-2
	via S1-2		via S2-1
	via S2-1		via S2-2
	via S2-2		

Pre-Failure: L1-1 Northbound		Post-Failure: L1-1 Northbound	
0.0.0.0/0	via S1-1	0.0.0.0/0	via S1-2
	via S1-2		

While this example yields a slightly larger blast radius than that of the previous one, you continue to see that the failure is contained to the minimum number of devices. Furthermore, because no disaggregation is required, flooding is also kept to a minimum.

2.2.7.3.1 ZTP considerations

The previous example effectively showed how the fabric would respond if S1-1 were to completely fail (for example lose power). It also assumed that level values were explicitly set in order to better illustrate the impact on the fabric.

Implementations that aim to take full advantage of RIFT's full ZTP functionality should also be explored.

Consider a variation on the failure where instead of S1-1 going totally offline, it simply loses its northbound adjacencies to ToF1-1 and ToF1-2, as shown in Figure 2.15.

Be sure to notice a couple of important changes in Figure 2.15:

- ToF nodes continue to have their level value defined.
- Spine nodes now have their level values undefined (as to allow ZTP to automatically derive it).
- Leaf nodes now have their level value set to 0.

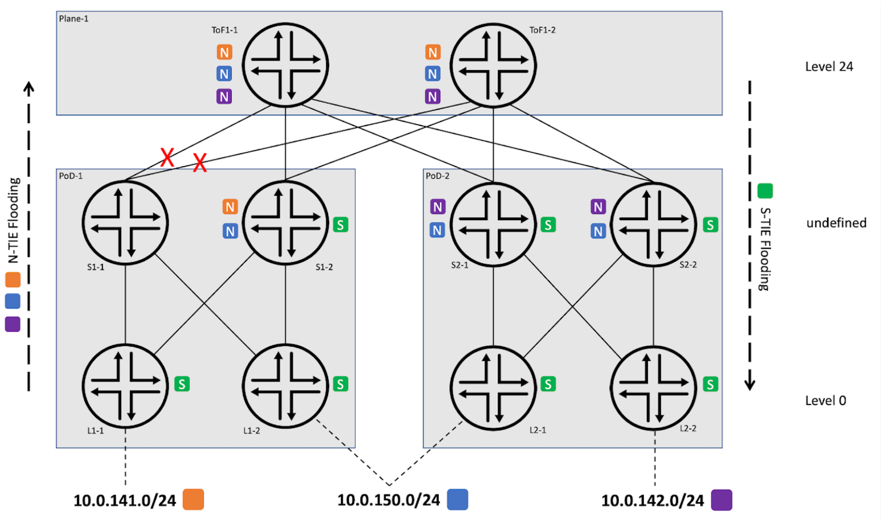


Figure 2.15 *Failure Results of Total Northbound Adjacency Loss in a ZTP Deployment*

Now, onto the failure:

1. Interfaces between S1-1 and the ToF nodes fail, causing these associated adjacencies to also fail:

- S1-1 - ToF1-1
- S1-1 - ToF1-2

2. S1-1 was previously able to automatically derive its level value from ToF1-1 and ToF1-2 due to the fact that they were at a higher level. With the adjacencies down, it will no longer receive ZTP offers from those nodes:

- S1-1 will ultimately attempt to recompute a new level value, but will first wait a short period of time.

3. L1-1 and L1-2 will advertise ZTP offers to S1-1 with a level value of 0 (due to having the LEAF_ONLY flag set).

4. S1-1 will ignore the offers from L1-1 and L1-2 because offers from leaves are considered invalid.

5. With no adjacencies to ToF1-1 and ToF1-2 (and therefore no ZTP offers) and having ignored the ZTP offers from L1-1 and L1-2, this means that S1-1 cannot successfully derive a level value and must tear down the remaining adjacencies:

- S1-1 - L1-1
- S1-1 - L1-2

6. With all four adjacencies down, the final result of route convergence will be the same as in the previous example.

2.2.7.4 Partitioned Fabric

Figure 2.16 illustrates one of the more catastrophic failure scenarios where multiple links fail resulting in a partitioned fabric.

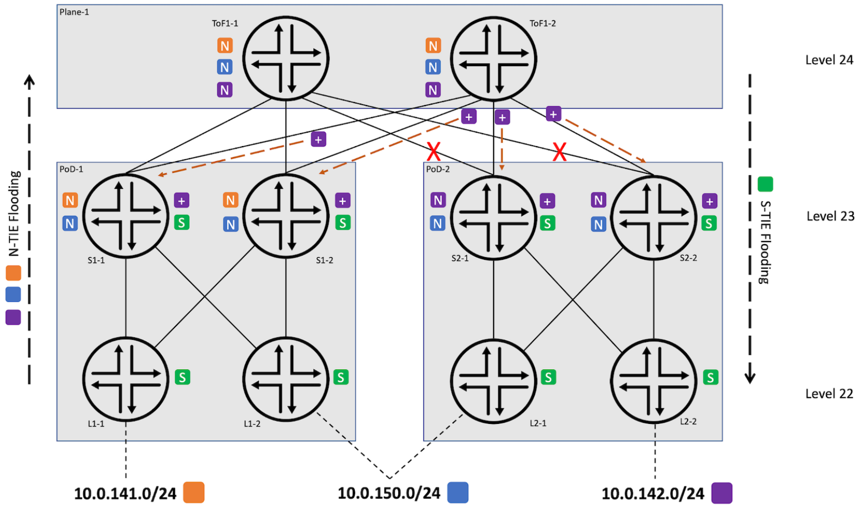


Figure 2.16 Partitioned Fabric

This failure means that ToF1-1 no longer has any reachability to 10.0.142.0/24.

1. Adjacencies between S2-1 - ToF1-1 and S2-2 - ToF1-1 are torn down.

2. As a result, new TIEs are generated by S2-1, S2-2, and ToF1-1:

- S2-1 and S2-2 advertise Node N-TIEs to ToF1-2
- S2-1 and S2-2 advertise Node S-TIEs to L2-1 and L2-2
- ToF1-1 advertises Node S-TIEs to S1-1 and S1-2

3. S1-1 and S1-2 reflect ToF1-1's S-TIEs up to ToF1-2, signaling that positive disaggregation is required.

4. ToF1-2 begins advertising positively disaggregated routes for prefix 10.0.142.0/24 in addition to the default prefix to all spines. This enables S1-1 and S1-2 to utilize the more specific route toward prefix 10.0.142.0/24. Without positive disaggregation, 50% of the traffic toward this prefix would be blackholed because ToF1-1 cannot reach it.

Again you can see that the scope of the failure is contained to only the affected level. You can also highlight the non-transitive nature of positive disaggregation. That is to say, positive routes are only advertised as far south as required (one level), rather than burdening the entire fabric. In this example, advertising the positive routes beyond the spine nodes would be of no benefit as the leaf nodes still maintain all available paths via the default route.

Anytime positive disaggregation is required at the ToF level, you should consider that you may see an influx in all traffic for the given prefix as the first ToF node announces the positively disaggregated route. This is transient behavior, but if an operator deems it to be problematic, the overload bit could be used to ensure convergence is complete prior to forwarding traffic in order to prevent such a scenario. In case many prefixes are affected, a proper implementation technique can assure that each of the ToF nodes advertising disaggregated prefixes, do so in a varying order to more evenly distribute traffic and impact.

2.2.7.5 Fallen Leaf

So far we've looked at examples where only positive disaggregation is required, so let's explore negative disaggregation as well. Remember, negative disaggregation is only required in multiplane topologies, which is typically by design, but *can* occur through an unlikely series of many cascading failures in a single plane topology. Figure 2.17 shows a multiplane fabric.

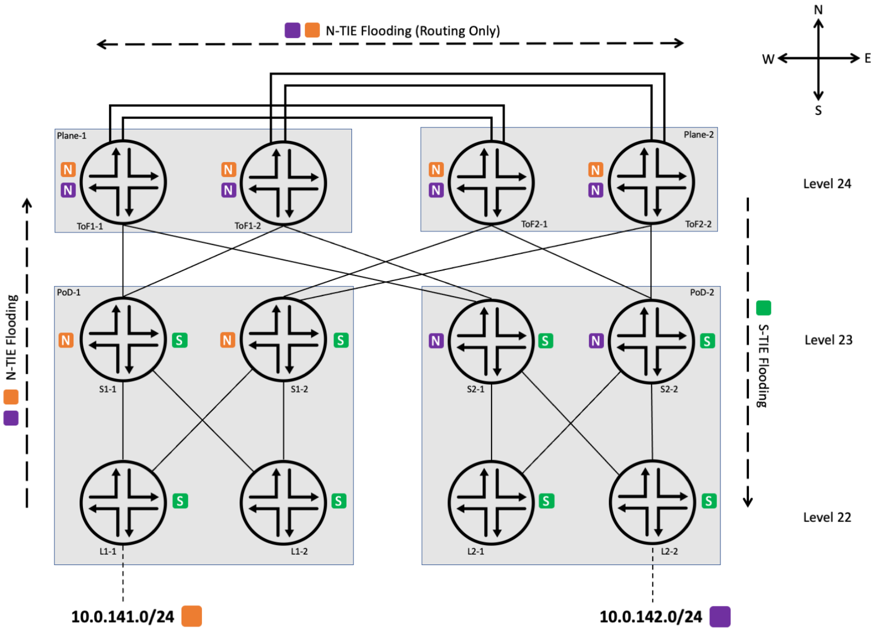


Figure 2.17 Multiplane Fabric

Be sure to make note of the new rings interconnecting ToF nodes between the different planes – their use will become apparent.

Finally, it may be helpful to clarify which nodes are members of which plane:

- Plane-1: ToF1-1, ToF1-2, S1-1, S2-1
- Plane-2: ToF2-1, ToF2-2, S1-2, S2-2
- Leaf nodes are not bound to a particular plane or rather "join" both planes at the bottom of the fabric.

Okay let's start with a single link failure (Figure 2.18) and then add a second (Figure 2.19) to show another extreme failure scenario, a fallen leaf.

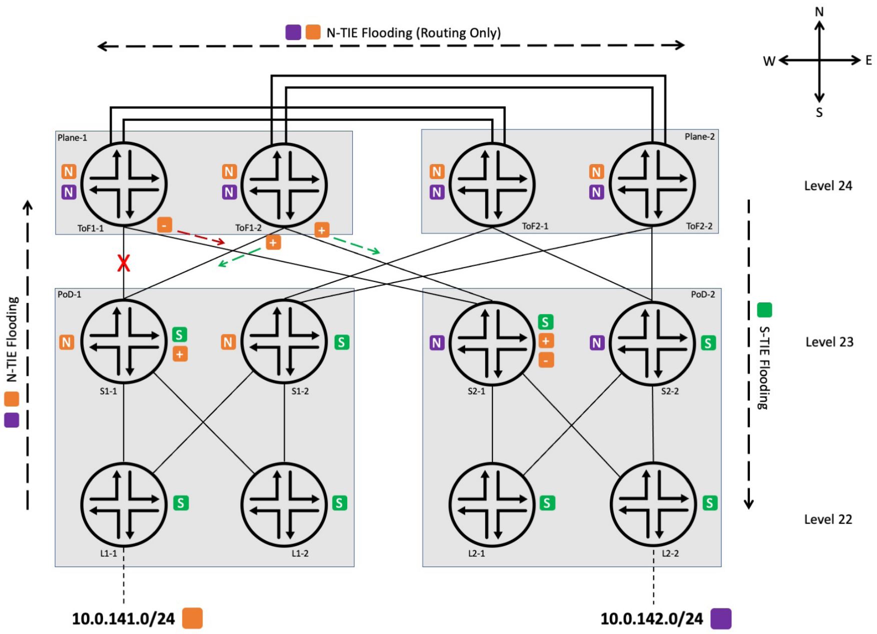


Figure 2.18 *Fallen Leaf (Partial)*

You can see that a fallen leaf is the situation where a leaf loses connectivity to all, or a portion of the ToF nodes, within one plane.

1. A link failure occurs between ToF1-1 and S1-1 in PoD-1, resulting in the adjacency being torn down.

2. New TIEs are generated by both ToF1-1 and S1-1, but let's focus on ToF1-1.

- ToF1-1 advertises Node S-TIEs to S2-1 that do not contain the southbound adjacency with S1-1, which are then reflected back to ToF1-2 as they are both part of Plane-1 (ToF nodes will not south reflect TIEs between different planes.) This causes ToF1-2 to see the missing southbound adjacency and advertise positively disaggregated S-TIEs to both S1-1 and S2-1.
- S1-1 advertises Node N-TIEs to ToF1-2 that do not contain the northbound adjacency with ToF1-1.

3. It's important to remember that leaf nodes are not part of a particular plane. This means that prefix 10.0.141.0/24 is learned by S1-2 and S2-2 and in turn ToF2-1 and ToF2-2.

4. Here is where the interplane rings come into play. Just a reminder that East-West links at ToF will only be used for routing but not forwarding. As just mentioned, ToF2-1 and ToF2-2 can also reach prefix 10.0.141.0/24; both will flood N-TIEs to ToF1-1 and ToF1-2 in Plane-1.

ToF1-1 and ToF1-2 receive those N-TIEs and see that 10.0.141.0/24 is reachable in Plane-2 causing ToF1-1 to advertise negatively disaggregated S-TIEs to S2-1.

5. At this point S2-1 has both positive and negative S-TIEs for 10.0.141.0/24. Let's observe two factors shown in the following figure, Figure 2.19:

- S2-1 will install the positive route because positive routes are preferred to negative routes.
- S2-1 will not re-originate the negative prefixes southbound to L2-1 and L2-2. However, in order for that to happen, all parent nodes (i.e. ToF1-1 and ToF1-2) would have to advertise negatively disaggregated prefixes first.

6. A link failure occurs between ToF1-2 and S1-1 in PoD-1, resulting in the adjacency being torn down. Now 10.0.141.0/24 is completely severed from Plane-1.

7. The previously positively disaggregated prefix route is withdrawn from ToF1-2 since it cannot calculate any paths to reach L1-1 and L1-2.

8. ToF1-2 calculates S-SPF using the N-TIEs flooded over East-West ToF links allowing it to see that 10.0.141.0/24 is no longer reachable in Plane-1. In response, a negatively disaggregated S-TIE is advertised toward S2-1 by ToF1-1 and ToF1-2.

9. S2-1 is now receiving negatively disaggregated S-TIEs from both parent nodes (ToF1-1 and ToF1-2) and re-originates its own negatively disaggregated S-TIEs to L2-1 and L2-2.

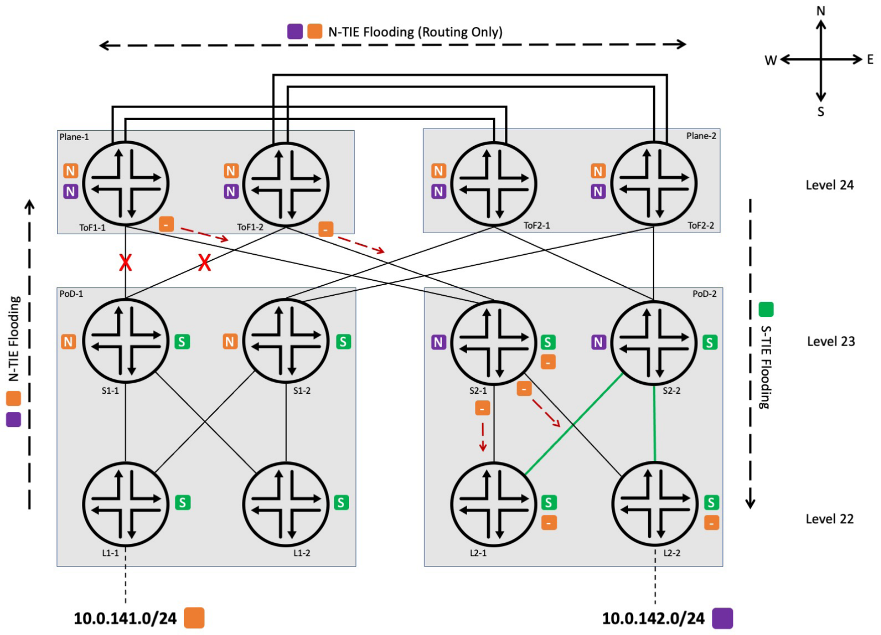


Figure 2.19 Fallen Leaf (Total)

10. L2-1 and L2-2 receive the negatively disaggregated S-TIEs and install routes towards S2-1 for 10.0.141.0/24 that contain only S2-2 as the next-hop. The default route toward S2-1 and S2-2 is still present. L2-1 and L2-2 will *not* reflect the negatively disaggregated S-TIEs simply due to the fact that TIEs containing prefixes are never reflected.

11. L2-1 and L2-2 *did not* receive negatively disaggregated advertisements from S2-2, so additional route recursion means that the FIB on both L2-1 and L2-2 will install 10.0.141.0/24 with a S2-2 as the next-hop. The default route toward both S2-1 and S2-2 will remain.

12. Reachability is now restored for 10.0.141.0/24 through Plane-2 while other traffic can still utilize the appropriate default route.

RIFT's ability to contain the failure scope still presents itself here by restricting impact only to the affected plane. However, this scenario is definitely more complex than previous examples, so let's review what we've learned about RIFT and multiplane fabrics.

In a multiplane fabric, certain failures cannot be solved by positive disaggregation alone. This means that negative disaggregation is mandatory and for it to function correctly it is necessary that all ToF nodes across different planes share the same northbound link-state database. This can be accomplished with a variety of methods. Using our example of a fat tree topology, interconnecting rings at the ToF level are recommended. If the fabric is a Clos network, simply adding more north/south connectivity will reduce the likelihood of this type of scenario, allowing positive disaggregation to be used.

We observed the *transitive* nature of negative disaggregation in that in order to maintain optimal routing, negative routes must sometimes be propagated down to the required subset of leaf nodes. It is also worth noting that like positive disaggregation, a node may also receive an influx of traffic, but in this case, the *last* node to advertise a negatively disaggregated prefix will receive the influx for that prefix.

Finally, we saw that negative disaggregation has the potential to be *inefficient* in regard to protocol implementation because of the additional route recursion that is needed to satisfy the requirements of negative disaggregation. However, this is unlikely, due to the fact that IP fabrics do not employ deep route resolution recursion which is CPU intensive and/or requires complex silicon.

Chapter 3

Juniper Implementation and Deployment

This chapter is illustrated with commands taken from the rift-in-action blueprint from Juniper Cloud Labs (https://jlabs.juniper.net/) with the topology displayed in Figure 3.1.

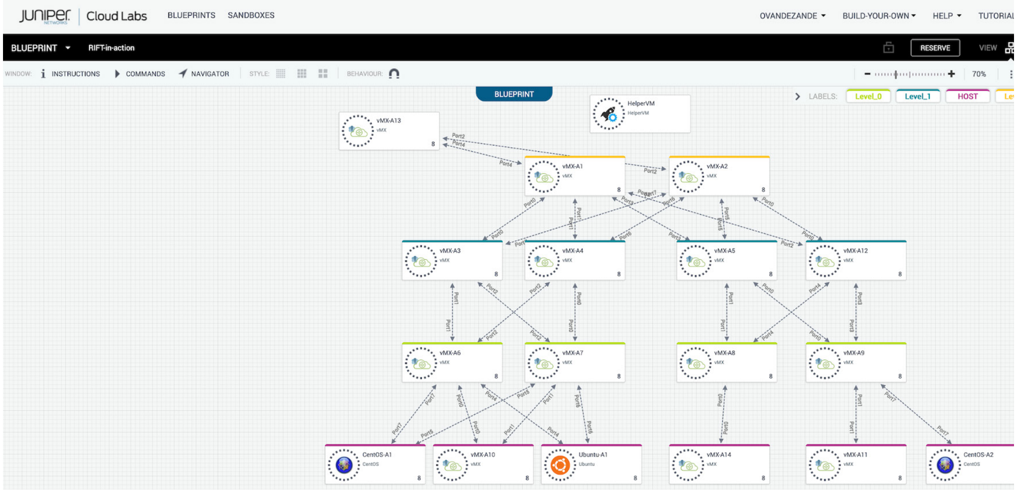


Figure 3.1 RIFT Deployment Lab in JCL

This chapter first covers the different Juniper RIFT components, followed by the installation and configuration.

3.1 RIFT Components

The Juniper RIFT implementation uses dedicated daemons which are installed as a standalone package and offer flexibility in the delivery of its new features, which is a must for the newer generation network fabrics (for example data centers) that require a level of agility.

In other words, RIFT implementation is decoupled from the Junos OS release cycle to allow for a fast delivery cycle.

Being decoupled from Junos also makes RIFT more independent, limiting the risk of destabilizing other components of the system. It also enables the ability to write RIFT in a different programming language than most of the Junos OS.

The Juniper RIFT daemon is written in the Rust language (https://www.rust-lang.org/) and in a highly multi-threaded fashion. Rust, unlike all other system programming languages like C/C++, is memory and thread safe. It is a very popular next generation system programming language due to its radically novel concepts allowing it to deliver and maintain much higher quality software than today's mainstream languages. In case you are interested in reading further on how Rust differs from other programming languages a nice blog can be found here: https://thenewstack.io/safer-future-rust/.

As said earlier, RIFT comes as a standalone Junos package. The package contains the necessary integration to interact with the Junos Routing Protocol Daemon (RPD) to program the routes, receive the status updates (for example interfaces state), and more. This section describes the different Junos components interacting with RIFT and the relations between them.

RIFT is based on two major daemons (riftd and rift-proxyd) shown in Figure 3.2.

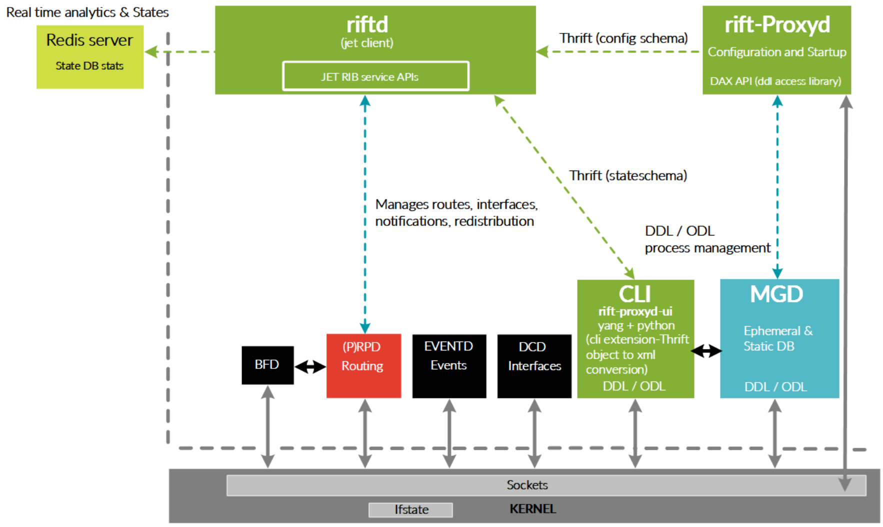


Figure 3.2 Simplified RIFT Basic Components Interactions

The white boxes in Figure 3.2 represent some of the standard Junos daemons. Among these, the RPD and the MGD have the most interactions with RIFT.

Let's look at the major RIFT elements.

3.1.1 RIFT Daemon (riftd)

The RIFT daemon (riftd) is written in Rust language which is, as said above, inherently memory and thread-safe. It is also highly threaded for efficiency and fully utilizes modern CPU architectures. In short, it's the heart and the brain of RIFT. It communicates with the programmable Routing Protocol Daemon (pRPD) through the Juniper Extension Toolkit (JET) RIB APIs. It is responsible for running RIFT (ZTP, adjacency establishment, neighbor authentication, database management, route exchange, paths calculation, route management and programming via the RPD, etc.).

MORE? Note that pRPD and JET are out of scope for this book but much more detailed information is available at https://www.juniper.net/documentation/en_US/junos/topics/concept/juniper-extenion-toolkit-overview.html.

Note that riftd also contains a Thrift server running in independent threads, associated with op-state (Thrift) schema to serve RIFT operational commands coming from the cli process but also to serve runtime configuration requests of the daemon coming from the mgd via rift-proxyd.

RIFT protocol packets are also exchanged using serialized Thrift models. Details on the protocol model are to be found in the IETF RIFT specification. At the time of this writing, found here: https://tools.ietf.org/html/draft-ietf-rift-rift-12#appendix-B (always make sure you are reading the latest draft or standard).

3.1.2 JET Services Daemon (jsd)

Juniper Extension Toolkit (JET), an evolution of the Junos SDK, provides a modern, programmatic interface (API) for developers of Juniper and third-party applications on Junos devices. It focuses on providing a standards-based interface to the Juniper Networks Junos operating system (Junos) for customizing management and control plane functionality. JET provides the common RPC framework to enable inter-daemon communication. Top daemons (rpd, dfwd, etc.) provide (or will provide) APIs based on the JET framework. Each API has an IDL-based definition. The JET framework exposes APIs internally directly to the Junos developed applications or externally via the jsd (JET Service Daemon) accessed via gRPC. In the RIFT case, riftd communicates directly via programmable RPD.

MORE? For more information on the JET framework: https://www.juniper.net/documentation/product/en_US/juniper-extension-toolkit.

3.1.3 Programmable Routing Protocol Daemon (pRPD)

The Programmable Routing Protocol Daemon (pRDP) is the Junos control plane based on a set of APIs allowing the 'user' to directly communicate with the routing protocol daemon, for example to collect/delete/modify existing routes or add new ones. As said, pRPD is a standard component of the Junos Extension Toolkit (JET). It enables the Juniper routing control plane to be programmed by application such as external routing controllers or internal Juniper applications.

The pRPD JET RIB Service APIs provide dynamic programmability for general RIB routes – similar to configured static routes. Currently, in pRPD, all routes added from the RIB APIs are static routes though the adding protocol and preferences are preserved (i.e. route client data), and for example, policies can refer to such routes by using the native protocol name.

3.1.4 Management Daemon (mgd)

The management daemon is part of the user interface (UI) infrastructure and is responsible for the command line interface (CLI), Junos scripts (Junos XML RPC API) and the Junos software installation. It was designed to be open to extensions such as other configuration and commands, but also infrastructure for daemon process management, API for daemon access to router configuration, etc. The `mgd` has a data definition language (DDL) and an associated library containing schemas describing the configuration statements and the operational commands. This DDL infrastructure is generic and open to extensions. DAX is the DDL access library APIs. The `mgd` also has an Output Definition Language (ODL) that defines the XML hierarchy of tags for CLI output. It produces a set of tools and shared libraries for use when outputting data (XML, plain text, etc.).

3.1.5 rift-proxyd-ui

The rift-proxyd-ui is the YANG package with the RIFT user interface extensions added to `mgd`. It comes with Python conversion scripts for converting the YANG based RIFT data models and including them into the Junos schemas. Newly added RPCs and configuration hierarchies are immediately available for use. In order to support collecting RIFT operational state, YANG data models also define the ODL format of the XML data presented to the user and it comes with associated Python action scripts. These Python action scripts contain a Thrift client code used to retrieve the operational states from riftd. The Thrift objects received by the CLI are then converted into XML for user presentation.

3.1.6 rift-proxyd

This daemon manages the RIFT configuration changes requested by mgd, translates the configuration into Thrift schemas, and updates the riftd Thrift server thread. The rift-proxyd is also in charge of managing riftd daemon (start, restart, etc.). The rift-proxyd is written in C/C++ to benefit from the junos tools (and libraries like DAX) written in C/C++ (for example DAX).

3.1.7 Redis Server

Riftd can interact with the open source database server, Redis (https://redis.io/) to store all the RIFT domain internal data such as TIEs and statistics. More information on Redis is in the Appendix.

3.2. Installation

Let's review the RIFT package installation for Junos devices.

The testing environment used to illustrate the command of this section is based on the following equipment and Junos version:

```
jcluser@vMX-A6> show version | match ":|kernel"
Hostname: vMX-A6
Model: vmx
Junos: 19.4R1.10
JUNOS OS Kernel 64-bit  [20191115.14c2ad5_builder_stable_11]

jcluser@vMX-A6>
```

Please note that at the time of writing this, Juniper's RIFT implementation was still in its beta phase and specifics may vary or change over time. The authors encourage readers to make suggestions so updates can be done to the book. To reach the authors please email dayone@juniper.net.

3.2.1 Prerequisites

At the time this book was published (make sure to refer to the release notes for your version prerequisites) the following prerequisites must be fulfilled when deploying RIFT:

- RIFT will only install on a 64-bit x86 version of Junos (32-bit Junos version is not supported)
- The minimum Junos version must be 19.4R1 or newer
- An upgrade to the newer version of RIFT for Junos devices requires you to first remove the non-default RIFT configuration prior to the new RIFT package activation.

As per the release notes:

> _The RIFT JUNOS package will install over any (V)MX or QFX product running 19.4R1 or newer *64-bit* JUNOS release._
>
> _The node should contain a default configuration only and if a previous RIFT package has been installed, the previous configuration and package should be removed. An installation upgrading a rift package over an already installed one may succeed but is not guaranteed. Also, default values installed with a package may change so it is recommended to install a package on a node without any previous RIFT content._
>
> _The switches MUST be cabled in a Clos with a single plane as top-of-fabric, i.e. every spine MUST be connected to all superspines._

3.2.2 Download the RIFT base package

At the time of this writing, RIFT isn't available yet via https://support.juniper.net/support/downloads/ hence the RIFT installation package should be obtained via https://www.juniper.net/us/en/dm/free-rift-trial/ or your Juniper account team. The latter potentially has access to a more recent version of the code.

3.2.3 Install the RIFT Base Package

The RIFT package comes as a TAR Archive file that has been compressed using Gnu Zip (gzip) software. Once the RIFT base package has been downloaded, it must be uncompressed as it includes the installation image file and the documentation.

3.2.3.1 Step 1: Extract the RIFT installation file

Use your favorite archive utility to extract the RIFT package contents.

In this next example, use the Terminal app on a Mac running OSX, to enter the `tar -xzvf rift-1.2.0.junos.bundle.tgz` command:

```
rift_rocks@mbp packages % tar -xzvf rift-1.2.0.junos.bundle.tgz
x content/
x content/junos-rift-x86-64-19.4I20200322_0610_prz.tgz.sha1.dirdep
x content/CONFIGURE.md
x content/CHANGELOG.md
x content/LICENSE.md
x content/junos-rift-x86-64-19.4I20200322_0610_prz.tgz.sha1
x content/junos-rift-x86-64-19.4I20200322_0610_prz.tgz.dirdep
x content/README.md
x content/rli-37959-documentation.pdf
x content/extra-scripts/
x content/extra-scripts/decode-some-opstate.py
x content/LIMITATIONS.md
x content/COPYRIGHT.md
x content/FAQ.md
```

```
x content/junos-rift-x86-64-19.4I20200322_0610_prz.tgz
x content/VERSION
x content/INSTALL.md
```

By default, when using the TAR command, the files are extracted from the package into a new folder named `content`. The file names extracted and having the extension `.md` are general information files in text format using the Markdown language.

For configuration documentation, refer to the CONFIGURE.md and rli-37959-documentation.pdf files.

3.2.3.2 Step 2: Install the RIFT package

Copy the extracted RIFT installation file (junos-rift-x86-64-19.4I20200322_0610_prz.tgz) to the /var/tmp directory on the Junos device where RIFT will be installed:

```
jcluser@vMX-A6> file list detail /var/tmp/junos-rift*
-rwxr-xr-x  1 root  wheel   31976919 May 15 08:03 /var/tmp/junos-rift-x86-64-19.4I20200322_0610_prz.
tgz*
total files: 1

jcluser@vMX-A6>
```

Then install the RIFT package using the: `request system software add <software-package>` command:

```
jcluser@vMX-A6> request system software add <software-package>
NOTICE: Validating configuration against junos-rift-x86-64-19.4I20200322_0610_prz.tgz.
NOTICE: Use the 'no-validate' option to skip this if desired.
Verified junos-rift-x86-64-19.4I20200322_0610_
prz signed by PackageDevelopmentECP256_2020 method ECDSA256+SHA256
Adding junos-rift-x86-64-19.4I20200322_0610_prz ...
Initializing...
Mounting os-libs-11-x86-64-20200411.2b552dd_builder_stable_11
Mounting os-runtime-x86-64-20200411.2b552dd_builder_stable_11
Mounting os-zoneinfo-20200411.2b552dd_builder_stable_11
Mounting junos-net-prd-x86-64-20200415.051749_builder_junos_194_r1
Mounting junos-libs-x86-64-20200415.051749_builder_junos_194_r1
Mounting os-libs-compat32-11-x86-64-20200411.2b552dd_builder_stable_11
Mounting os-compat32-x86-64-20200411.2b552dd_builder_stable_11
Mounting junos-libs-compat32-x86-64-20200415.051749_builder_junos_194_r1
Mounting junos-runtime-x86-32-20200415.051749_builder_junos_194_r1
Mounting jsim-pfe-x86-32-20200415.051749_builder_junos_194_r1
Mounting sflow-mx-x86-32-20200415.051749_builder_junos_194_r1
Mounting py-extensions2-x86-32-20200415.051749_builder_junos_194_r1
Mounting py-extensions-x86-32-20200415.051749_builder_junos_194_r1
Mounting py-base2-x86-32-20200415.051749_builder_junos_194_r1
Mounting py-base-x86-32-20200415.051749_builder_junos_194_r1
Mounting os-vmguest-x86-64-20200411.2b552dd_builder_stable_11
Mounting os-support-x86-64-20200411.2b552dd_builder_stable_11
Mounting os-crypto-x86-64-20200411.2b552dd_builder_stable_11
Mounting na-telemetry-x86-32-19.4R1.10
Mounting junos-secintel-x86-32-20200415.051749_builder_junos_194_r1
Mounting junos-libs-compat32-mx-x86-64-20200415.051749_builder_junos_194_r1
Mounting junos-runtime-mx-x86-32-20200415.051749_builder_junos_194_r1
Mounting junos-rpd-telemetry-application-x86-64-19.4R1.10
Mounting junos-rift-x86-64-19.4I20200322_0610_prz
Mounting junos-redis-x86-32-20200415.051749_builder_junos_194_r1
Mounting junos-platform-x86-32-20200415.051749_builder_junos_194_r1
Mounting junos-openconfig-x86-32-19.4R1.10
Mounting junos-modules-x86-64-20200415.051749_builder_junos_194_r1
Mounting junos-modules-mx-x86-64-20200415.051749_builder_junos_194_r1
Mounting junos-libs-mx-x86-64-20200415.051749_builder_junos_194_r1
Mounting junos-jsqlsync-x86-32-20200415.051749_builder_junos_194_r1
Mounting junos-dp-crypto-support-mtx-x86-32-20200415.051749_builder_junos_194_r1
Mounting junos-daemons-x86-64-20200415.051749_builder_junos_194_r1
Mounting junos-daemons-mx-x86-64-20200415.051749_builder_junos_194_r1
Mounting junos-appidd-mx-x86-32-20200415.051749_builder_junos_194_r1
Mounting jsim-wrlinux-x86-32-20200415.051749_builder_junos_194_r1
Mounting jsim-pfe-vmx-x86-32-20200415.051749_builder_junos_194_r1
Mounting jsim-pfe-internal-x86-32-20200415.051749_builder_junos_194_r1
Mounting jsdn-x86-32-19.4R1.10
Mounting jsd-x86-32-19.4R1.10-jet-1
Mounting jpfe-wrlinux9-x86-32-20200415.051749_builder_junos_194_r1
Mounting jpfe-wrlinux-x86-32-20200415.051749_builder_junos_194_r1
Mounting jpfe-spc3-mx-x86-32-19.4R1.10
Mounting jpfe-X960-x86-32-20200415.051749_builder_junos_194_r1
Mounting jpfe-common-x86-32-20200415.051749_builder_junos_194_r1
Mounting jpfe-aft-x86-32-20200415.051749_builder_junos_194_r1
Mounting jpfe-X-x86-32-20200415.051749_builder_junos_194_r1
Mounting jmrt-base-x86-64-x86-64-20200415.051749_builder_junos_194_r1
Mounting jinsight-x86-32-19.4R1.10
Mounting jfirmware-x86-32-19.4R1.10
Mounting jdocs-x86-32-20200415.051749_builder_junos_194_r1
Hardware Database regeneration succeeded
Validating against /config/juniper.conf.gz
mgd: commit complete
Validation succeeded
Mounting junos-rift-x86-64-19.4I20200322_0610_prz
Rebuilding schema and Activating configuration...
mgd: commit complete
Restarting MGD ...

WARNING: cli has been replaced by an updated version:
CLI release 20200407.122723_builder.r1099298 built by builder on 2020-04-07 12:44:45 UTC
Restart cli using the new version ? [yes,no] (yes)

Restarting cli ...
jcluser@vMX-A6>
```

The installation file is verified, the components are extracted, copied in their final location, and installed. The new RIFT specific CLI commands are also enabled, which requires a restart of mgd and the CLI. Answer *yes* or press Enter when prompted.

At this stage, RIFT is installed and can be validated using the `show rift version info` command:

```
jcluser@vMX-A6> show rift versions info
Package: 1.2.0.1093653
Built On: 2020-03-22T05:57:21.227211423+00:00
Built In: PVT_194_RIFT_11
Encoding Version: 4.0
Statistics Version: 3.0
Services Version: 18.0
```

3.2.3.3 Step 3: Activate the RIFT package

Once the RIFT package is installed, it needs to be activated. RIFT package activation is done using the `request rift package activate` command.

NOTE The RIFT package activation can only happen if RIFT is not configured in the device, or the RIFT configuration is *default*. A non-default RIFT configuration will cause the activation process to fail. If the RIFT configuration is not the default, the activation will not proceed. Please, review the prerequisites section if need be.

```
jcluser@vMX-A6> request rift package activate
RIFT activation information logged in /var/log/rift-activate.log
Command name is activate
Opening device for junos-rift activation
Entering device configuration mode for device
Locking configuration
Loading junos-rift package default configuration
Platform is MX240(virtual)
Loading junos-rift platform default configuration
Committing configuration
Unlocking configuration
Closing rpc connection
Open: Ok
Config: Ok
Config lock: Ok
Config load package defaults: /etc/config/junos-rift/package-defaults.conf
Config load platform defaults: /etc/config/junos-rift/vmx/platform-defaults.conf
Config commit: Ok
Config unlock: Ok
Close: Ok
junos-rift activation completed successfully!

jcluser@vMX-A6>
```

The top of fabric nodes must have RIFT activated using the `activate-as-top-of-fabric` knob:

```
jcluser@vMX-A6> request rift package ?
Possible completions:
  activate          Install package and platform default configuration and enable RIFT
  activate-as-top-of-
fabric  Install package and platform default configuration and enable RIFT as top-of-fabric
jcluser@vMX-A6>
```

RIFT is now active and a basic configuration is loaded. See the next chapter for a default configuration that is loaded and activated while executing the activation command.

3.3 Configuration

3.3.1 Default Configuration

After the activation of the RIFT package, the following default configuration is automatically enabled. Activation also causes the generic and platform specific RIFT configuration parameters to be loaded into the existing configuration. These RIFT default configuration files are located in the /etc/config/junos-rift/ folder:

```
jcluser@vMX-A6> file list detail /etc/config/junos-rift/

/etc/config/junos-rift/:
total blocks: 16
drwxr-xr-x  2 root  wheel     512 May 15 08:03 mx/
lrwxr-xr-x  1 root  wheel      68 May 15 08:03 package-defaults.conf@ -> /packages/mnt/junos-rift/
etc/config/junos-rift/package-defaults.conf
drwxr-xr-x  2 root  wheel     512 May 15 08:03 ptx/
drwxr-xr-x  2 root  wheel     512 May 15 08:03 qfx/
drwxr-xr-x  2 root  wheel     512 May 15 08:03 vmx/
total files: 1

jcluser@vMX-A6>
```

The `/etc/config/junos-rift/package-defaults.conf` contains the RIFT defaults for all the platforms, and `/etc/config/junos-rift/<platform>/platform-defaults.conf` contains the default RIFT configuration specific to the <platform> platform.

As RIFT was designed as a dedicated app out of the Junos rpd, the Junos default `ddos-protection` configuration needs to be adapted for RIFT to operate. This default `ddos-protection` configuration change is also managed by the RIFT activation process. As an example, here is the associated `ddos-protection` change for a MX platform:

```
jcluser@vMX-A6> file show /etc/config/junos-rift/mx/platform-defaults.conf
system {
    ddos-protection {
        protocols {
            unclassified {
                aggregate {
                    disable-routing-engine;
                    disable-fpc;
                }
                control-v4 {
                    disable-routing-engine;
                    disable-fpc;
                    no-flow-logging;
                    bypass-aggregate;
                }
                control-v6 {
                    disable-routing-engine;
                    disable-fpc;
                    no-flow-logging;
                    bypass-aggregate;
                }
                host-route-v4 {
                    disable-routing-engine;
                    disable-fpc;
                    no-flow-logging;
                    bypass-aggregate;
                }
                host-route-v6 {
                    disable-routing-engine;
                    disable-fpc;
                    no-flow-logging;
                    bypass-aggregate;
                }
            }
            l3nhop {
                aggregate {
                    bandwidth 100000;
                    burst 100000;
                    disable-fpc;
                }
            }
        }
    }
}
```

As mentioned in previous sections, RIFT is fully automated and supports ZTP. Devices acting as ToF nodes must be configured as such. For easy installation as ToF, during the activation process, specify the `activate-as-top-of-fabric` flag which will enable the necessary ToF configuration:

When using the `activate-as-top-of-fabric` knob, it will add the following configuration:

```
protocols {
    rift {
        level top-of-fabric;
    }
}
```

This knob must be used for the ToF nodes only. All other RIFT nodes (super-spine, spine, leaf, etc.) must use the `activate` knob only.

As the ToF is key in the ZTP process (RIFT level derivation, etc.), it is recommended to install RIFT at least on one ToF node first. Installing it later will work as well but meanwhile, the deployed nodes will not form any adjacencies and complain regularly about missing ToF.

By default, RIFT is enabled on the most likely interfaces for the platform. If the interfaces of interest are not listed into the default `rift-interfaces` interface range, you have to add (or replace) them manually into the configuration. This can be easily done using a dedicated interface range group.

The interfaces without a configured description and for which RIFT is enabled get a default description as follows.

```
jcluser@vMX-A6> show interfaces descriptions | match up
ge-0/0/0     up up   Match interfaces that RIFT could use.
ge-0/0/1     up up   Match interfaces that RIFT could use.
ge-0/0/2     up up   Match interfaces that RIFT could use.
ge-0/0/3     up down Match interfaces that RIFT could use.
ge-0/0/4     up up   Match interfaces that RIFT could use.
ge-0/0/5     up down Match interfaces that RIFT could use.
ge-0/0/6     up down Match interfaces that RIFT could use.
ge-0/0/7     up up   Match interfaces that RIFT could use.

jcluser@vMX-A6>
```

By default, RIFT auto-detects neighbors on these interfaces. Once a neighbor is discovered and passing all the adjacency checks, the neighborship is established. Then ZTP method is started.

Remember that by default RIFT is dual-stack and will try to discover neighbors for both address families. However, RIFT will refuse to send on an address family (AF) unless it has a listening address for it in order to prevent asymmetric adjacencies.

```
protocols {
    rift {
        apply-groups rift-defaults;
        interface rift-interfaces;
    }
}
```

The RIFT protocol is enabled and activated for the auto-selected interfaces for the platform. Also, the RIFT default parameters are applied to the configuration.

The default RIFT parameters instruct the node to auto-configure based on the ZTP process:

- The node-id is configured to be automatically generated
- Also the node level
- IPv4 and IPv6 multicast addresses used by RIFT and requested at IANA:
    - ff02::a1f7: ALL_V6_RIFT_ROUTERS
    - 224.0.0.120: ALL_V4_RIFT_ROUTERS

Here is the RIFT default configuration group:

```
groups {
    rift-defaults {
        protocols {
            rift {
                node-id auto;
                level auto;
                lie-receive-address {
                    family {
                        inet 224.0.0.120;
                        inet6 ff02::a1f7;
                    }
                }
                interface <*> {
                    lie-transmit-address {
                        family {
                            inet 224.0.0.120;
                            inet6 ff02::a1f7;
                        }
                    }
                    bfd-liveness-detection minimum-interval 1000;
                }
            }
        }
    }
}
```

Interfaces partaking in RIFT are grouped into an interface range (`rift-interfaces`):

```
interfaces {
    interface-range rift-interfaces {
        member ge-0/0/*;
        description "Match interfaces that RIFT could use.";
    }
}
```

Also, enabling Dynamic Next-Hop interface support may be necessary on older Junos releases:

```
routing-options {
    programmable-rpd {
        rib-service {
            dynamic-next-hop-interface {
                enable;
            }
        }
    }
}
```

This configuration is required and allows RIFT to program the RIB service via pRPD. The `dynamic-next-hop-interface` support makes the pRPD RIB service responsive to interface events.

3.3.2 Minimum Configuration

When deploying RIFT, only the root of the fabric nodes must be manually configured as top of the fabric using the configuration as following:

```
protocols {
    rift {
        level top-of-fabric;
    }
}
```

This configuration can be automatically inserted during the RIFT package activation when the node is activated using the optional `activate-as-top-of-fabric` knob (see section: 3.2.3.3 Step 3: Activate the RIFT package).

Also, if some interfaces that need to run RIFT but are not part of the default 'rift-interfaces' range, these interfaces must be manually added to the RIFT configuration as well. As an example, here manually adding some xe-2/0/0 and xe-0/2/1 interfaces to the RIFT protocol:

```
interfaces {
    interface-range grp-fabric_interfaces {
       member xe-0/2/0;
       member xe-0/2/1;
    }
}

protocols {
    rift {
       interface grp-fabric_interfaces;
    }
}
```

As RIFT relies on IP, the minimum is to have IPv6 enabled on the RIFT interfaces and an IPv6 link-local address allocated to it, and also IPv6 Neighbor Discovery protocol working for the interface.

If IPv4 needs to be supported, the IPv4 address family must be enabled on the interface as well. In this case, an IPv4 over IPv6 RIFT configuration.

NOTE With IPv4 underlay, there is currently a software issue (PR1526927) that prevents a Routing Engine (RE) running Junos from sending out IPv4 packets over IPv6 next hops. Therefore, as a temporary workaround, each RE interface participating in the RIFT domain must be manually configured with an IPv4 address in order to provide connectivity between the loopback IPv4 addresses. Transit traffic is currently not impacted by this issue.

IPv6 configuration can be omitted if it's a pure IPv4 link but then the interfaces need valid IPv4 addresses. It is very risky to build a fabric that has links with different combinations of address families since connectivity in an address family may not be guaranteed under such a scenario and a breakage will be very challenging to find operationally.

```
groups {
    rift_ge_ifd {
        interfaces {
            <ge-*> {
                unit 0 {
                    family inet;
                    family inet6;
                }
            }
        }
    }
}
apply-groups rift_ge_ifd;
```

This configuration enables the IPv4 and IPv6 address families for the Gigabit Ethernet interfaces.

3.3.3 Recommended Configuration

This section gives up tips and tricks the authors believe should be part of a proper configuration. They are not essential for the operation of RIFT itself but should be present for a more secure operation, for example.

3.3.3.1 General Configuration

A good practice is to enable a minimum of RIFT tracing options in order to be able to analyze issues as they happen. The `rift-proxyd` process is responsible for configuration and startup and `riftd` runs the protocol. To enable basic tracing for these daemons, configure the following:

```
protocols {
    rift {
        traceoptions {
            file riftd size 10m files 2;
            level info;
        }
        proxy-process {
            traceoptions {
                file rift-proxyd size 10m files 2;
                level info;
            }
        }
    }
}
```

Hosts are usually more exposed to security vulnerabilities. So, when routing on the host is configured (cRPD with RIFT, currently planned for Junos 21.2), to increase the security (authentication, message integrity and anti-replay) you can configure the interface protection (outer key) to secure RIFT with the host:

```
protocols {
    rift {
        authentication-key 12 {
            key "<key omitted>"; ## SECRET-DATA
        }
        interface ge-0/0/0.0 {
            lie-authentication strict;
            allowed-authentication-keys 12;  # (Example using key-id 12)
            lie-origination-key 12;
        }
    }
}
```

This output configures the `strict` authentication mode which means that the interface accepts authentication only if a key is present and valid.

With the configuration, the Link Information Element (LIE) is authenticated. The LIE is equivalent to the hellos in other IGP protocols and are used by RIFT to form the three-way adjacency on the interface.

Keys and authentication must be configured on both sides of the link.

You can also secure the Topology Information Element (TIE), which is similar to an LSP/LSA with a link-state routing protocol. This is done by configuring the inner keys:

```
protocols {
    rift {
        authentication-key 8453 {
            key "<key omitted>"; ## SECRET-DATA
        }
        tie-origination-key 8453;  # (Example using key-id 8453)
    }
}
```

Any node where TIE origin authentication is applied must have the necessary TIE keys configured. Even if TIEs are not origin authenticated on a node they are secured when transmitted over links that are LIE secured nevertheless

NOTE At the time of this writing, November 2020, RIFT authentication is available as preview functionality only, therefore it's not an officially released beta. Check release notes for its current status.

To obtain the most effective results from ZTP, it is recommended that LIE and/or TIE authentication not be configured on routing and switching nodes in the RIFT fabric until ZTP functions are complete.

3.3.3.2 Platform Specific Configuration

For Broadcom-based QFX switches, it is recommended to configure the flexible Unified Forwarding Table (UFT) to the forwarding profile `l3-profile` that increases the scale of the unicast IPv4 and IPv6 address tables:

```
chassis {
    forwarding-options {
        l3-profile;
    }
}
```

From RIFT v1.2 and beyond, this profile configuration is part of the QFX platform RIFT defaults configuration template and is automatically added to the configuration.

3.3.4 Optional Configuration Statements

If you want to disable RIFT on some default interfaces, you can replace the `rift-interfaces` range with your own interface range under protocols/rift, or you can overwrite the configured interfaces default by specifically disabling RIFT on the interface, an example for the interface ge-0/0/0.0 shown here:

```
protocols {
    rift {
        interface ge-0/0/0.0 {
            disable;
        }
    }
}
```

By default, RIFT uses the system host name as the RIFT system name and the chassis private base mac-address as the RIFT system ID (appended with an unique number in the presence of a routing-instance):

```
jcluser@vMX-A9> show rift node status
System Name: vMX-A9, System ID: 002c6bf55fe0c000
Level: 22, RIFT Encoding Major: 4, Minor: 0
Flags: overload=False
Capabilities: flood-reduction: True
LIE v4 RX: 224.0.0.120, LIE v6 RX: ff02::a1f7, LIE RX Port: 914
 Re-Connections: 0
Peers: 8, 3-way: 2, South: 0, North: 2

jcluser@vMX-A9> show chassis mac-addresses
MAC address information:
  Public base address   2c:6b:f5:5f:d9:00
  Public count          1984
  Private base address  2c:6b:f5:5f:e0:c0
  Private count         64

jcluser@vMX-A9>
```

However, you can override these default values by manually configuring the RIFT system `name` and `node-id` (therefore the RIFT system id). (It's important to remember that the RIFT system name and the RIFT system id must be unique across the RIFT domain.)

```
protocols {
    rift {
        node-id 19;
        name leaf09;
    }
}
```

The `node-id` can be configured in decimal (numeral 19 in the above example) but could also be configured using a hexadecimal number (e.g. 0x13). However, `show configuration` will always output it as a decimal. On the other hand, the RIFT system-id displayed with the operational `show` commands is converted into hexadecimal (hex(19) = 0x13):

```
jcluser@vMX-A9> show rift node status
System Name: leaf09, System ID: 0000000000000013
Level: 22, RIFT Encoding Major: 4, Minor: 0
Flags: overload=False
Capabilities: flood-reduction: True
LIE v4 RX: 224.0.0.120, LIE v6 RX: ff02::a1f7, LIE RX Port: 914
 Re-Connections: 0
Peers: 8, 3-way: 2, South: 0, North: 2
```

During the zero touch provisioning phase, a node automatically allocates its RIFT level based on its position in the fabric compared to the top of the fabric. It means that if a new leaf node is connected to an existing leaf, not being a spine, this existing leaf will become an aggregation node for that new leaf which will allocate a level '-1' compared to the leaf it is connected to.

However, if you do not want to allow a leaf node to become an aggregation node, but always stay a leaf, you can configure it manually with the lowest level '0' (LEAF_ONLY flag). It means that this leaf will never form a RIFT adjacency with another leaf connected south of it:

```
protocols {
    rift {
        level leaf;
    }
}
```

This could be, as an example, interesting to configure in a host running cRPD plus RIFT. In the same way, it could be an additional control to configure a switch (for example, ToR) as level '1', to allow only direct RIFT host nodes to connect to it, but not another extra layer behind.

RIFT is, by default dual-stack. To disable an address family (AF) from RIFT, for example for the IPv4 address family, the following command can be used:

```
deactivate groups rift-defaults protocols rift lie-receive-address family inet
```

This configuration command will remove any IPv4 adjacencies and will remove any IPv4 routing next-hops from RIFT as well.

Also, RIFT will refuse to send on an AF unless it has a listening address for it to prevent asymmetric adjacencies. It means that if no IPv4 receive address is configured, the adjacent node is forbidden from transmitting on IPv4 addresses since you would end up with adjacencies in one direction in one AF, which is not a desirable outcome.

There is also the option to advertise a RIFT ThreeWay interface subnet, by default, in nominal conditions, and only the default route is advertised southbound and nothing northbound. Use the configuration below so the RIFT ThreeWay interface subnets are advertised:

```
protocols {
    rift {
        interface ge-0/0/0.0 {
            mode advertise-subnets;
        }
    }
}
```

This command injects into RIFT the ThreeWay subnets for the interface ge-0/0/0.0 and those will show up as the RIFT internal route (for example, S (South direction) or N (North direction)) and not NExt (North External) as it would be if it was redistributed using a well-known export policy.

The following displays the route received from the top of the fabric view:

```
jcluser@vMX-A1> show rift routes content

Prefix                       Active Metric N-Hop All Present
-------------------------------+------+------+--------+-------------------
0.0.0.0/0                    Disc                Disc
::/0                         Disc                Disc
fc32:1:f:8500::/127          N        3 80000f35 N
```

The ThreeWay advertised subnet route from the leaf vMX-A9 is seen as ‘N’ (North meaning received from South and to be advertised Northbound).

Depending on the DC fabric configuration, another optional configuration that could be needed is to advertise a fabric specific default instead of the typical default route. This could happen, for example, if a default route is already used for the management of the DC fabric and the management is not isolated into a dedicated management VRF. In this case, the auto-generated RIFT default route is in conflict with the management one that could be configured, as an example, as a static route.

The way to change the RIFT auto generated default route by an aggregate route is shown here:

```
 protocols {
    rift {
        default-prefixes {
            family {
                inet 172.32.0.0/16;
            }
        }
    }
}
```

By using the `set protocols rift default-prefixes <value>` on each RIFT node (could be up to the level Leaf +1, as there is an automatic one level propagation), the default route will be replaced by the configured prefix.

As you can see next, the default prefix in RIFT is now an aggregate prefix:

```
jcluser@vMX-A3> show rift routes content

Prefix                       Active Metric N-Hop All Present
-------------------------------+------+------+--------+-------------------
172.32.0.0/16                S        2 8000d072 S
172.32.0.1/32                SExt     2 8000d071 SExt
172.32.0.2/32                SExt     2 8000d070 SExt
172.32.0.6/32                NExt     2 8000d075 NExt
172.32.0.7/32                NExt     2 8000d074 NExt
::/0                         S        2 8000d072 S
fc32:1:f::1/128              SExt     2 8000d071 SExt
fc32:1:f::2/128              SExt     2 8000d070 SExt
fc32:1:f::6/128              NExt     2 8000d075 NExt
fc32:1:f::7/128              NExt     2 8000d074 NExt

jcluser@vMX-A3>
```

3.3.4.1 Disabling the auto-generated default prefix

It could happen that you need to disable the auto-generated default prefix in the ToF nodes to use instead of the default prefix received by BGP and coming from the DC gateways. The focus is on IPv4 here but this it's exactly similar for IPv6. By default, the ToF generates a default prefix as you can see here:

```
jcluser@vMX-A2> show rift routes content

Prefix                       Active Metric N-Hop All Present
-------------------------------+------+------+--------+-------------------
0.0.0.0/0                    Disc                Disc
...
```

By using the `default-prefixes-advertisement never` RIFT configuration knob, there is the option to deactivate the auto-generated default prefix in the ToF nodes. Then, by configuring the export southbound of the default prefix received into the BGP session with the DC gateways, the ToF nodes and the RIFT nodes on the level ToF -1 start using this external default prefix instead.

Here's a configuration example in the ToF node to use the external default prefix instead of the ToF nodes' auto-generated one:

```
policy-options {
    policy-statement ps-rift_south {
        term DEFAULT4 {
            from {
                family inet;
                protocol bgp;
                route-filter 0.0.0.0/0 exact;
            }
            then accept;
        }
        term DEFAULT_DENY {
            then reject;
        }
    }
}

protocols {
    rift {
        default-prefixes-advertisement never;
        export {
            southbound {
                ps-rift_south;
            }
        }
    }
}
```

The auto-generated default prefix is not in the RIFT routes content, and the default one received via BGP is actually used for routing traffic out of the DC:

```
jcluser@vMX-A2> show rift routes content

Prefix                        Active Metric N-Hop All Present
-------------------------------+------+------+--------+-------------------
172.32.0.6/32                 NExt     3 80002235 NExt
172.32.0.7/32                 NExt     3 80002235 NExt
172.32.0.8/32                 NExt     3 8000223b NExt
172.32.0.9/32                 NExt     3 8000223b NExt
172.32.1.3/32                 NExt     2 80002234 NExt
172.32.1.12/32                NExt     2 8000223a NExt

jcluser@vMX-A2> show route table inet.0 match-prefix 0.0.0.0/0

inet.0: 24 destinations, 24 routes (23 active, 0 holddown, 1 hidden)
+ = Active Route, - = Last Active, * = Both

0.0.0.0/0          *[BGP/170] 01:34:50, localpref 100
                     AS path: 65888 I, validation-state: unverified
                   >  to 172.32.4.5 via ge-0/0/2.0

jcluser@vMX-A2>
```

Now this default prefix is advertised by the ToF as an external route:

```
jcluser@vMX-A2> show rift database content | match "Dir|---|002c6bf5c09ac000"
Dir Originator    Type   ID       SeqNr
Lifetime   Origin Creation Time   Origin   Content   Key ID
---+----------------+---------+--------+-----------------+--------+----------------------+-Lifetime-+-
-Size-+--------
S   002c6bf5c09ac000 Node      10000000  5f4d739d8cd4   602359 2020/08/31 22:04:15  604800  439     0
S   002c6bf5c09ac000 External  60000067  5f4d740d2c4e   602405 2020/08/31 22:05:01  604800  175     0
S   002c6bf5c09ac000 External  6000007f  5f4d739c5756   602405 2020/08/31 22:05:01  604800  175     0
N   002c6bf5c09ac000 Node      10000000  5f4d739d8f16   602359 2020/08/31 22:04:15  604800      None

jcluser@vMX-A2> show rift tie 002c6bf5c09ac000/S/external/60000067
TIE ID: 002c6bf5c09ac000/S/External/60000067

 Prefix                        Metric LPB ATT On Link
 -------------------------------+------+---+---+-------
  0.0.0.0/0                         1   yes

jcluser@vMX-A2>
```

And the node in the ToF level -1 install now uses this route as an external prefix:

```
jcluser@vMX-A3> show rift routes content

Prefix                        Active Metric N-Hop All Present
-------------------------------+------+------+--------+-------------------
0.0.0.0/0                     SExt     2 80005f92 SExt
...
```

With the default prefix advertisement method, the default prefix is seen as the RIFT internal ‘S’ (South) route and not external in the ToF level -1 (and lower) nodes as shown above when using the BGP method.

3.3.4.3 Injecting the loopback interfaces into RIFT

Be cautious when injecting or exporting routes into RIFT, specifically southbound. RIFT was designed with scale and simplicity in mind, using the auto-aggregation concept and certainly not flooding all routes at all the levels in the fabric. Adding and removing leaf nodes should have limited impact on the fat tree and little to any in other leaf nodes, but only the upper layers in the fabric proportionally to their level and, of course the ToF nodes, but they should have the hardware scale designed for it as well.

By default, the interface's loopback IP addresses are not transported by RIFT but they may be needed in certain scenarios. For example, if you want to build an overlay network on top of RIFT, then you may need to transport the loopback IP addresses into RIFT to build the overlay control plane (for example BGP).

The following should be considered when injecting routes into RIFT:

- Route injection into RIFT can be done in the south direction (southbound) or in the north direction (northbound). Each direction is independent and the same prefix from both directions undergoes an internal tie breaking process based on route preferences.
- Northbound routes:
  - Northbound routes injected into RIFT will be automatically propagated north to all levels, including the ToF, based on the rule that flooding always transitively progresses northbound.
  - A RIFT route received from a northbound neighbor will never be propagated southbound unless specifically configured using a southbound export policy together with the `allow-rift-routes` knob configuration under `[edit protocols rift export southbound]`.
- Southbound routes:
  - Southbound injected routes into RIFT will only be propagated one level to the south.
    - If a southbound injected route into RIFT needs to be propagated further south toward the leaf nodes, a southbound export policy together with the `allow-rift-routes` configuration is required on all the intermediate nodes between the route injector node and the leaf nodes.
  - A RIFT route received southbound will never be propagated further southbound to other links unless specifically configured using a southbound export policy together with the `allow-rift-routes`, and only as long as this route exists as a northbound route.

- So a ToF node receiving routes from a ToF -1 level node will never propagate back that route to the south to other links even if configured. This because that route does not exist as a northbound route in the ToF node.
- On the other hand, a spine node receiving a route from a leaf node south of it will propagate back that route to other leaf nodes south of it according to the southbound export policy associated with the `allow-rift-routes` configuration.

As evocated above, it could be needed, for optimal routing into the lower levels of the fat tree, that the IP address (loopback) of a service in the RIFT domain (for example BGP Route Reflector) be propagated south down the tree. In this case, the next steps are needed for that IP address to be propagated.

3.3.4.3.1 Step 1: for the service node

The service node is the node providing the service into the RIFT domain (for example BGP RR). It is good practice to position the BGP RRs on the ToF -1 level for a 5-stage Clos. In this case, you need to export the loopback IP addresses for the service nodes, southbound and northbound on the tree, and as an example the following northbound and southbound export policies can be configured. In fact, under normal conditions in a single plane fabric such a configuration is actually not even needed but in case of ToF links breaking the ToF may become unreachable by the leaves.

```
policy-options {
    policy-statement ps-rift_service {
        term LO0_SERVICE {
            from {
                family inet6;
                protocol direct;
                route-filter fc32:1:f:1::/64 prefix-length-range /128-/128;
            }
            then accept;
        }
        term LO0_SERVICE4 {
            from {
                family inet;
                protocol direct;
                route-filter 172.32.1.0/24 prefix-length-range /32-/32;
            }
            then accept;
        }
        term DEFAULT_DENY {
            then reject;
        }
    }
}
```

First, the above policy is configured matching on the protocol `Direct` and the prefix range associated to the loopback interface IP addresses. This injects the node loopback IP addresses into RIFT.

Then the export policy is used with a southbound and a northbound `export` command for the protocol RIFT:

```
protocols {
    rift {
        export {
            northbound {
                ps-rift_service;
            }
            southbound {
                ps-rift_service;
            }
        }
    }
}
```

This will redistribute to the north and to the south the node loopback IP address(es).

3.3.4.3.2 Step 2: for the intermediate nodes

If the exported loopbacks are required to reach the entire fabric and be visible all the way to the bottom of the tree, every level has to redistribute the obtained RIFT northbound route southbound again, as by default it is not (exporting southbound is only one level downwards by default).

This is ensured by the combination of a southbound policy matching on the loopback IP addresses range and the `allow-rift-routes` knob with the export command:

```
policy-options {
    policy-statement ps-rift_south {
        term LO0_SERVICE {
            from {
                family inet6;
                protocol rift;
                route-filter fc32:1:f:1::/64 prefix-length-range /128-/128;
            }
            then accept;
        }
        term LO0_SERVICE4 {
            from {
                family inet;
                protocol rift;
                route-filter 172.32.1.0/24 prefix-length-range /32-/32;
            }
            then accept;
        }
        term DEFAULT_DENY {
            then reject;
        }
    }
}
```

The above policy example matches on the routes coming from the RIFT protocol for the specified prefixes.

Let's look at the allow-rift-routes knob in the RIFT export statement:

```
protocols {
    rift {
        export {
            southbound {
                ps-rift_south;
                allow-rift-routes;
            }
        }
    }
}
```

The allow-rift-routes knob in the RIFT export statement is only configurable in the southbound direction and allows calculated RIFT northbound routes to be considered in the southbound redistribution. If you remember, it allows you to redistribute RIFT routes southbound as long as there are associated northbound routes.

The leaf nodes at the bottom of the tree will receive the northbound routes propagated southbound by the intermediate nodes without specific configuration.

If you want to also build an overlay network on top of the RIFT domain, you probably need to redistribute the loopback IP addresses for the ToF nodes and the leaf nodes.

In this case, you have to configure the following on these nodes.

3.3.4.3.3 Step 3: for the ToF nodes

The next policy displayed matches on protocol direct, the subnet of interest which is configured under the loopback interface:

```
policy-options {
    policy-statement ps-rift_south {
        term LO0_ONLY {
            from {
                family inet6;
                protocol direct;
                route-filter fc32:1:f::/64 prefix-length-range /128-/128;
            }
            then accept;
        }
        term LO0_ONLY4 {
            from {
                family inet;
                protocol direct;
                route-filter 172.32.0.0/24 prefix-length-range /32-/32;
            }
            then accept;
        }
        term DEFAULT_DENY {
            then reject;
        }
    }
}
```

Then the export is configured in the southbound direction with the policy selecting the routes of interest:

```
protocols {
        export {
            southbound {
                ps-rift_south;
            }
        }
    }
}
```

3.3.4.3.4 Step 4: for the leaf nodes

The leaf nodes at the bottom of the tree need to use a northbound export policy in order to advertise their loopback IP addresses up the tree to the ToF nodes. So, a northbound export policy needs to be configured to match on the loopback address to be exported:

```
policy-options {
    policy-statement ps-rift_north {
        term LO0_ONLY {
            from {
                family inet6;
                protocol direct;
                route-filter fc32:1:f::/64 prefix-length-range /128-/128;
            }
            then accept;
        }
        term LO0_ONLY4 {
            from {
                family inet;
                protocol direct;
                route-filter 172.32.0.0/24 prefix-length-range /32-/32;
            }
            then accept;
        }
        term DEFAULT_DENY {
            then reject;
        }
    }
}
```

And the routes matching the export policy are injected and exported in the northbound direction in the RIFT domain:

```
protocols {
    rift {
        export {
            northbound {
                ps-rift_north;
            }
        }
    }
}
```

NOTE By default, any intermediate node will automatically advertise the routes received southbound from the leaf nodes in the north direction up to the ToF nodes without any special configuration.

3.3.4.3.5 RIFT tags list

As an alternative from RIFT version 1.2, it is also possible to inject the IP addresses from an interface (IFL) using the RIFT interface passive mode. By default, a route injected with the RIFT interface passive mode is an internal RIFT route propagated northbound through the RIFT domain up to the ToF, but also *one level* southbound.

Beginning with RIFT version 1.3, it is also possible to assign a tag or list of tags (the current implementation is a maximum of two tags) to a RIFT interface (IFL) resulting in having the tags associated with the IP addresses of the interface where it is configured. Similar to a BGP community list, these tags are attached to the RIFT route and propagated with it.

Based on these tags, it is easy to configure generic filters in all the RIFT domain nodes to control RIFT route propagation.

As an example, using the configuration below to inject and propagate a service loopback interface address up to the ToF and also one level southbound. At the node injecting the routes into RIFT, the passive interface is configured with a tag, for example, `8453`:

```
protocols {
    rift {
        interface lo0.0 {
            mode passive;
            tags 8453;
        }
    }
}
```

As you can see in the node injecting the route, the tag is associated with the route of type internal:

```
jcluser@vMX-A3> show rift tie 002c6bf5bdc93200/s/prefix/2000006d
TIE ID: 002c6bf5bdc93200/S/Prefix__/2000006d

 Prefix                       Metric LPB ATT On Link
 -------------------------------+------+---+---+-------
  172.32.1.3/32                  1   yes

jcluser@vMX-A3> show rift python-tie 002c6bf5bdc93200/s/prefix/2000006d
TIE ID: 002c6bf5bdc93200/S/Prefix__/2000006d
Content: TIEElement(node=None, positive_disaggregation_prefixes=None, positive_external_
disaggregation_prefixes=None, negative_disaggregation_prefixes=None, keyvalues=None, prefixes=Prefix
TIEElement(prefixes={IPPrefixType(ipv6prefix=None, ipv4prefix=IPv4PrefixType(prefixl
en=32, address=-1407188733)): PrefixAttributes(from_
link=None, loopback=None, metric=1, tags=frozenset([8453]), monotonic_clock=None, directly_
attached=True)}), external_prefixes=None)

jcluser@vMX-A3>
```

As shown in the command output, per RIFT standard, the tags list is supported from the beginning and has an associated field in the RIFT TIE.

In the node north of it, the route is seen as RIFT internal with the tags propagated with it:

```
jcluser@vMX-A1> show rift routes content

Prefix                      Active Metric N-Hop  Tag   All Present
-------------------------------+------+------+--------+--------+-----------
172.32.0.0/16               Disc                          Disc
172.32.0.4/32               N       2 8000c1e3       N
172.32.0.5/32               N       2 8000c11b       N
172.32.0.6/32               N       3 8000c11e       N
172.32.0.7/32               N       3 8000c11e       N
172.32.0.8/32               N       3 8000c1ce       N
172.32.0.9/32               N       3 8000c1ce       N
172.32.1.3/32               N       2 8000c1e9 8453  N
172.32.1.12/32              N       2 8000c116 8453  N
::/0
Disc                        Disc

jcluser@vMX-A1> show route 172.32.1.3/32

inet.0: 25 destinations, 26 routes (24 active, 0 holddown, 1 hidden)
+ = Active Route, - = Last Active, * = Both

172.32.1.3/32  *[Static/20/100] 00:09:07, metric2 0, tag 8453
               >  to 172.32.129.2 via ge-0/0/0.0

jcluser@vMX-A1>
```

In the southbound node one level down, the route is also seen as an internal RIFT route with its tag:

```
jcluser@vMX-A6> show rift routes content

Prefix                       Active Metric N-Hop Tag   All Present
-------------------------------+------+------+--------+--------+-----------
172.32.0.0/16                S        2 8000a6b9       S
172.32.0.4/32                S        2 8000a6ba       S
172.32.1.3/32                S        2 8000a6b8 8453  S
::/0
                             S        2 8000a6b9       S

jcluser@vMX-A6> show route 172.32.1.3/32

inet.0: 12 destinations, 13 routes (12 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

172.32.1.3/32   *[Static/20/100] 00:06:33, metric2 0, tag 8453
                >  to 172.32.131.1 via ge-0/0/1.0

jcluser@vMX-A6>
```

Then, for each intermediate node, use a policy statement to match on the associated tag, independently of the prefix to which it is attached (or the protocol family):

```
policy-options {
    policy-statement ps-rift_south {
        term LO0_SERVICE {
            from {
                protocol rift;
                tag 8453;
              }
            then accept;
        }
        term DEFAULT-DENY {
            then reject;
        }
    }
}

protocols {
    rift {
        export {
            southbound {
                ps-rift_south;
                allow-rift-routes;
            }
        }
    }
}
```

In the nodes south of it, two levels down or more from the node injecting the prefix, the route becomes external due to the export policy required to propagate it further down. So, the export policy changes the route type to external as you can see in RIFT running in the host below the router vMX-A6:

```
root@cRPD-A1> show rift routes content

Prefix                         Active Metric N-Hop    All Present
-------------------------------+------+------+--------+-------------------
172.32.0.0/16                S        2 8000a0cf S
172.32.0.6/32                S        2 8000a0c9 S
172.32.0.7/32                S        2 8000a0cc S
172.32.1.3/32                SExt     2 8000a0c9 8453  SExt
::/0                         S        2 8000b3f0 S

root@cRPD-A1> show route 172.32.1.3/32

inet.0: 8 destinations, 8 routes (8 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

172.32.1.3/32   *[Static/200/100] 00:00:24, metric2 0, tag 8453
                 >  to fe80::250:56ff:fea2:e27e via eth4

root@cRPD-A1>
```

In conclusion, this alternate method of using tags provides the simplicity that even if new IP addresses are injected into the RIFT domain, the generic filters remain valid, and there is no need to update them as the match is done on the tag used.

Also, using the RIFT interface passive mode automatically propagates the associated IP addresses as internal routes into RIFT without the need to configure a northbound export and its policy-statement.

The IP addresses for the lo0 interface that need to be propagated *down* the IP fabric have their interface (`mode passive`) configured *with* the tag allowing it. The IP addresses for the lo0 interface that need to be propagated only *up* the IP fabric have their interface (`mode passive`) configured *without* the tag.

3.3.4.4 Exporting RIFT routes to the data center gateway

Typically in DCI (Data Center Interconnect) aggregate spines are connected to the data center gateway routers for connectivity of the data center to the external network and vice versa.

In the case of RIFT, a broader spine is also the ToF which learns all RIFT routes prefixes from the southbound leaf or servers – these routes can be distributed in the BGP by using a Junos BGP export policy. As an example, here is policy to export the RIFT route from the aggregate spine to the data center gateway router:

```
jcluser@AGG-SPINE# show policy-options
policy-statement IPv4-RIFT-TO-BGP {
    term 1 {
        from {
            /* RIFT Protocol IPv4 server/host routes */
            route-filter 1.1.1.0/24 prefix-length-range /30-/32;
        }
        then accept;
    }
    term default {
        then reject;
    }
}
policy-statement IPv6-RIFT-TO-BGP {
    term 1 {
        from {
            /* RIFT Protocol IPv6 server/host routes */
            route-filter 2000:1:1:1:1::/64 prefix-length-range /126-/128;
        }
        then accept;
    }
    term default {
        then reject;
    }
}

[edit]
jcluser@AGG-SPINE#
```

RIFT specific prefixes from the leaf (southbound) are exported to BGP by applying the above policy to the BGP.

For redundancy, BGP peering to two data center gateways is recommended:

```
jcluser@AGG-SPINE# show protocols bgp
group IPv4-DC-GATEWAY {
    type external;
    local-address 192.168.100.2;
    export IPv4-RIFT-TO-BGP;
    peer-as 65000;
    /* Data Center Gateway Router IPv4 address */
    neighbor 192.168.100.1;
}
group IPv6-DC-GATEWAY {
    type external;
    local-address 2001:192:168:100::2;
    export IPv6-RIFT-TO-BGP;
    peer-as 65000;
    /* Data Center Gateway Router IPv6 address */
    neighbor 2001:192:168:100::1;
}

[edit]
jcluser@AGG-SPINE#

jcluser@AGG-SPINE# run show bgp summary
Threading mode: BGP I/O
Groups: 2 Peers: 2 Down peers: 0
Table          Tot Paths  Act Paths Suppressed    History Damp State    Pending
inet.0
                       0          0          0          0          0          0
inet6.0
                       0          0          0          0          0          0
Peer                     AS      InPkt     OutPkt    OutQ   Flaps Last Up/Dwn State|#Active/
Received/Accepted/Damped...
192.168.100.1          65000          6          7       0       0        2:12 Establ
  inet.0: 0/0/0/0
2001:192:168:100::1    65000          6          7       0       0        2:08 Establ
  inet6.0: 0/0/0/0

[edit]
jcluser@AGG-SPINE#
```

RIFT specific prefixes are getting advertise via BGP to the Data Center Gateway router:

```
jcluser@AGG-SPINE# run show route advertising-protocol bgp 2001:192:168:100::1

inet6.0: 21 destinations, 21 routes (21 active, 0 holddown, 0 hidden)
  Prefix                  Nexthop              MED     Lclpref    AS path
* 2000:1:1:1:1::/128      Self                                    I
* 2000:1:1:1:1::1/128     Self                                    I

[edit]
jcluser@AGG-SPINE#

jcluser@AGG-SPINE# run show route protocol rift table inet6.0 match-prefix 2000:1:1:1:1::/128

inet6.0: 21 destinations, 21 routes (21 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both
```

```
2000:1:1:1:1::/128 *[Static/200/100] 00:27:49, metric2 0
                    >  to fe80::250:56ff:fea2:a28d via ge-0/0/2.0
                       to fe80::250:56ff:fea2:108 via ge-0/0/3.0

[edit]
jcluser@AGG-SPINE#
```

3.3.5 Multiple-routing Instances

NOTE The examples described in this section are based on a RIFT experimental image that is not available in the current release but expected to be soon.

As an example, it could be that a data center implementation is simpler without overlay and just needs to support a few environments (< 5). In this case, RIFT allows building independent parallel topologies running in different routing-instances.

This section documents a use case where two independent environments need to be supported (see Figure 3.5):

- A public routing instance covering the whole fabric and having two gateways connected to two PoDs and providing access to external networks (B2B, etc.)
- A private routing instance covering the whole fabric and having several services connected to two PoDs

The inter-connection between the two environments is ensured by a cluster of firewalls connected to the two PoDs. The logical topology can be seen in Figure 3.3.

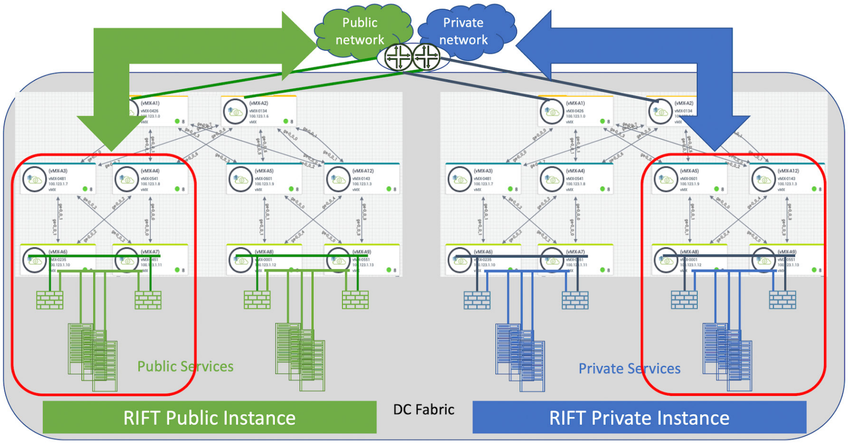


Figure 3.3 RIFT Multiple Routing Instances Lab Topology

In this use case, let's configure a RIFT topology in the global routing table and use a virtual routing instance (vpub) with another RIFT topology inside.

To guarantee isolation between the RIFT topologies, a routing-instance of type vrf is created with its own logical interfaces (ifl) associated with the routing-instance. To support this, create an 802.1q trunk on each of the physical interfaces with a dedicated VLAN per routing-instance.

Here is an example of the routing-instance configuration:

```
routing-instances {
    vpub {
        protocols {
            rift {
                traceoptions {
                    file riftd-topo01 size 3m files 2;
                    level info;
                }
                node-id auto;
                level top-of-fabric;
                lie-receive-address {
                    family {
                        inet 224.0.0.120;
                        inet6 ff02::a1f7;
                    }
                }
                export {
                    southbound {
                        ps-rift_south;
                        allow-rift-routes;
                    }
                }
                interface xe-0/2/0.1 {
                    lie-transmit-address {
                        family {
                            inet 224.0.0.120;
                            inet6 ff02::a1f7;
                        }
                    }
                    bfd-liveness-detection minimum-interval 1000 multiplier 3;
                }
                interface xe-0/2/1.1 {
                    lie-transmit-address {
                        family {
                            inet 224.0.0.120;
                            inet6 ff02::a1f7;
                        }
                    }
                    bfd-liveness-detection minimum-interval 1000 multiplier 3;
                }
                interface lo0.1 {
                    mode passive;
                tags 8453;
                }
            }
        }
        instance-type vrf;
        interface xe-0/2/0.1;
        interface xe-0/2/1.1;
        interface xe-0/3/0.1;
        interface lo0.1;
        route-distinguisher 172.32.0.21:3001;
        vrf-target target:64567:3001;
    }
}
```

Then, any RIFT command for the routing instance must have the `instance $instance-name$` added to it. The instance name can be specified at the beginning of each RIFT command:

```
root@tof01_RE0> show rift instance vpub ?
Possible completions:
  <[Enter]>          Execute this command
  database           Show RIFT link-state database information
  flood-reduction    Show RIFT flood reduction information
  interface          Show RIFT interface information
  node               Show RIFT Node Information
  path-computation   Show RIFT path computation information
  python-tie         Show RIFT TIE information for <node-hex|node-
name>/<North|South>/<node|prefix|positive|negative|key-value|external|ex-disaggregate>/<TIE-number-
hex>
  routes             Show RIFT Routing Table Information.
  tie                Show RIFT TIE information for <node-hex|node-
name>/<North|South>/<node|prefix|positive|negative|key-value|external|ex-disaggregate>/<TIE-number-
hex>
  topology           Show RIFT Topology Information
  versions           Show various package versions
  zero-touch-provisioning  Show RIFT zero-touch provisioning information
  |                  Pipe through a command
root@tof01_RE0>
```

Here's an example of TIEs generated by the TOF node:

```
root@tof01_RE0> show rift instance vpub database content | match "dir|---|002c6bf545e3f502"
Dir Originator     Type   ID      SeqNr  Lifetime   Origin Creation Time Origin   Content   Key ID
---+----------------+---------+--------+----------------+--------+------------+-Lifetime-+--
Size-+--------
S = 002c6bf545e3f502 Node 10000000   5f9030151d24   269 2020/10/21 05:56:35.209     300     None
S  002c6bf545e3f502 Node   10000001  5f9030375658   300 2020/10/21 05:57:06.467     300     None
S  002c6bf545e3f502 Node   10000002  5f903023971f   300 2020/10/21 05:57:06.457     300     None
S  002c6bf545e3f502 Node   10000003  5f9030342cce   294 2020/10/21 05:57:00.573     300     None
S  002c6bf545e3f502 Prefix 2000000e  5f902ff139a4   604751 2020/10/21 05:56:16.920  604800  None
N  002c6bf545e3f502 Node   10000000  5f903015b82a   269 2020/10/21 05:56:35.209     300     None
N  002c6bf545e3f502 Node   10000001  5f903037da99   300 2020/10/21 05:57:06.467     300     None
N  002c6bf545e3f502 Node   10000002  5f903023b24c   300 2020/10/21 05:57:06.457     300     None
N  002c6bf545e3f502 Node   10000003  5f903034af2a   294 2020/10/21 05:57:00.573     300     None
N  002c6bf545e3f502 Prefix 20000046  5f902ff1fc6c   604751 2020/10/21 05:56:16.920  604800  None
```

Or the instance name can also be specified at the end of each RIFT command:

```
root@tof01_RE0> show rift node status ?
Possible completions:
  <[Enter]>          Execute this command
  instance           Routing instance running RIFT
  |                  Pipe through a command
root@tof01_RE0>
```

The different RIFT topologies act like *ships-in-the-night* and are completely isolated. Each RIFT node has a dedicated system-id per instance:

```
root@tof01_RE0> show rift node status instance vpub | grep system
System Name: tof01_RE0, System ID: 002c6bf545e3f502

root@tof01_RE0> show rift node status | grep system
System Name: tof01, System ID: 0000000000000031
```

3.3.6 EVPN Over RIFT

The purpose of this section is to show a brief example of EVPN configuration over RIFT using VXLAN as a tunneling technique. While VXLAN is traditionally used in the data centers, MPLS is also an encapsulation option with EVPN, used more in the metro area but will not be considered here for the sake of brevity. However, the mechanisms with MPLS encapsulation are similar, mainly using MPLS labels versus VXLAN Network IDs (VNI) for VXLAN to identify the LAN. Remember, the VNI is a 24-bit field that is used to uniquely identify the VXLAN network; the VNI is similar to a VLAN ID but having 24 bits allows you to create many more VXLANs than VLANs.

3.3.6.1 Implementation Overview

This section and the associated EVPN configuration examples are based on the legacy way of configuring EVPN over RIFT. Starting with Junos version 20.4, there is a new CLI and a new way of configuring EVPN in an uniform way whatever the platform: using a `mac-vrf` routing instance type. This newer CLI is the recommended option in the future.

The EVPN services supported by the IETF standards and implemented as an example in this section is the `vlan aware bundle service` and is typically used in data center deployments.

So, in this service model, the EVPN instance consists of multiple broadcast domains (for example, multiple VLANs) with each VLAN having its own bridge table (bridge domain). The different bridge tables are maintained by a single MAC-VRF corresponding to the common EVPN instance. So, each VLAN is configured to use a different VXLAN bridge-domain by defining an unique VXLAN Network ID (VNID) per VLAN/Bridge-domain.

The overlay network (EVPN/VXLAN, in this case) can be built over IPv4, over IPv6, or both at the same time.

For node loopback IP address reachability, the links between the nodes must be addressed per address family (IPv4, IPv6). As IPv6 automatically allocates Link-Local Addresses (LLA), the effort to enable EVPN/VXLAN over IPv6 is much simpler as no specific configuration is needed to the auto-allocated LLA addresses.

However, today, IPv4 in the underlay has less limitations than IPv6.

In this implementation used to illustrate EVPN over RIFT, EVPN/VXLAN is built over RIFT only for the IPv4 address family. However, EVPN/VXLAN overlay supports dual-stack IPv4/IPv6 independently of the underlay address family used.

Traditionally, the hosts connect to the fabric using NIC teaming (LACP aggregate interface using EVPN ESI LAG) to two ToR switches. However, due to a current limitation in the virtual lab used, LACP is not supported, so this implementation example is based on single homed hosts.

The topology used in the virtual lab is shown in Figure 3.4.

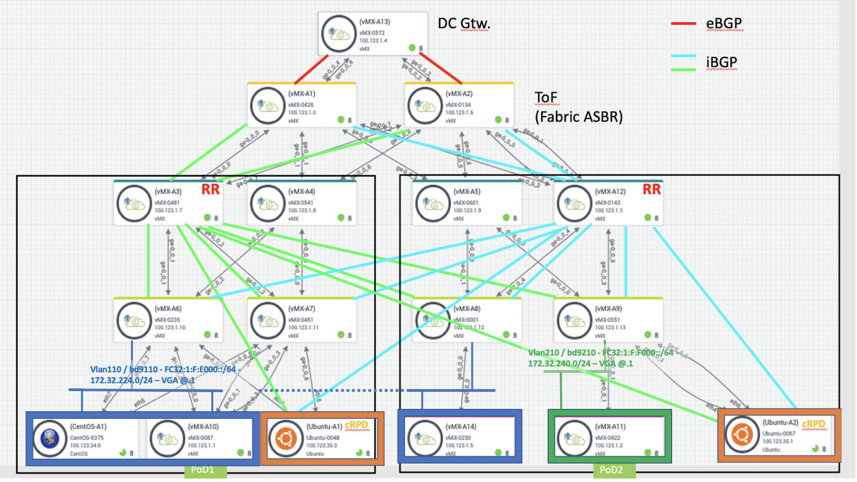


Figure 3.4 *EVPN/VXLAN Overlay Over RIFT for IPv6 Overview*

The virtual lab environment is based on the JCL infrastructure where the DC gateway connects to two RIFT ToF nodes, each of them being connected to two Point of Service Delivery (PoDs). Each PoD is organized in CLOS topology with two spine nodes and two leaf nodes. The leaf nodes provide connectivity to two hosts and one of the hosts has RIFT running on it.

A bridge domain (BD) is configured per VLAN. To show the Layer 2 connectivity inside the PoD and between the PoDs, the VLAN 110 is stretched between the two PoDs. To illustrate the inter-VLAN connectivity, another VLAN 210 is configured in only the second PoD.

An EVPN Edge Routed Bridging (ERB) model is used, meaning that the routing occurs at the edge-routed access device (the leaf layer) where end systems are connected.

As you try to isolate the data center fabric from the outside world as much as possible, a dedicated DC gateway is used to simulate a connection to the MPLS-VPN data center interconnect / distribution / core network.

NOTE In the real environment, for redundancy purposes, at least two DC gateways are used, however, in the virtual lab, only the functionality is provided without redundancy.

From a routing domain point of view, the DC fabric is one domain (BGP AS 64567) and the DC gateway is another (BGP AS 65888). External BGP Labeled Unicast (eBGP-LU) is configured between the DC gateway and the RIFT ToF nodes to exchange the routing information between the two domains while preserving the routing-instances isolation.

Inside the DC fabric BGP domain, two BGP Route Reflectors (BGP RR) are selected (one spine of each PoD) on the RIFT ToF – one level to benefit from the upper layer ToF nodes redundant connectivity and avoid the need to add a horizontal link between the ToF nodes. Also, in the topology used in this example, the spine nodes do not connect any host or gateway, so they do not have any RIFT leaf/edge functionality (RIFT non-edge node), as such, they do not need to be part of the EVPN/VXLAN overlay domain and are therefore less exposed.

The RIFT edge nodes (where the overlay network is built) have an iBGP session between them via the BGP Route Reflectors (RRs). The non-edge RIFT nodes (except if it is a BGP RR) do not participate in the BGP/EVPN/VXLAN, are pure underlay nodes and simply forward the VXLAN tunneled traffic based on the IP addresses of the tunnel endpoints known via RIFT.

3.3.6.2 EVPN/VXLAN Configuration

This section uses the MX style configuration per lab design. Some minor adaptations are needed for QFX switches.

As EVPN (VXLAN) pure Type-5 routes with IPv6. only underlay exists today in beta code (EVPN (VXLAN) pure Type-5 support with IPv6 only underlay is planned for 2021.). Therefore, in this example, the EVPN/VXLAN is built as overlay network on IPv4 only, so IPv6 support is disabled in each RIFT node:

```
deactivate groups rift-defaults protocols rift lie-receive-address family inet6
```

The IPv6 support is disabled for the RIFT node as you can see in the following command output:

```
jcluser@vMX-A6> show rift node status
System Name: vMX-A6, System ID: 002c6bf5b972c000
Level: 22, RIFT Encoding Major: 4, Minor: 0
Flags: overload=False
Capabilities: flood-reduction: True
LIE v4 RX: 224.0.0.120, LIE RX Port: 914
```

```
 Re-Connections: 0
Peers: 6, 3-way: 2, South: 0, North: 2

jcluser@vMX-A6>
```

Two types of loopback IP addresses are used for EVPN/VXLAN that need to be advertised by RIFT in the underlay to support the overlay network:

- Type-1: EVPN control plane / VXLAN data plane address. This address is used by the EVPN nodes to establish the iBGP sessions with the BGP RRs and also as VXLAN tunnel termination addresses.
- Type-2: BGP RR service address. This address is used by the BGP RR to establish the iBGP sessions for the EVPN/VXLAN nodes BGP clients.

Here's the Type-1 loopback address configuration example for the RIFT node vMX-A1:

```
interfaces {
    lo0 {
        unit 0 {
            family inet {
                address 172.32.0.1/32 {
                    primary;
                    preferred;
                }
            }
        }
    }
}
```

A type-2 loopback IP address is also configured in the BGP RR. Here's the Type-2 loopback address configuration example for the RIFT node vMX-A3:

```
interfaces {
    lo0 {
        unit 0 {
            family inet {
                address 172.32.1.3/32;
            }
        }
    }
}
```

As this IP address is used only for the BGP RR, it does not need to be configured as primary/preferred. Moreover, since in this topology the BGP RR is a non-leaf node, it has no VXLAN needs, and there is no type-1 loopback needed. However a type-1 lo0 IP address could be configured as well for configuration uniformity and management.

As you can see above, a dedicated IP address range is used for the type-1 loopback IP addresses and another one for the Type-2 loopback IP addresses. As each loopback address type is exported differently into RIFT, it eases the configuration and the management.

NOTE Please, refer to the section *3.3.4.3 Injecting the Loopback Interfaces into RIFT* for more information on the loopback addresses injection into RIFT.

In the BGP RR, its direct Type-2 loopback IP address is exported southbound and northbound:

```
policy-options {
    policy-statement ps-rift_service {
        term LO0_SERVICE4 {
            from {
                family inet;
                protocol [ direct rift ];
                route-filter 172.32.1.0/24 prefix-length-range /32-/32;
            }
            then accept;
        }
        term DEFAULT_DENY {
            then reject;
        }
    }
}

protocols {
    rift {
        export {
            northbound {
                ps-rift_service;
            }
            southbound {
                ps-rift_service;
                allow-rift-routes;
            }
    }
}
```

The Type-1 and Type-2 loopbacks are all propagated northbound up to the ToF:

```
jcluser@vMX-A1> show rift routes content

Prefix                       Active Metric N-Hop All Present
-------------------------------+------+------+--------+-------------------
172.32.0.0/16                Disc                Disc
172.32.0.6/32                NExt     3 800050cc NExt
172.32.0.7/32                NExt     3 800050cc NExt
172.32.0.8/32                NExt     3 800050cf NExt
172.32.0.9/32                NExt     3 800050cf NExt
172.32.1.3/32                NExt     2 800050cd NExt
172.32.1.12/32               NExt     2 800050ce NExt
```

The other RIFT non-edge nodes propagate southbound the Type-2 loopback addresses received into RIFT:

```
policy-options {
    policy-statement ps-rift_south {
        term LO0_SERVICE4 {
            from {
                family inet;
                protocol rift;
                route-filter 172.32.1.0/24 prefix-length-range /32-/32;
            }
            then accept;
        }
        term DEFAULT_DENY {
            then reject;
        }
    }
}

protocols {
    rift {
        export {
            southbound {
                ps-rift_south;
                allow-rift-routes;
            }
        }
    }
}
```

The RIFT leaf nodes receive the service loopback address of the BGP RR in their PoD:

```
jcluser@vMX-A6> show rift routes content

Prefix                       Active Metric N-Hop All Present
-------------------------------+------+------+--------+-------------------
172.32.0.0/16                S        2 8000b3f1 S
172.32.1.3/32                SExt     2 8000b3f0 SExt
```

The RIFT leaf nodes export their direct Type-1 loopback IP address northbound into the RIFT domain:

```
policy-options {
    policy-statement ps-rift_north {
        term LO0_ONLY4 {
            from {
                family inet;
                protocol direct;
                route-filter 172.32.0.0/24 prefix-length-range /32-/32;
            }
            then accept;
        }
        term DEFAULT_DENY {
            then reject;
        }
    }
}

protocols {
    rift {
        export {
            northbound {
                ps-rift_north;
            }
        }
    }
}
```

Similar to the RIFT leaf nodes, the ToF nodes export their direct Type-1 loopback IP address, but southbound:

```
policy-options {
    policy-statement ps-rift_south {
        term LO0_ONLY4 {
            from {
                family inet;
                protocol direct;
                route-filter 172.32.0.0/24 prefix-length-range /32-/32;
            }
            then accept;
        }
        term DEFAULT_DENY {
            then reject;
        }
    }
}

protocols {
    rift {
        export {
            southbound {
                ps-rift_south;
            }
        }
    }
}
```

To benefit from the DC fabric multi-paths, per flow load sharing is enabled:

```
policy-options {
    policy-statement ps-loadshare {
        term default {
            then {
                load-balance per-packet;
            }
        }
    }
}
routing-options {
    forwarding-table {
        export ps-loadshare;
    }
}
```

In the next example for the BGP RR vMX-A3, a BGP group is configured with the EVPN leaf node as client. The BGP Address Family Identifier AFI=25 (Layer 2 VPN) and sub-Address Family Identifier SAFI=70 (EVPN) is enabled:

```
routing-options {
    router-id 172.32.0.3;
    autonomous-system 64567;
}
protocols {
    bgp {
        group DC_FABRIC_RR_CLIENTS_LEAF {
            type internal;
            local-address 172.32.1.3;
            log-updown;
            family evpn {
                signaling;
            }
            cluster 172.32.1.3;
            multipath;
            neighbor 172.32.0.1;
            neighbor 172.32.0.2;
            neighbor 172.32.0.6;
            neighbor 172.32.0.7;
            neighbor 172.32.0.8;
            neighbor 172.32.0.9;
        }
        mtu-discovery;
        log-updown;
        graceful-restart;
    }
}
```

Also each EVPN node is configured with BGP and the two BGP RRs as neighbors:

```
routing-options {
    router-id 172.32.0.6;
    autonomous-system 64567;
}
protocols {
    bgp {
        group DC_FABRIC_RR {
            type internal;
            local-address 172.32.0.6;
            log-updown;
            family evpn {
                signaling;
            }
            multipath;
            neighbor 172.32.1.3;   # BGP RR vMX-A3
            neighbor 172.32.1.12;  # BGP RR vMX-A12
        }
        mtu-discovery;
        log-updown;
        graceful-restart;
    }
}
```

The BGP session is established with the two BGP RRs and the EVPN address family is negotiated:

```
jcluser@vMX-A6> show bgp summary
Threading mode: BGP I/O
Groups: 1 Peers: 2 Down peers: 0
Table       Tot Paths  Act Paths Suppressed History Damp State Pending
bgp.evpn.0
                   64      32       0       0       0       0
Peer                   AS   InPkt  OutPkt OutQ   Flaps Last Up/Dwn State|#Active/Received/Accepted/
Damped...
172.32.1.3         64567     498     486    0     1  3:15:40 Establ
  __default_evpn__.evpn.0: 0/0/0/0
  bgp.evpn.0: 32/32/32/0
  evi0_vswitch.evpn.0: 26/26/26/0
  v110.evpn.0: 6/6/6/0
172.32.1.12        64567     497     485    0     1  3:15:36 Establ
```

```
  __default_evpn__.evpn.0: 0/0/0/0
  bgp.evpn.0: 0/32/32/0
  evi0_vswitch.evpn.0: 0/26/26/0
  v110.evpn.0: 0/6/6/0
```

As an example for the EVPN leaf node, vMX-A6, two interfaces are configured as access interfaces and associated with the VLAN 110:

```
groups {
    grp-access_type0_1ge {
        interfaces {
            <ge-*> {
                traceoptions {
                    flag event;
                }
                traps;
                Flexible-vlan-tagging;
                mtu 9100;
                encapsulation flexible-ethernet-services;
                aggregated-ether-options {
                    no-flow-control;
                }
                unit <*> {
                    encapsulation vlan-bridge;
                }
            }
        }
    }
}
interfaces {
    ge-0/0/0 {
        apply-groups grp-access_type0_1ge;
        description "vMX-A10 ge-0/0/0 - Customer DC Fabric access - 1000BaseT";
        enable;
        native-vlan-id 110;
        gigether-options {
            auto-negotiation;
        }
        unit 110 {
            enable;
            description "Vlan110: Client Access Interface";
            vlan-id 110;
        }
    }
    ge-0/0/7 {
        apply-groups grp-access_type0_1ge;
        description "Centos-A1 eth7 - Customer DC Fabric access - 1000BaseT";
        enable;
        native-vlan-id 110;
        gigether-options {
            auto-negotiation;
        }
        unit 110 {
            enable;
            description "Vlan110: Client Access Interface";
            vlan-id 110;
        }
    }
}
```

You can see that the interface configuration is based on the SP style interface configuration for the MX that offers great flexibility, however, the enterprise style configuration is also massively used as it offers greater scalability.

It is recommended to have a higher MTU, therefore, jumbo MTU support for the EVPN DC fabric to accommodate the extra headers like VXLAN (50 bytes), 802.1Q (4 bytes), and so on.

A good practice is to configure the DC fabric leaf/spines/DC gateway nodes' core interfaces MTU to 9192 bytes. Any server-facing access interface and any Layer 3 gateway interfaces (IRB interfaces) may be set to an MTU of 9100 bytes.

The VLAN associated IRB interface is configured in each leaf node where the VLAN is connected.

An example for the leaf node vMX-A6 and the VLAN 110 is shown next.

The Virtual Gateway Address (similar to the Virtual Router Redundancy protocol), with a common IP address (IPv4 and IPv6 as dual-stack is supported in the overlay), and associated virtual mac-address, are configured in all the leaf nodes having an IRB interface in the same VLAN.

To ping support on Virtual Gateway Address (VGA) IP, you have to configure the `virtual-gateway-accept-data` knob.

Moreover, if the VGA IP address is lower than the IRB IP address (which is the case here), arping would fail for the IRB IP address because by default the lower IP configured is preferred, so used for ARP resolution. To overcome this, if the VGA is lower than the IRB IP address, configure the `preferred` knob for the IRB IP address:

```
interfaces {
    irb {
        unit 110 {
            virtual-gateway-accept-data;
            description "Tenant 110";
            family inet {
                address 172.32.224.2/24 {
                    preferred;
                    virtual-gateway-address 172.32.224.1;
                }
            }
            family inet6 {
                address fe80::e000:2/64;
                address fc32:1:f:e000::2/64 {
                    virtual-gateway-address fc32:1:f:e000::1;
                }
            }
            virtual-gateway-v4-mac 0a:00:00:00:10:6e;
            virtual-gateway-v6-mac 0a:00:00:00:60:6e;
        }
    }
}
```

The MAC aging timer and the ARP / NDPv6 stale timer should be also optimized, like in any switched network, to reduce the unknown unicast flooding traffic.

The format of VGA MAC address follows the use of locally administered MAC addresses which have the U/L bit (the second least significant bit of the first octet) set to '1' (e.g. starting with '0a').

The default gateway address and the associated virtual MAC addresses are stored in the EVPN database with the state of virtual gateway for the VNI and also the associated remote origins:

```
jcluser@vMX-A6> show evpn database l2-domain-id 9110 extensive
Instance: evi0_vswitch

VN Identifier: 9110, MAC address: 0a:00:00:00:10:6e
  State: 0x0
  Source: 05:00:00:fc:37:00:00:23:96:00, Rank: 1, Status: Active
 Remote origin: 172.32.0.7
 Remote origin: 172.32.0.8
 Mobility sequence number: 0 (minimum origin address 172.32.0.6)
 Timestamp: Aug 28 09:58:14 (0x5f48d536)
 State: <Local-Virtual-Gateway Local-To-Remote-Adv-Allowed Remote-To-Local-Adv-Done>
 MAC advertisement route status: Created
 IP address: 172.32.224.1
   Local origin: irb.110
   Remote origin: 172.32.0.7
   Remote origin: 172.32.0.8
 History db: <No entries>

VN Identifier: 9110, MAC address: 0a:00:00:00:60:6e
  State: 0x0
  Source: 05:00:00:fc:37:00:00:23:96:00, Rank: 1, Status: Active
 Remote origin: 172.32.0.7
 Remote origin: 172.32.0.8
 Mobility sequence number: 0 (minimum origin address 172.32.0.6)
 Timestamp: Aug 28 09:58:14 (0x5f48d536)
 State: <Local-Virtual-Gateway Local-To-Remote-Adv-Allowed Remote-To-Local-Adv-Done>
 MAC advertisement route status: Created
 IP address: fc32:1:f:e000::1
 Local origin: irb.110
   Remote origin: 172.32.0.7
   Remote origin: 172.32.0.8
 History db: <No entries>
```

The EVPN Virtual Instance (EVI) is then configured. As the VLAN Aware Bundle service is used in this example, a single EVI and a bridge domain per VLAN is needed. As the book's demo lab is based on vMX devices, the EVI in this case is configured using a routing-instance of type `virtual-switch`. In this EVI, EVPN is configured with VXLAN as data encapsulation tunneling and the `extended-vni-list all` configuration automatically associates all the configured bridge domains in the EVPN as part of the EVI instance.

As multicast is not configured, Broadcast, Unknown unicast and Multicast (BUM) traffic is flooded using the ingress-replication. However, due to the EVPN control plane managing the Layer 2 addresses' learning and distribution, Broadcast and Unknown unicast traffic is highly reduced. Nevertheless, if there is high bandwidth multicast traffic (such as IPTV), optimizations should be considered like Selective Multicast Ethernet Tag, Assisted Replication (AR), and multicast distribution (ERB multicast is planned for Junos 21.1), etc.

Since a VGA is used, there is no need to advertise to the other gateways the MAC address and the IP address besides the VGA address of the IRB interface. Therefore, the `default-gateway no-gateway-community` is configured for that. Without the `no-gateway-community` knob, each EVPN with a configured VGA will treat other IRB IP addresses (and associated MAC addresses) as its own IP address to provide GW redundancy for VM motion. But in this case, VGA provides such a redundancy.

In the EVI, each `bridge-domain` is a VLAN with some associated Level 2 interfaces and also an Integrated Routing and Bridging (IRB) interface where the default gateway function for the VLAN happens. Per `bridge-domain`, an unique VXLAN Network Identifier is used. In the example below, the VNI `9110` is configured for the VLAN 110. This VNI tag identifies the bridge domain in the VXLAN world.

```
routing-instances {
    evi0_vswitch {
        protocols {
            evpn {
                encapsulation vxlan;
                extended-vni-list all;
                multicast-mode ingress-replication;
                default-gateway no-gateway-community;
            }
        }
        description "Vlan Aware Bundle Default EVI";
        vtep-source-interface lo0.0;
        instance-type virtual-switch;
        bridge-domains {
            bd9110 {
                vlan-id 110;
                interface ge-0/0/0.110;
                interface ge-0/0/7.110;
                routing-interface irb.110;
                vxlan {
                    vni 9110;
                    ingress-node-replication;
                }
            }
        }
        route-distinguisher 172.32.0.6:1;
        vrf-target {
            target:64567:9000;
            auto;
        }
    }
}
```

Ingress node replication is also used, meaning BUM traffic propagation requires the ingress node to copy the traffic for each destination leaf node. A route distinguisher is uniquely configured per node and per virtual instance.

Remember that the route target is used to automatically import/export routes into a routing instance (that can be manually overwritten with an import/export policy).

As a best practice, two different route targets are configured as Type-1 routes (EVI and ESI auto-discovery) and are network wide messages while Type-2 and 3 are per-VLAN routes. Having two different RTs allows you to limit the distribution of bridge domain specific EVPN route types (Type 2 and Type 3) to only the interested devices.

You can see that the first route target configuration (`target:64567:9000`) is used to tag the EVPN type-1 routes. This route target must be configured to the same value for all the node members of the same EVI. The second route target is configured as `auto` (supported beginning with Junos 19.1R1). As this route target is automatically defined based on the BGP Autonomous System (AS) number and the VNI, this statement ensures that all devices use the same VNI and the same global ASN to automatically generate a consistent target community and the corresponding policy to go with the community. The RT is derived from the BGP AS number and the VNI as specified in the RFC8365 (see https://tools.ietf.org/html/rfc8365). So attention must be paid if the BGP AS number is a four octet, or if two DC locations with stretched VLANs do not use the same BGP AS number. In these cases, the route target will have to be overwritten manually.

When calculating the RT according to RFC8365, the route target should be:

```
target:AS#:ATTTDDDDSSSSSSSSSSSSSSSSSSSSSSSS
```

Where:

- `A` is 0 when auto derived (1 for manual configuration), so 0 in this case
- `TTT` is the type, so in this case '001' for the VXLAN type
- `SSS...S` is the service, so in this example the VNI ID binary (9110), which can be calculated as:
  `(16 * 224) + 9110 = 268444566`

So the calculated RT is 64567:268444566 as seen in the next output. The routing instance can be verified with its interfaces, auto configured route target, and the associated import/export policies:

```
jcluser@vMX-A6> show route instance evi0_vswitch extensive
evi0_vswitch:
  Description: Vlan Aware Bundle Default EVI
  Router ID: 0.0.0.0
  Type: evpn           State: Active
  Interfaces:
 vtep.32771
 vtep.32770
 vtep.32769
 ge-0/0/7.110
 ge-0/0/0.110
  Route-distinguisher: 172.32.0.6:1
  Vrf-import: [ __vrf-import-autoderive-evi0_vswitch-internal__ ]
  Vrf-export: [ __vrf-export-evi0_vswitch-internal__ ]
  Vrf-import-target: [ target:64567:268444566 ]
  Vrf-import-target: [ target:64567:9000 ]
  Vrf-export-target: [ target:64567:9000 ]
  Fast-reroute-priority: low
  Tables:
 evi0_vswitch.evpn-mac.0: 0 routes (0 active, 0 holddown, 0 hidden)
 evi0_vswitch.evpn.0 : 66 routes (40 active, 0 holddown, 0 hidden)

jcluser@vMX-A6> show policy __vrf-import-autoderive-evi0_vswitch-internal__
Policy __vrf-import-autoderive-evi0_vswitch-internal__:
 Term 9110:
     from community __vrf-community-evi0_vswitch-9110-internal__ [target:64567:268444566 ]
     then accept
 Term unnamed:
     from community __vrf-community-evi0_vswitch-common-internal__ [target:64567:9000 ]
     then accept
 Term unnamed:
     then reject

cluser@vMX-A6> show policy __vrf-export-evi0_vswitch-internal__
Policy __vrf-export-evi0_vswitch-internal__:
 Term unnamed:
     then community + __vrf-community-evi0_vswitch-common-internal__ [target:64567:9000 ] accept

jcluser@vMX-A6> show policy __vrf-export-v110-internal__
Policy __vrf-export-v110-internal__:
 Term unnamed:
     then community + __vrf-community-v110-common-internal__ [target:64567:1110 ] accept

jcluser@vMX-A6>
```

The interfaces for a bridge domain in the EVI are verified with the following command:

```
jcluser@vMX-A6> show bridge domain

Routing instance     Bridge domain         VLAN ID  Interfaces
evi0_vswitch         bd9110                110
                                                    esi.582
                                                    ge-0/0/0.110
                                                    ge-0/0/7.110
                                                    vtep.32769
                                                    vtep.32770
                                                    vtep.32771

jcluser@vMX-A6>
```

The list of vtep interfaces can be also identified with the following command:

```
jcluser@vMX-A6> show interfaces vtep
Physical interface: vtep, Enabled, Physical link is Up
  Interface index: 134, SNMP ifIndex: 519
  Type: Software-Pseudo, Link-level type: VxLAN-Tunnel-Endpoint, MTU: Unlimited, Speed: Unlimited
  Device flags   : Present Running
  Link type   : Full-Duplex
  Link flags  : None
 Last flapped   : Never
 Input packets : 0
 Output packets: 0
  Logical interface vtep.32768 (Index 340) (SNMP ifIndex 555)
 Flags: Up SNMP-Traps 0x4000 Encapsulation: ENET2
 Ethernet segment value: 00:00:00:00:00:00:00:00:00:00, Mode: single-homed, Multi-
homed status: Forwarding
 VXLAN Endpoint Type: Source, VXLAN Endpoint Address: 172.32.0.6, L2 Routing Instance: evi0_
vswitch, L3 Routing Instance: default
 Input packets : 0
 Output packets: 0

  Logical interface vtep.32769 (Index 339) (SNMP ifIndex 559)
 Flags: Up SNMP-Traps Encapsulation: ENET2
 VXLAN Endpoint Type: Remote, VXLAN Endpoint Address: 172.32.0.8, L2 Routing Instance: evi0_
vswitch, L3 Routing Instance: default
 Input packets : 102
 Output packets: 3298
 Protocol bridge, MTU: Unlimited
   Flags: Trunk-Mode

  Logical interface vtep.32770 (Index 342) (SNMP ifIndex 560)
 Flags: Up SNMP-Traps Encapsulation: ENET2
VXLAN Endpoint Type: Remote, VXLAN Endpoint Address: 172.32.0.9, L2 Routing Instance: evi0_
vswitch, L3 Routing Instance: default
 Input packets : 0
 Output packets: 3298
 Protocol bridge, MTU: Unlimited
   Flags: Trunk-Mode
  Logical interface vtep.32771 (Index 346) (SNMP ifIndex 561)
 Flags: Up SNMP-Traps Encapsulation: ENET2
 VXLAN Endpoint Type: Remote, VXLAN Endpoint Address: 172.32.0.7, L2 Routing Instance: evi0_
vswitch, L3 Routing Instance: default
 Input packets : 3155
 Output packets: 3298
 Protocol bridge, MTU: Unlimited
   Flags: Trunk-Mode
```

You can check as well that the EVPN Type-1 routes are tagged with the EVI unique static route target and imported in the EVI default table:

```
jcluser@vMX-A6> show route table bgp.evpn.0 match-prefix 1:172.32.0.7:0::050000fc370000239600::FFFF:
FFFF/192 detail <<< FFFF:FFFF for type-1 means ADiscovery
ESI Specific route
bgp.evpn.0: 49 destinations, 81 routes (49 active, 0 holddown, 0 hidden)
1:172.32.0.7:0::050000fc370000239600::FFFF:FFFF/192 AD/ESI (2 entries, 0 announced)
     *BGP Preference: 170/-101
             Route Distinguisher: 172.32.0.7:0
             Next hop type: Indirect, Next hop index: 0
             Address: 0xc7425f4
             Next-hop reference count: 52
             Source: 172.32.1.3
             Protocol next hop: 172.32.0.7
             Indirect next hop: 0x2 no-forward INH Session ID: 0x0
             State: <Active Int Ext>
             Local AS: 64567 Peer AS: 64567
             Age: 7:10:14 Metric2: 0
             Validation State: unverified
             Task: BGP_64567.172.32.1.3
             AS path: I  (Originator)
             Cluster list:  172.32.1.3
             Originator ID: 172.32.0.7
             Communities: target:64567:9000 encapsulation:vxlan(0x8) esi-label:0x0:all-
active (label 0)
             Import Accepted
             Route Label: 1
             Localpref: 100
             Router ID: 172.32.0.3
             Secondary Tables: evi0_vswitch.evpn.0
```

As you can see, this EVPN Type-1 route is imported into the EVI EVPN table based on its manually configured route target and the auto import policy:

```
jcluser@vMX-A6> show route match-prefix 1:172.32.0.7:0::050000fc370000239600::FFFF:FF
FF/192 extensive | match "1:172|destinat|Communit"

bgp.evpn.0: 49 destinations, 81 routes (49 active, 0 holddown, 0 hidden)
1:172.32.0.7:0::050000fc370000239600::FFFF:FFFF/192 AD/ESI (2 entries, 0 announced)
             Communities: target:64567:9000 encapsulation:vxlan(0x8) esi-label:0x0:all-
active (label 0)
             Communities: target:64567:9000 encapsulation:vxlan(0x8) esi-label:0x0:all-
active (label 0)

v110.evpn.0: 8 destinations, 14 routes (8 active, 0 holddown, 0 hidden)

__default_evpn__.evpn.0: 1 destinations, 1 routes (1 active, 0 holddown, 0 hidden)

evi0_vswitch.evpn.0: 40 destinations, 66 routes (40 active, 0 holddown, 0 hidden)
1:172.32.0.7:0::050000fc370000239600::FFFF:FFFF/192 AD/ESI (2 entries, 1 announced)
             Communities: target:64567:9000 encapsulation:vxlan(0x8) esi-label:0x0:all-
active (label 0)
             Communities: target:64567:9000 encapsulation:vxlan(0x8) esi-label:0x0:all-
active (label 0)

jcluser@vMX-A6>
```

And the EVPN Type-2 and Type-3 routes are tagged with an automatic prefix derived from the BGP AS number and the VNI:

```
jcluser@vMX-A6> show route table bgp.evpn.0 match-prefix 3:172.32.0.7:1::9110::172.32.0.7/248 detail

bgp.evpn.0: 49 destinations, 81 routes (49 active, 0 holddown, 0 hidden)
3:172.32.0.7:1::9110::172.32.0.7/248 IM (2 entries, 0 announced)      <<< type-3 route
     *BGP Preference: 170/-101
             Route Distinguisher: 172.32.0.7:1
             PMSI: Flags 0x0: Label 569: Type INGRESS-REPLICATION 172.32.0.7
             Next hop type: Indirect, Next hop index: 0
             Address: 0xc7425f4
             Next-hop reference count: 52
             Source: 172.32.1.3
             Protocol next hop: 172.32.0.7
             Indirect next hop: 0x2 no-forward INH Session ID: 0x0
             State: <Active Int Ext>
             Local AS: 64567 Peer AS: 64567
             Age: 7:27:45 Metric2: 0
             Validation State: unverified
             Task: BGP_64567.172.32.1.3
             AS path: I  (Originator)
             Cluster list:  172.32.1.3
             Originator ID: 172.32.0.7
             Communities: target:64567:268444566 encapsulation:vxlan(0x8)
             Import Accepted
             Localpref: 100
             Router ID: 172.32.0.3
             Secondary Tables: evi0_vswitch.evpn.0
jcluser@vMX-A6> show route table bgp.evpn.0 match-prefix 2:172.32.0.7:1::9110::2c:6b:f5:e1:1f
:f0::172.32.224.3/304 detail

bgp.evpn.0: 49 destinations, 81 routes (49 active, 0 holddown, 0 hidden)
2:172.32.0.7:1::9110::2c:6b:f5:e1:1f:f0::172.32.224.3/304 MAC/IP (2 entries, 0 announced)
     *BGP Preference: 170/-101
             Route Distinguisher: 172.32.0.7:1
             Next hop type: Indirect, Next hop index: 0
             Address: 0xc7425f4
             Next-hop reference count: 52
             Source: 172.32.1.3
             Protocol next hop: 172.32.0.7
             Indirect next hop: 0x2 no-forward INH Session ID: 0x0
             State: <Active Int Ext>
             Local AS: 64567 Peer AS: 64567
             Age: 7:30:27 Metric2: 0
             Validation State: unverified
             Task: BGP_64567.172.32.1.3
             AS path: I  (Originator)
             Cluster list:  172.32.1.3
             Originator ID: 172.32.0.7
             Communities: target:64567:268444566 encapsulation:vxlan(0x8)
             Import Accepted
             Route Label: 9110
             ESI: 00:00:00:00:00:00:00:00:00:00
             Localpref: 100
             Router ID: 172.32.0.3
             Secondary Tables: evi0_vswitch.evpn.0
```

And they are also auto imported into the EVI routing instance table based on the auto configured route target:

```
jcluser@vMX-A6> show route match-prefix 2:172.32.0.7:1::9110::2c:6b:f5:e1:1f
:f0::172.32.224.3/304 | match "2:172|destinat|communit"
bgp.evpn.0: 48 destinations, 80 routes (48 active, 0 holddown, 0 hidden)
2:172.32.0.7:1::9110::2c:6b:f5:e1:1f:f0::172.32.224.3/304 MAC/IP

evi0_vswitch.evpn.0: 39 destinations, 65 routes (39 active, 0 holddown, 0 hidden)
2:172.32.0.7:1::9110::2c:6b:f5:e1:1f:f0::172.32.224.3/304 MAC/IP
```

The MAC addresses learned by the EVPN leaf nodes in the bridge domain are exchanged by the BGP EVPN family and they populate the EVPN database:

```
jcluser@vMX-A6> show evpn database l2-domain-id 9110
Instance: evi0_vswitch
VLAN  DomainId  MAC address     Active source                Timestamp     IP address
  9110    0a:00:00:00:10:6e  05:00:00:fc:37:00:00:23:96:00  Aug 28 09:58:14  172.32.224.1
  9110    0a:00:00:00:60:6e  05:00:00:fc:37:00:00:23:96:00  Aug 28 09:58:14  fc32:1:f:e000::1
  9110    00:50:56:a2:58:3e  ge-0/0/0.110                   Aug 28 17:32:34  172.32.224.10
                                                                                 fc32:1:f:e000::10

fe80::250:56ff:fea2:583e
  9110    00:50:56:a2:60:ec  172.32.0.8                     Aug 28 13:03:37  172.32.224.14
```

```
                                                                                  fc32:1:f:e000::14

fe80::250:56ff:fea2:60ec
  9110     00:50:56:a2:cd:97  172.32.0.7                    Aug 28 17:31:39
  9110     00:50:56:a2:e3:e4  ge-0/0/7.110                  Aug 28 17:31:46
  9110     2c:6b:f5:6d:c6:f0  172.32.0.8                    Aug 28 09:58:14  172.32.224.4
                                                                                  fc32:1:f:e000::4
                                                                                  fe80::e000:4
  9110     2c:6b:f5:b9:72:f0  irb.110                       Aug 28 07:50:24  172.32.224.2
                                                                                  fc32:1:f:e000::2
                                                                                  fe80::e000:2
  9110     2c:6b:f5:e1:1f:f0  172.32.0.7                    Aug 28 09:58:14  172.32.224.3
                                                                                  fc32:1:f:e000::3
                                                                                  fe80::e000:3
```

Finally, the routing-instance of type VRF is configured to contain the Level 3 routes (Direct, EVPN type-5, or optionally other imported).

In this example, a group is used to configure general routing-instance features like `multipath` (protocol-independent load balancing for Layer 3 / ECMP), `auto-export` (leak routes between VPN routing and forwarding (VRF) instances that are locally configured by evaluating the (auto) export policy of each VRF).

The VXLAN encapsulation is configured for the EVPN Type-5 routes:

```
groups {
    grp-vrf_evpn_defaults {
        routing-instances {
            <*> {
                routing-options {
                    multipath;
                    auto-export;
                }
                protocols {
                    evpn {
                        ip-prefix-routes {
                            encapsulation vxlan;
                        }
                    }
                }
                instance-type vrf;
                vrf-table-label;
            }
        }
    }
}
```

The routing instance v110 is configured using the default values of the `grp-vrf_evpn_defaults` group.

Next, the `ip-prefix-routes advertise direct-nexthop` generates pure EVPN Type-5 routes for the connected (direct) networks (irb.110 in this case) and associate the VNI ‘1110’ in this example for the routing instance (similar to the VRF table tag in the MPLS-VPN scenario). Remember, a pure Type-5 route advertises the summary IP prefix and includes a BGP extended community called a *router MAC*, which is used to carry the MAC address of the sending switch (IRB IFD interface) and provides next-hop reachability information for the prefix.

The interfaces part of the `routing-instances` is also defined.

When a manually configured routing instance import policy is used as in this example, you need to import the EVPN type-5 routes from the routing instance v210 into this routing instance to have connectivity between both. For stronger traffic control between routing instances (and example being from different tenant networks), a firewall is usually deployed instead of simply leaking the routes between the routing instances:

```
routing-instances {
    v110 {
        apply-groups grp-vrf_evpn_defaults;
        routing-options {
            rib v110.inet6.0 {
                multipath;
            }
        }
        protocols {
            evpn {
                ip-prefix-routes {
                    advertise direct-nexthop;
                    vni 1110;
                }
            }
        }
        description "VRF TENANT 110";
        interface irb.110;
        interface lo0.110;
        route-distinguisher 172.32.0.6:1110;
        vrf-import ps-prefixes_to_v110;
        vrf-target target:64567:1110;
    }
}
policy-options {
    policy-statement ps-prefixes_to_v110 {
        term import_same_rt {
            from community co-rt_bd9110_pfx;
            then accept;
        }
        term import_other_rt {
            from community co-rt_bd9210_pfx;
            then accept;
        }
        term reject {
            then reject;
        }
    }
    community co-rt_bd9110_pfx members target:64567:1110;
    community co-rt_bd9210_pfx members target:64567:1210;
}
```

You can see that the direct routes exported as EVPN Type-5 in the configuration are the subnet prefixes configured at the irb interfaces (/24 subnets in this case, or /64 for IPv6). It means that the traffic with this configuration has a high chance to be asymmetric (traffic from leaf vMX-A6 going directly to leaf vMX-A9 but returning traffic going from leaf vMX-A9 to leaf vMX-A7, then forwarded back to leaf vMX-A6). This asymmetry happens because vMX-A9 has no idea behind what leaf the host centos-A1 (source of the traffic here) is connected and has only the same information of multiple leaf nodes (vMX-A6, vMX-A7, and vMX-A8 in this topology) advertising the same subnet. To avoid this issue and to be asymmetric, in this case, an export policy must be configured under `routing-instances v110 protocols evpn ip-prefix-routes` to export the direct routes but also the host EVPN routes with a prefix length of 32 (or 128 for IPv6). Like this, the host routes of the servers connected behind the irb interfaces will be advertised as Type-5 as well.

A summary of the created context for pure EVPN Type-5 exported routes can be seen here:

```
jcluser@vMX-A6> show evpn l3-context v110 extensive
L3 context: v110
  Type: Configured
  Advertisement mode: Direct nexthop, Router MAC: 2c:6b:f5:b9:72:f0
  Encapsulation: VXLAN, VNI: 1110
  IPv4 source VTEP address: 172.32.0.6
  Flags: 0x9 <Configured IRB-MAC>
  Change flags: 0x0
  Composite nexthop support: Enabled
  Route Distinguisher: 172.32.0.6:1110
  Reference count: 11

jcluser@vMX-A6>
```

The prefixes received via EVPN pure Type-5 route from the routing instance v110 are imported per the configured policy into the routing instance v210:

```
jcluser@vMX-A9> show evpn ip-prefix-database l3-context v210 prefix 172.32.224.0/24
L3 context: v210

EVPN->IPv4 Imported Prefixes
Prefix                                    Etag
172.32.224.0/24                           0
  Route distinguisher VNI/Label  Router MAC       Nexthop/Overlay GW/ESI
  172.32.0.6:1110     1110    2c:6b:f5:b9:72:f0  172.32.0.6     <<< The 3 leaf nodes hosting the VNI
  172.32.0.7:1110     1110    2c:6b:f5:e1:1f:f0  172.32.0.7
  172.32.0.8:1110     1110    2c:6b:f5:6d:c6:f0  172.32.0.8

jcluser@vMX-A9> show evpn ip-prefix-database l3-context v210 prefix 172.32.224.0/24 extensive
L3 context: v210

EVPN->IPv4 Imported Prefixes

Prefix: 172.32.224.0/24, Ethernet tag: 0
  Change flags: 0x0
  Remote advertisements:
 Route Distinguisher: 172.32.0.6:1110
   VNI: 1110
   Router MAC: 2c:6b:f5:b9:72:f0
   BGP nexthop address: 172.32.0.6
   IP route status: Created
 Route Distinguisher: 172.32.0.7:1110
   VNI: 1110
   Router MAC: 2c:6b:f5:e1:1f:f0
   BGP nexthop address: 172.32.0.7
   IP route status: Created
 Route Distinguisher: 172.32.0.8:1110
   VNI: 1110
   Router MAC: 2c:6b:f5:6d:c6:f0
   BGP nexthop address: 172.32.0.8
   IP route status: Created
```

As seen above, the EVPN pure Type-5 route contains the MAC address of the remote gateway in order to build the routed frame tunneled to the remote gateway where the destination resides.

As said previously, this MAC address is common for all the VNIs configured per router. The MAC address of the IRB physical interface is used for that purpose:

```
jcluser@vMX-A6> show interfaces irb
Physical interface: irb, Enabled, Physical link is Up
  Interface index: 133, SNMP ifIndex: 512
  Type: Ethernet, Link-level type: Ethernet, MTU: 1514
  Device flags   : Present Running
  Interface flags: SNMP-Traps
  Link type   : Full-Duplex
  Link flags  : None
  Current address: 2c:6b:f5:b9:72:f0, Hardware address: 2c:6b:f5:b9:72:f0
```

And all the paths are installed into the v210.inet.0 table due to the configured multipath option:

```
jcluser@vMX-A9> show route table v210.inet.0 172.32.224.0/24

v210.inet.0: 6 destinations, 9 routes (6 active, 0 holddown, 0 hidden)
@ = Routing Use Only, # = Forwarding Use Only
+ = Active Route, - = Last Active, * = Both

172.32.224.0/24 @[EVPN/170] 08:04:18
                    to 172.32.133.5 via ge-0/0/0.0
                 >  to 172.32.134.5 via ge-0/0/3.0
                 [EVPN/170] 2d 05:41:31
                    to 172.32.133.5 via ge-0/0/0.0
                 >  to 172.32.134.5 via ge-0/0/3.0
                 [EVPN/170] 2d 05:41:31
                    to 172.32.133.5 via ge-0/0/0.0
                 >  to 172.32.134.5 via ge-0/0/3.0
                #[Multipath/255] 08:04:18, metric2 0
                    to 172.32.133.5 via ge-0/0/0.0
                 >  to 172.32.134.5 via ge-0/0/3.0
                    to 172.32.133.5 via ge-0/0/0.0
                 >  to 172.32.134.5 via ge-0/0/3.0
                    to 172.32.133.5 via ge-0/0/0.0
                 >  to 172.32.134.5 via ge-0/0/3.0
```

This section of Chapter 3 showed how RIFT in the underlay brings simplicity and efficiency to the overlay (EVPN-VXLAN in this example) and better layer isolation, using a simple and efficient RIFT IGP versus a traditional DC design with BGP also in the underlay.

Chapter 4

Junos RIFT Monitoring and Troubleshooting

4.1 RIFT Monitoring

Chapter 4 offers examples of Junos CLI commands that can be used to monitor and troubleshoot the RIFT IP fabric.

As a general rule, residing at the root of the tree the ToF nodes have full visibility and a full database view of the IP fabric south of them, like watchmen. Therefore, this is a good place to start monitoring and investigating the IP fabric because of that global view.

4.1.1 Is There a Connectivity Issue in the RIFT Domain?

Positive disaggregation happens to recover from a connectivity issue, so their presence is an indication of a link or adjacency issue happening:

```
jcluser@vMX-A9> show rift database content | match "Dir|---|pos"
Dir Originator    Type   ID      SeqNr      Lifetime   Origin Creation Time   Origin   Content   Key ID
---+----------------+---------+--------+----------------+--------+---------------------+-Lifetime-
+--Size-+--------
S   002c6bf5754cc000 PosExt 70000011  5edf8b318ea2   604706 2020/06/09 13:14:25  604800  167     0
S   002c6bf5754cc000 PosExt 70000053  5edf8b315640   604706 2020/06/09 13:14:25  604800  183     0

jcluser@vMX-A9>
```

You can see there is a TIE related to a positive disaggregation.

The `Lifetime` field provides the lifetime of the TIE or how long the TIE is valid and will remain in the database. When the lifetime is over, the TIE is purged from the database. Look at the details for this external disaggregate TIE:

```
jcluser@vMX-A9> show rift tie 002c6bf5754cc000/S/ex-disaggregate/70000053
TIE ID: 002c6bf5754cc000/S/PosExt__/70000053

 Prefix                        Metric LPB ATT On Link
 -------------------------------+------+---+---+-------
  fc32:1:f::8/128                   2
```

You can see the information on the disaggregated route. In this case, the loopback IP address of the vMX-A8 is positively disaggregated by vMX-A12, which indicates an adjacency issue between vMX-A8 and vMX-A5 in our lab topology (See Figure 3.1)

4.1.2 Is the RIFT Domain Stable?

By looking at the number of times RIFT had to compute new paths, you can gauge the RIFT domain's stability:

```
jcluser@vMX-A9> show rift path-computation statistics

                 +--------------- Total ------------+
Dir   Runs   Nodes#  |Nodes#   Prfxs#   Deltas   Holds  |Last Run Last Trigger  On
-----+------+--------+--------+--------+--------+-------+-------------+---------+-------------
South  53     1    51   231    44    0  13:20:28.086         002c6bf5689bc000/S/
External/60000038 13:20:28.086
North  53     3   132     0     0   65  13:20:28.086     002c6bf52155c000/N/
Node____/10000000 12:44:00.940

jcluser@vMX-A9>
```

The number of times the path computation was run in the tree provides interesting information on the stability of the RIFT domain.

By looking at the TIE that triggered the path computation, you can identify the zone of concern for the last trigger.

Use `show rift database statistics` to see the number of positive disaggregation TIEs sent, which is also a good indication of RIFT domain connectivity instability:

```
jcluser@vMX-A9> show rift database statistics
Peers: Configured 8, in 3-WAY 2, Last UP/DOWN 2020/06/09 12:44:10.429
Last New TIE: 002c6bf5689bc000/S/External/60000038, on 2020/06/09 13:20:28.086
TIE Version Collisions: 0

Dir   Type   #TIES
-----+---------+-----+
South
   External  14
   PosExt     2
   Node       4
   Prefix     6

Dir   Type   #TIES
-----+---------+-----+
North
   External   2
   Node       1

jcluser@vMX-A9>
```

4.1.3 ZTP Auto-configuration Process Verification

A RIFT node will first determine its level based on the highest available level (HAL) value seen from all the valid offers levels (VOL) received. To verify the ZTP process, the `show rift zero-touch-provisioning statistics` command displays statistics about the offers:

```
jcluser@vMX-A6> show rift zero-touch-provisioning statistics
Current Level: 22, Last Change: 2020/05/15 08:04:50.968, Total Changes: 1
HAL Offers: 002c6bf56118c000, 002c6bf5e4f2c000
Last HAL Offer On: 2020/05/15 08:04:50.968

jcluser@vMX-A6>
```

It is important to verify that valid offers are received and that the node succeeded in auto-configuring its level. The `show rift interface status` provides you with the auto-discovered adjacencies:

```
jcluser@vMX-A6> show rift interface status
Link ID: 257, Interface: ge-0/0/0.0
Status Admin True, Platform True, State: OneWay
LIE TX V4: 224.0.0.120, LIE TX V6: ff02::a1f7, LIE TX Port: 914, TIE RX Port: 915
PoD 0, Nonce 19148

Link ID: 258, Interface: ge-0/0/1.0
Status Admin True, Platform True, BFD True, State: ThreeWay, 3-Way Uptime: 5 hours, 33 minutes, 39
seconds
LIE TX V4: 224.0.0.120, LIE TX V6: ff02::a1f7, LIE TX Port: 914, TIE RX Port: 915
PoD 0, Nonce 6953
Neighbor: ID 002c6bf5e4f2c000, Link ID 258, Name: vMX-A3:ge-0/0/1.0, Level: 23
 TIE V6: fe80::250:56ff:fea2:fbfa, TIE Port: 915, BW 1000 MBits/s
PoD: None, Nonce: 16484, Outer Key: 0, Holdtime: 3 secs

Link ID: 259, Interface: ge-0/0/2.0
Status Admin True, Platform True, BFD True, State: ThreeWay, 3-Way Uptime: 5 hours, 33 minutes, 39
seconds
LIE TX V4: 224.0.0.120, LIE TX V6: ff02::a1f7, LIE TX Port: 914, TIE RX Port: 915
PoD 0, Nonce 9457
Neighbor: ID 002c6bf56118c000, Link ID 259, Name: vMX-A4:ge-0/0/2.0, Level: 23
 TIE V6: fe80::250:56ff:fea2:9c86, TIE Port: 915, BW 1000 MBits/s
PoD: None, Nonce: 24090, Outer Key: 0, Holdtime: 3 secs

Link ID: 260, Interface: ge-0/0/3.0
Status Admin True, Platform False
LIE TX V4: 224.0.0.120, LIE TX V6: ff02::a1f7, LIE TX Port: 914, TIE RX Port: 915
PoD 0, Nonce 15097

Link ID: 264, Interface: ge-0/0/7.0
Status Admin True, Platform True, State: OneWay
LIE TX V4: 224.0.0.120, LIE TX V6: ff02::a1f7, LIE TX Port: 914, TIE RX Port: 915
PoD 0, Nonce 11820
...
```

Interfaces displayed with `Platform False` mean RIFT is enabled for the interface, but the interface is not up. Interfaces displayed with `State OneWay`, means RIFT is enabled for the interface, and the interface is up, but no RIFT neighbor was discovered on that interface.

Finally, the `show rift node status` command displays the status of the RIFT node (its auto-configured level and the number of southbound and northbound neighboring nodes).

```
jcluser@vMX-A6> show rift node status
System Name: vMX-A6, System ID: 002c6bf5f32fc000
Level: 22, RIFT Encoding Major: 4, Minor: 0
Flags: overload=False
Capabilities: flood-reduction: True
LIE v4 RX: 224.0.0.120, LIE v6 RX: ff02::a1f7, LIE RX Port: 914
 Re-Connections: 0
Peers: 8, 3-way: 2, South: 0, North: 2

jcluser@vMX-A6>
```

The counter for `Re-Connections` shown here is related to the Redis server. Please, refer to the Appendix (*Redis Persistency and Analytics*) for more details on the Redis server.

To monitor the performance needed for a RIFT node to auto-configure based on the ZTP process, first identify when the first RIFT interface started, using the `show rift interface statistics | match Start` command:

```
jcluser@vMX-A6> show rift interface statistics | match Start
Link ID: 257, Interface: ge-0/0/0.0, Started: 2020/05/15 08:04:50.587
Link ID: 258, Interface: ge-0/0/1.0, Started: 2020/05/15 08:04:50.716
Link ID: 259, Interface: ge-0/0/2.0, Started: 2020/05/15 08:04:50.791
Link ID: 260, Interface: ge-0/0/3.0, Started: 2020/05/15 08:04:50.868
Link ID: 264, Interface: ge-0/0/7.0, Started: 2020/05/15 08:04:51.870
Link ID: 261, Interface: ge-0/0/4.0, Started: 2020/05/15 08:04:50.944
Link ID: 262, Interface: ge-0/0/5.0, Started: 2020/05/15 08:04:51.020
Link ID: 263, Interface: ge-0/0/6.0, Started: 2020/05/15 08:04:51.096

jcluser@vMX-A6>
```

Then, find the time when the node level changed using the `show rift zero-touch-provisioning statistics` command:

```
jcluser@vMX-A6> show rift zero-touch-provisioning statistics
Current Level: 22, Last Change: 2020/05/15 08:04:50.968, Total Changes: 1
HAL Offers: 002c6bf56118c000, 002c6bf5e4f2c000
Last HAL Offer On: 2020/05/15 08:04:50.968

jcluser@vMX-A6>
```

The difference between the two values gives you the convergence time. In this case, 08:04:50.968 - 08:04:50.587 = 381ms.

4.1.3.1 Determining the Topology

Each RIFT node has a topological view of the connected nodes and also the nodes underneath the topology. As an example, a leaf node has no southbound neighbors. This next output contains large amounts of information condensed into a simple view: the number of adjacencies with their properties, mis-cablings, address families, and the amount of prefixes inserted by each node are all easily visible, as is the last TIE originated, a good indication of last change on the node. Moreover, a node displayed without a reachable direction (except at the same level) is a good indication of a fabric anomaly preventing reachability.

The `show rift topology nodes` command provides you with a topological view from the node (a leaf in this case):

```
jcluser@vMX-A6> show rift topology nodes

                                     +------ Links ------+--- TIEs ----+- Prefixs -+
Lvl Name    Originator     Ovld Dir|3way|Mscb|Sec |BFD | Auth | Non  | V4  | V6  |Newest TIE Issued
---+----------+-----------------+----+---+----+----+----+----+------+------+-----+-----+------------
 23 vMX-A4  002c6bf56118c000    N   4           4          4  1  1 2020/05/15 08:05:11
 23 vMX-A3  002c6bf5e4f2c000    N   4           4          4  1  1 2020/05/15 08:08:58
 22 vMX-A6  002c6bf5f32fc000    N   2           2          4          2020/05/15 08:05:52

jcluser@vMX-A6>
```

While a RIFT node sees connected and underneath nodes in the topology, only the ToF nodes have the full view:

```
jcluser@vMX-A1> show rift topology nodes

                                     +------ Links ------+--- TIEs ----+- Prefixs -+
Lvl Name    Originator     Ovld Dir|3way|Mscb|Sec |BFD | Auth | Non  | V4  | V6  |Newest TIE Issued
---+----------+-----------------+----+---+----+----+----+----+------+------+-----+-----+------------
 24 vMX-A1  002c6bf51110c000    N   5           5          6  1  1 2020/05/15 08:07:11
 24 vMX-A2  002c6bf5bf8cc000         5           5          2          2020/05/15 08:05:17
 23 vMX-A13 002c6bf502d1c000    S   2           2          1          2020/05/15 08:05:10
 23 vMX-A12 002c6bf51ac0c000    S   4           4          1          2020/05/15 08:06:00
 23 vMX-A4  002c6bf56118c000    S   4           4          2          2020/05/15 08:05:11
 23 vMX-A5  002c6bf5a471c000    S   4           4          1          2020/05/15 08:05:41
 23 vMX-A3  002c6bf5e4f2c000    S   4           4          2          2020/05/15 08:08:58
 22 vMX-A7  002c6bf582c9c000    S   2           2          2          2020/05/15 08:05:08
 22 vMX-A9  002c6bf5b5ecc000    S   2           2          3        1 2020/05/15 08:04:59
 22 vMX-A6  002c6bf5f32fc000    S   2           2          2          2020/05/15 08:05:52
 22 vMX-A8  002c6bf5f9cbc000    S   2           2          3        1 2020/05/15 08:05:08

jcluser@vMX-A1>
```

4.1.4 Authentication Verification

4.1.4.1 Interface Protection Verification (Outer Key)

The `show rift interface status` command provides you with the interface outer key configured to authenticate the LIEs (Hellos) required (when configured) to form the `ThreeWay` adjacency:

```
jcluser@vMX-A9> show rift interface status
Link ID: 257, Interface: ge-0/0/0.0
Status Admin True, Platform True, BFD True, State: ThreeWay, 3-Way Uptime: 2 minutes, 32 seconds
LIE TX V4: 224.0.0.120, LIE TX V6: ff02::a1f7, LIE TX Port: 914, TIE RX Port: 915
PoD 0, Nonce 14413
Neighbor: ID 002c6bf59ad2c000, Link ID 257, Name: vMX-A5:ge-0/0/0.0, Level: 23
 TIE V6: fe80::250:56ff:fea2:79c5, TIE Port: 915, BW 1000 MBits/s
PoD: None, Nonce: 17984, Outer Key: 12, Holdtime: 3 secs
```

The outer key number is the key-id of the key configured for the LIEs authentication.

When the key-id displayed is zero (0), it means no authentication and no interface adjacency protection:

```
Link ID: 260, Interface: ge-0/0/3.0
Status Admin True, Platform True, BFD True, State: ThreeWay, 3-Way Uptime: 2 minutes, 32 seconds
LIE TX V4: 224.0.0.120, LIE TX V6: ff02::a1f7, LIE TX Port: 914, TIE RX Port: 915
PoD 0, Nonce 23848
Neighbor: ID 002c6bf55001c000, Link ID 260, Name: vMX-A12:ge-0/0/3.0, Level: 23
 TIE V6: fe80::250:56ff:fea2:df4c, TIE Port: 915, BW 1000 MBits/s
PoD: None, Nonce: 32009, Outer Key: 0, Holdtime: 3 secs
...<output omitted for brevity>...
```

The nodes topology provides also a summary of the authentication, here with vMX-A5 and Leaf09 have both 1 link protected in common.

```
jcluser@vMX-A9> show rift topology nodes

                                     +------ Links ------+--- TIEs ----+- Prefixs -+
Lvl Name    Originator     Ovld Dir|3way|Mscb|Sec |BFD | Auth | Non  | V4  | V6  |Newest TIE Issued
---+----------+-----------------+----+---+----+----+----+----+------+------+-----+-----+-----------
 23 vMX-A12 002c6bf55001c000    N   4         4   6         1  1 2020/05/29 21:07:17
 23 vMX-A5  002c6bf59ad2c000    N   4      1 4   4         1  1 2020/05/29 21:07:11
  0 leaf09  0000000000000013    N   2      1 2   2   1        1 2020/05/29 21:08:01

jcluser@vMX-A9>
```

4.1.4.2 TIE Origin Protection Verification

The `show rift database content` command contains the TIEs and their associated authentication key-id:

```
jcluser@vMX-A9> show rift database content
```

```
Dir Originator   Type  ID       SeqNr
Lifetime   Origin Creation Time   Origin   Content   Key ID
---+----------------+---------+---------+----------------+--------+---------------------+-Lifetime-+-
-Size-+--------
S   0000000000000013 Node   10000000  5ed1797184aa   604301 2020/05/29 21:08:01  604800          None
S   002c6bf55001c000 Node   10000000  5ed17943c7ae   604301 2020/05/29 21:07:17  604800  282  8453
S   002c6bf55001c000 Node   10000002  5ed17944c83e   604302 2020/05/29 21:07:17  604800  404  8453
S   002c6bf55001c000 Prefix 2000004f  5ed1794e59f8   604246 2020/05/29 21:06:22  604800  203  8453
S   002c6bf55001c000 Prefix 20000071  5ed1794e3349   604246 2020/05/29 21:06:22  604800  203  8453
S   002c6bf59ad2c000 Node   10000000  5ed1793dc56c   604290 2020/05/29 21:07:00  604800  281  8453
S   002c6bf59ad2c000 Node   10000002  5ed1793e5d57   604301 2020/05/29 21:07:11  604800  407  8453
S   002c6bf59ad2c000 Prefix 20000018  5ed17949f539   604247 2020/05/29 21:06:17  604800  203  8453
S   002c6bf59ad2c000 Prefix 20000026  5ed17949d18f   604247 2020/05/29 21:06:17  604800  203  8453
N   0000000000000013 Node   10000000  5ed1797170f1   604301 2020/05/29 21:08:01  604800  353  8453
N   0000000000000013 External  6000005e  5ed1796f87d7   604235 2020/05/29 21:06:55  604800  223  8453

jcluser@vMX-A9>
```

As you can see, each TIE is authenticated using the key-id 8453.

The node's topology also provides a summary of the authentication, as in the example below, the `Auth` column of the TIEs section provides the number of TIEs authenticated (and non-authenticated)

```
jcluser@vMX-A9> show rift topology nodes

                                     +------ Links ------+--- TIEs ----+- Prefixs -+
Lvl Name    Originator     Ovld Dir|3way|Mscb|Sec |BFD | Auth | Non  | V4  | V6  |Newest TIE Issued
---+----------+----------------+----+---+----+----+----+----+------+-------+-----+-----+------------
 23 vMX-A12 002c6bf55001c000  N   4                4   4            1    1 2020/05/29 21:07:17
 23 vMX-A5  002c6bf59ad2c000  N   4      1         4   4        1    1 2020/05/29 21:07:11
  0 leaf09  0000000000000013  N   2      1         2   2          1    1 2020/05/29 21:08:01

jcluser@vMX-A9>
```

You can see that, as an example, vMX-A5 sent four TIEs with authentication.

4.1.5 RIFT Interface Monitoring

A RIFT interface can be monitored by looking at its statistics. The `show rift interface statistics` command provides the statistics for the interfaces:

```
jcluser@vMX-A9> show rift interface statistics
Link ID: 257, Interface: ge-0/0/0.0, Started: 2020/06/23 15:30:22.184
TIES RX: 19, TX: 13, REQ: 1, Neighbor REQ: 6
Same TIE RX: 2
TIDES TX: 2477, RX: 2462, TIRES TX: 17, TIRES RX: 10
Pkt Rate/100msecs Highest: 51, Current: 50, Packet Sequence Losses: 36
Last TIE RX: 002c6bf5e136c000/S/Node____/10000002, Last TIE RX on: 2020/06/24 18:11:06.024
Last Newer TIE RX: 002c6bf5e136c000/S/Node____/10000002, Last Newer TIE RX on: 2020/06/24 18:11:06.019
Last TIE TX: 002c6bf5e90bc000/S/Node____/10000002, Last TIE TX on: 2020/06/23 15:31:52.963
TIE TX Queue Len: 0, Last TIE TX Queued: 002c6bf534bbc000/N/External/60000015
Last TIE REQ'ed: 002c6bf5e136c000/S/
Node____/10000002, Last TIE REQ on: 2020/06/23 15:30:48.415, Reason: UnknownOnTIDE
Largest TX'ed: TIE: 441, (002c6bf5e90bc000/S/Node____/10000002), TIDE: 546, TIRE: 120
```

```
Three-Way UP 1, DOWN 0, Last UP 2020/06/23 15:30:48.400
Last Reason DOWN: None
LIE TX 150636, RX 148557
Last LIE TX 2020/06/25 09:35:12.119, RX 2020/06/25 09:35:11.830, Reject Reason: RemoteUndefinedLevel
Current Level Self 22, Neighbor 23, Level Changes Self 1, Neighbor 2
Flood Leader: False, Changes: 2, Last Change: 2020/06/23 15:30:48.400
```

The interesting information highlighted here gives good information on a RIFT interface stability:

- `Three-Way UP`: Number of times adjacency came up
- `DOWN`: Number of times adjacency came down
- `Last UP`: Date and time the last adjacency came up
- Or, `Last DOWN`: Date and time the last adjacency went down (if the interface is in state Three-Way DOWN)
- `Last Reason DOWN`: Reason why the last adjacency went down (for example, `HoldtimeExpired`)

From a performance perspective, by monitoring how long it took for the interface to establish a RIFT adjacency, you can compare the time the interface started (Started: 2020/06/23 15:30:22.184 in the previous example) with the time the RIFT interface was `Three-Way UP` (Last UP: 2020/06/23 15:30:48.400), which is a difference of 26s 216ms.

4.1.6 Flood Reduction Monitoring

Flood reduction is a key advantage of RIFT specifically designed from the beginning for Fat Tree topologies. As RIFT TIEs (link-state information) are not propagated southbound (except for the node TIE one-level southbound reflection), RIFT flood reduction focuses on RIFT TIEs flooding in the north direction.

Depending on the size of the DC fabric (number of levels, number of nodes per level, and number of links between nodes), without flood reduction the number of TIEs coming from a leaf node toward the ToF nodes would be proportionally duplicated to the size of the DC fabric.

As mentioned, the RIFT protocol takes this into consideration from the beginning and automatically minimizes the amount of flooding. With flood reduction, the nodes reflood the TIEs only if they are flood leader on the incoming link.

By default, two or more flood leaders per level and per branch are elected (by algorithm calculation) to ensure minimum double coverage by the flooding graph. In the RIFT protocol specifications, this is named the redundancy factor R (= 2 by default). Also, flood leaders are elected only up to two stages below the ToF nodes, as there is nothing to optimize northbound of these nodes. It means that in a 5-stage Clos (so, three levels), only the leaf nodes will elect the flood leaders, but not the spines nor the ToF nodes since it isn't necessary. In a 7-stage fabric both the leafs and the spines above them would choose flood leaders

To determine if a RIFT ingress interface is elected as a flood leader, the information can be found using the `show rift interface statistics | match "interface|Leader"` command.

Remember that the flood leaders are responsible for flooding northbound TIEs coming from a southbound ingress interface further north.

```
jcluser@vMX-A3> show rift interface status | match "name"
Neighbor: ID 002c6bf5f0a5c000, Link ID 257, Name: vMX-A1:ge-0/0/0.0, Level: 24
Neighbor: ID 002c6bf55e76c000, Link ID 258, Name: vMX-A6:ge-0/0/1.0, Level: 22
Neighbor: ID 002c6bf56bcfc000, Link ID 259, Name: vMX-A7:ge-0/0/2.0, Level: 22
Neighbor: ID 002c6bf50be5c000, Link ID 264, Name: vMX-A2:ge-0/0/7.0, Level: 24

jcluser@vMX-A3> show rift interface statistics | match "interface|Leader"
Link ID: 257, Interface: ge-0/0/0.0, Started: 2020/06/23 15:26:06.865
Flood Leader: False, Changes: 2, Last Change: 2020/06/23 15:26:22.470
Link ID: 258, Interface: ge-0/0/1.0, Started: 2020/06/23 15:26:06.989
Flood Leader: True, Changes: 3, Last Change:  2020/06/23 15:26:11.887
Link ID: 259, Interface: ge-0/0/2.0, Started: 2020/06/23 15:26:07.102
Flood Leader: True, Changes: 7, Last Change:  2020/06/25 06:52:29.360
Link ID: 260, Interface: ge-0/0/3.0, Started: 2020/06/23 15:26:07.217
Link ID: 264, Interface: ge-0/0/7.0, Started: 2020/06/23 15:26:07.601
Flood Leader: False, Changes: 1, Last Change: 2020/06/23 15:26:09.064
Link ID: 261, Interface: ge-0/0/4.0, Started: 2020/06/23 15:26:07.293
Link ID: 262, Interface: ge-0/0/5.0, Started: 2020/06/23 15:26:07.445
Link ID: 263, Interface: ge-0/0/6.0, Started: 2020/06/23 15:26:07.523
jcluser@vMX-A3>

jcluser@vMX-A4> show rift interface status | match "name"
Neighbor: ID 002c6bf56bcfc000, Link ID 257, Name: vMX-A7:ge-0/0/0.0, Level: 22
Neighbor: ID 002c6bf5f0a5c000, Link ID 258, Name: vMX-A1:ge-0/0/1.0, Level: 24
Neighbor: ID 002c6bf55e76c000, Link ID 259, Name: vMX-A6:ge-0/0/2.0, Level: 22
Neighbor: ID 002c6bf50be5c000, Link ID 263, Name: vMX-A2:ge-0/0/6.0, Level: 24

jcluser@vMX-A4> show rift interface statistics | match "interface|Leader"
Link ID: 257, Interface: ge-0/0/0.0, Started: 2020/06/23 15:26:31.666
Flood Leader: True, Changes: 7, Last Change:  2020/06/25 06:52:51.160
Link ID: 258, Interface: ge-0/0/1.0, Started: 2020/06/23 15:26:31.753
Flood Leader: False, Changes: 2, Last Change: 2020/06/23 15:26:45.011
Link ID: 259, Interface: ge-0/0/2.0, Started: 2020/06/23 15:26:31.830
Flood Leader: True, Changes: 3, Last Change:  2020/06/23 15:26:34.216
Link ID: 260, Interface: ge-0/0/3.0, Started: 2020/06/23 15:26:31.907
Link ID: 264, Interface: ge-0/0/7.0, Started: 2020/06/23 15:26:32.360
Link ID: 261, Interface: ge-0/0/4.0, Started: 2020/06/23 15:26:32.135
Link ID: 262, Interface: ge-0/0/5.0, Started: 2020/06/23 15:26:32.210
Link ID: 263, Interface: ge-0/0/6.0, Started: 2020/06/23 15:26:32.286
Flood Leader: False, Changes: 1, Last Change: 2020/06/23 15:26:33.045
jcluser@vMX-A4>
```

```
jcluser@vMX-A5> show rift interface status | match "name"
Neighbor: ID 002c6bf534bbc000, Link ID 257, Name: vMX-A9:ge-0/0/0.0, Level: 22
Neighbor: ID 002c6bf59d29c000, Link ID 258, Name: vMX-A8:ge-0/0/1.0, Level: 22
Neighbor: ID 002c6bf5f0a5c000, Link ID 260, Name: vMX-A1:ge-0/0/3.0, Level: 24
Neighbor: ID 002c6bf50be5c000, Link ID 262, Name: vMX-A2:ge-0/0/5.0, Level: 24

jcluser@vMX-A5> show rift interface statistics | match "interface|Leader"
Link ID: 257, Interface: ge-0/0/0.0, Started: 2020/06/23 15:27:41.332
Flood Leader: True, Changes: 5, Last Change:  2020/06/24 18:07:59.010
Link ID: 258, Interface: ge-0/0/1.0, Started: 2020/06/23 15:27:41.456
Flood Leader: True, Changes: 3, Last Change: 2 020/06/23 15:27:42.377
Link ID: 259, Interface: ge-0/0/2.0, Started: 2020/06/23 15:27:41.573
Link ID: 260, Interface: ge-0/0/3.0, Started: 2020/06/23 15:27:41.649
Flood Leader: False, Changes: 1, Last Change: 2020/06/23 15:27:42.806
Link ID: 264, Interface: ge-0/0/7.0, Started: 2020/06/23 15:27:42.617
Link ID: 261, Interface: ge-0/0/4.0, Started: 2020/06/23 15:27:41.803
Link ID: 262, Interface: ge-0/0/5.0, Started: 2020/06/23 15:27:41.880
Flood Leader: False, Changes: 2, Last Change: 2020/06/23 15:27:42.380
Link ID: 263, Interface: ge-0/0/6.0, Started: 2020/06/23 15:27:41.957
jcluser@vMX-A5>

jcluser@vMX-A12> show rift interface status | match "name"
Neighbor: ID 002c6bf50be5c000, Link ID 257, Name: vMX-A2:ge-0/0/0.0, Level: 24
Neighbor: ID 002c6bf5f0a5c000, Link ID 259, Name: vMX-A1:ge-0/0/2.0, Level: 24
Neighbor: ID 002c6bf534bbc000, Link ID 260, Name: vMX-A9:ge-0/0/3.0, Level: 22
Neighbor: ID 002c6bf59d29c000, Link ID 261, Name: vMX-A8:ge-0/0/4.0, Level: 22

jcluser@vMX-A12> show rift interface statistics | match "interface|Leader"
Link ID: 257, Interface: ge-0/0/0.0, Started: 2020/06/23 15:28:07.704
Flood Leader: False, Changes: 1, Last Change: 2020/06/23 15:28:09.587
Link ID: 258, Interface: ge-0/0/1.0, Started: 2020/06/23 15:28:07.791
Link ID: 259, Interface: ge-0/0/2.0, Started: 2020/06/23 15:28:07.942
Flood Leader: False, Changes: 2, Last Change: 2020/06/23 15:28:21.744
Link ID: 260, Interface: ge-0/0/3.0, Started: 2020/06/23 15:28:08.057
Flood Leader: True, Changes: 7, Last Change:  2020/06/24 18:08:38.178
Link ID: 264, Interface: ge-0/0/7.0, Started: 2020/06/23 15:28:08.405
Link ID: 261, Interface: ge-0/0/4.0, Started: 2020/06/23 15:28:08.172
Flood Leader: True, Changes: 3, Last Change:  2020/06/23 15:28:22.766
Link ID: 262, Interface: ge-0/0/5.0, Started: 2020/06/23 15:28:08.250
Link ID: 263, Interface: ge-0/0/6.0, Started: 2020/06/23 15:28:08.327
jcluser@vMX-A12>
```

You can see that there are only two aggregation nodes per branch in our JCL lab and they are all elected as flood leader for the southbound ingress interfaces (towards leaf nodes).

To view stats on flood leader election, like the number of runs, the number of computations held down, the chosen flood leaders, and the time of last election, use the following show command:

```
jcluser@vMX-A6> show rift flood-reduction statistics

Runs  Holds Leaders                                                    Last On
-----+-----+----------------------------------------------------------------+---------------------
 2567  3015                    002c6bf54f85f600, 002c6bf5d99dda00  2020/10/21 21:55:55.611
```

As you can see now, leaf node vMX-A6 has selected two direct northbound nodes (vMX-A3 and vMX-A4) as flood leaders.

As flood leader election and computation is not periodic but purely event-driven triggered by node TIE changes, the number of computations associated with the node uptime gives another indication on the status of IP fabric stability.

The number of computations held down (`Holds`) means that a computation was not performed since it was too soon after the last one. This number can indicate some flapping or multi-node issue.

And stats per interface like this:

```
jcluser@vMX-A6> show rift interface statistics | match flood
Flood Leader: False, Changes: 245, Last Change: 2020/10/19 22:26:28.842
Flood Leader: False, Changes: 33, Last Change: 2020/10/20 06:36:54.692
Flood Leader: True, Changes: 69, Last Change: 2020/10/19 22:27:14.944

jcluser@vMX-A6>
```

4.1.7 Overload Bit

RIFT supports the overload bit that takes the node out of traffic bearing paths. When it is set, the node can be accessed only to reach its direct IP addresses (for example SSH access) but cannot be anymore used as a transit node.

For better stability, the overload bit is automatically set at RIFT protocol startup, to leave the time to the protocol to converge before actually forwarding traffic. The time the overload bit is enabled, depends on the RIFT database stability in combination with an hold timer (10..1800 seconds).

By looking at the `riftd.log` log file you can see when the overload bit has been unset at RIFT startup:

```
jcluser@vMX-A1> show log riftd
Oct 23 08:00:49.919 INFO changed tracing configuration [], pid: 11922
Oct 23 08:00:49.927 INFO
...
Oct 23 08:01:04.165 INFO adjacency 3-way up (bfd) vMX-A4:ge-0/0/1.0 level Some(23) on ValidReflection,
subsystem: lies, peer: ge-0/0/1.0, nodename: vMX-A1, pid: 11922
Oct 23 08:01:04.552 INFO adjacency 3-way up (bfd) vMX-A5:ge-0/0/3.0 level Some(23) on ValidReflection,
subsystem: lies, peer: ge-0/0/3.0, nodename: vMX-A1, pid: 11922
Oct 23 08:01:04.677 INFO adjacency 3-way up (bfd) vMX-A3:ge-0/0/0.0 level Some(23) on ValidReflection,
subsystem: lies, peer: ge-0/0/0.0, nodename: vMX-A1, pid: 11922
Oct 23 08:01:09.494 INFO Database stable, removing overload, subsystem: lsdb, nodename: vMX-A1, pid:
11922
...
```

For maintenance purposes, a RIFT node can be also removed from the IP fabric data plane by manually setting the overload bit. This next example shows how to enable the `overload` bit:

```
jcluser@vMX-A4# set protocols rift overload ?
Possible completions:
  <[Enter]>          Execute this command
+ apply-groups       Groups from which to inherit configuration data
+ apply-groups-except  Don't inherit configuration data from these groups
  timeout            Time in seconds after which overload bit is reset (10..1800 seconds)
  |                  Pipe through a command
[edit]
jcluser@vMX-A4# set protocols rift overload

[edit]
jcluser@vMX-A4# show | compare
[edit protocols rift]
+ overload;

[edit]
jcluser@vMX-A4#
```

As you can see in the output, it is also possible to configure the overload bit timeout (10..1800 seconds) to be used at RIFT startup (for example, if you have a very big IP fabric and need longer delays or want extra time for your management system). Be aware that, with the tested RIFT version, setting the overload bit timeout will set the overload bit directly and for the time of the timeout as well, so it has an impact when configured.

Once the overload bit is set manually, it's shown in its node status:

```
jcluser@vMX-A4> show rift node status
System Name: vMX-A4, System ID: 002c6bf5be56c000
Level: 23, RIFT Encoding Major: 4, Minor: 0
Flags: overload=True
Capabilities: flood-reduction: True
LIE v4 RX: 224.0.0.120, LIE v6 RX: ff02::a1f7, LIE RX Port: 914
 Re-Connections: 0
Peers: 8, 3-way: 4, South: 2, North: 2
```

And also propagated via its node TIE to the fabric north of it (up to the ToF) but also one level south of it (southbound reflection):

```
jcluser@vMX-A4> show rift tie 002c6bf5be56c000/n/node/10000000
TIE ID: 002c6bf5be56c000/N/Node____/10000000
Name: vMX-A4, Level: 23
Capabilities: protocol_minor_version=0, flood_reduction=True, Flags: overload=True
...
```

The overload status of the node is now propagated up to the top of the IP fabric:

```
jcluser@vMX-A1> show rift topology nodes

.                               +------ Links ------+--- TIEs ----+- Prefixs -+
Lvl Name   Originator    Ovld Dir|3way|Mscb|Sec |BFD | Auth | Non  | V4  | V6  |Newest TIE Issued
---+----------+-----------------+----+---+----+----+----+----+------+------+-----+-----+----------
 24 vMX-A2  002c6bf5369dc000          5           5          2          2020/10/23 07:59:34
 24 vMX-A1  002c6bf55073c000      N   4           4          6     1  1 2020/10/23 08:02:12
 23 vMX-A3  002c6bf53e19c000      S   4           4          2          2020/10/23 07:59:24
 23 vMX-A12 002c6bf5a702c000      S   5           5          2          2020/10/23 07:59:32
 23 vMX-A4  002c6bf5be56c000  yes S   4           4          2          2020/10/23 09:05:00
```

4.1.8 Other Interesting Monitoring Commands

Let's quickly run through this list of other interesting commands that can be used to monitor and troubleshoot RIFT:

```
show route protocol rift
show route protocol rift table inet6.0
show route protocol rift table inet.0
show rift routes next-hops
show rift route statistics
show route protocol rift extensive display-client-data
```

A RIFT protocol routing table information can be checked for all available RIFT routes.

The `show route protocol rift` command displays all route information of the RIFT protocol. In the next output RIFT IPv4 and IPv6 default routes are installed in the routing table on leaf:

```
jcluser@vMX-A9> show route protocol rift

inet.0: 4 destinations, 5 routes (4 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

0.0.0.0/0           [Static/20/100] 1d 06:57:44, metric2 0
                    >  to fe80::250:56ff:fea2:796 via ge-0/0/3.0
                       to fe80::250:56ff:fea2:4019 via ge-0/0/0.0
224.0.0.120/32     *[RIFT/20/100] 1d 12:52:07
                       MultiRecv

inet6.0: 13 destinations, 13 routes (13 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

::/0               *[Static/20/100] 1d 06:57:44, metric2 0
                    >  to fe80::250:56ff:fea2:796 via ge-0/0/3.0
                       to fe80::250:56ff:fea2:4019 via ge-0/0/0.0
ff02::a1f7/128     *[RIFT/20/100] 1d 12:52:07
                       MultiRecv
```

The `show route protocol rift table inet.0` command displays IPv4 route information of the RIFT protocol. And `show route protocol rift 0/0 exact detail` displays net-hop, bandwidth % per next-hop link, and protocol next-hop information:

```
jcluser@vMX-A9> show route protocol rift table inet.0

inet.0: 4 destinations, 5 routes (4 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

0.0.0.0/0           [Static/20/100] 1d 07:04:18, metric2 0
                    >  to fe80::250:56ff:fea2:796 via ge-0/0/3.0
                       to fe80::250:56ff:fea2:4019 via ge-0/0/0.0
224.0.0.120/32     *[RIFT/20/100] 1d 12:58:41
                       MultiRecv

jcluser@vMX-A9>

jcluser@vMX-A9> show route protocol rift 0/0 exact detail

inet.0: 4 destinations, 5 routes (4 active, 0 holddown, 0 hidden)
0.0.0.0/0 (2 entries, 1 announced)
         Static Preference: 20/100
                Next hop type: Indirect, Next hop index: 0
                Address: 0xc7421a8
                Next-hop reference count: 1
                Next hop type: Router, Next hop index: 0
                Next hop: fe80::250:56ff:fea2:796 via ge-0/0/3.0 weight 0x1 balance 57%, selected
                Session Id: 0x0
                Next hop: fe80::250:56ff:fea2:4019 via ge-0/0/0.0 weight 0x1 balance 43%
                Session Id: 0x0
                Protocol next hop: fe80:100::1:0:8b14
                Indirect next hop: 0xc6bde84 - INH Session ID: 0x0 Weight 0x1
                State: <Int Changed NSR-incapable Programmed>
                Inactive reason: Route Preference
                Age: 1d 7:55:37         Metric2: 0
                Validation State: unverified
                AS path: I

jcluser@vMX-A9>
```

The `show route protocol rift table inet6.0` command displays IPv6 route information of the RIFT protocol:

```
jcluser@vMX-A9> show route protocol rift table inet6.0

inet6.0: 13 destinations, 13 routes (13 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

::/0               *[Static/20/100] 1d 07:02:27, metric2 0
                    >  to fe80::250:56ff:fea2:796 via ge-0/0/3.0
                       to fe80::250:56ff:fea2:4019 via ge-0/0/0.0
ff02::a1f7/128     *[RIFT/20/100] 1d 12:56:50
                       MultiRecv

jcluser@vMX-A9>
```

The `show rift routes next-hops` command display next-hop information of the RIFT protocol. The output field from this command includes the ID of RIFT next-hop, System ID of the adjacent node, and Link IDs leading to the adjacent node:

```
jcluser@vMX-A9> show rift routes next-hops

Nexthop  SystemID        Links
---------+-----------------+-----------------
80004585  002c6bf53e71c000  260 (ge-0/0/3.0)
            002c6bf56487c000  257 (ge-0/0/0.0)
80004588  002c6bf53e71c000  260 (ge-0/0/3.0)
8000458a  002c6bf53e71c000  260 (ge-0/0/3.0)
          002c6bf56487c000  257 (ge-0/0/0.0)
8000458b  002c6bf53e71c000  260 (ge-0/0/3.0)
8000458f  002c6bf53e71c000  260 (ge-0/0/3.0)
8000bf6f  002c6bf58579c000  {}

jcluser@vMX-A9>

jcluser@vMX-A9> show rift routes content

Prefix                          Active Metric N-Hop    All Present
-------------------------------+------+------+--------+-------------------
0.0.0.0/0                       S           2 8000458a S
::/0                            S           2 8000458a S

jcluser@vMX-A9>
```

The next useful command, the **`show rift routes statistics`** command, displays the routing table statistics of the RIFT protocol. The output field from this command includes

- Nhops: Current number of next hops, number of next hops deleted, number of next hops added or changed.
- Last route transaction: Timestamp for last route transaction, number of next hop differentials downloaded, number of route differentials downloaded and route type.
- AF: Per address family number of prefixes, number of deletes performed over the lifetime, number of additions or changes performed over the lifetime.

```
jcluser@vMX-A9> show rift routes statistics

Nhops Deletes  Adds/Changes
-----+--------+------------
    6        0            5

Last Transaction          Nhop Diff# Route Diff# Route Type
-------------------------+----------+-----------+-----------------------
2020/10/30 06:45:52.272                        2                    S
```

```
AF   Prefixes Deletes Adds/Changes
----+--------+-------+------------
IPv4        1       1            7
IPv6        1       4           16

jcluser@vMX-A9>
```

To get the RIFT installed route with all the involved details use the `show route protocol rift extensive display-client-data` command:

```
jcluser@vMX-A9> show route protocol rift extensive display-client-data

inet.0: 4 destinations, 5 routes (4 active, 0 holddown, 0 hidden)
0.0.0.0/0 (2 entries, 1 announced)
TSI:
KRT in-kernel 0.0.0.0/0 -> {100.123.0.1}
         Static Preference: 20/100
                Next hop type: Indirect, Next hop index: 0
                Address: 0xc7421a8
                Next-hop reference count: 1
                Next hop type: Router, Next hop index: 0
                Next hop: fe80::250:56ff:fea2:796 via ge-0/0/3.0 weight 0x1 balance 57%, selected
                Session Id: 0x0
                Next hop: fe80::250:56ff:fea2:4019 via ge-0/0/0.0 weight 0x1 balance 43%
                Session Id: 0x0
                Protocol next hop: fe80:100::1:0:8b14
                Indirect next hop: 0xc6bde84 - INH Session ID: 0x0 Weight 0x1
                State: <Int Changed NSR-incapable Programmed>
                Inactive reason: Route Preference
                Age: 1d 8:01:36         Metric2: 0
                Validation State: unverified
             AS path: I
             Client id: 2c6bf58579c000_11889_2147483675_N, Cookie: 11670975044300242946, Protocol:
RIFT
             Indirect next hops: 1
                        Protocol next hop: fe80:100::1:0:8b14
                        Indirect next hop: 0xc6bde84 - INH Session ID: 0x0 Weight 0x1
                        Indirect path forwarding next hops: 2
                                Next hop type: Router
                                Next hop: fe80::250:56ff:fea2:796 via ge-
0/0/3.0 weight 0x1 balance 57%
                                Session Id: 0x0
                                Next hop: fe80::250:56ff:fea2:4019 via ge-
0/0/0.0 weight 0x1 balance 43%
                                Session Id: 0x0
                                fe80:100::1:0:8b14/128 Originating RIB: inet6.3
                                  Node path count: 1
                                  Forwarding nexthops: 2
                                        Nexthop: fe80::250:56ff:fea2:796 via ge-0/0/3.0
                                        Session Id: 0
                                        Nexthop: fe80::250:56ff:fea2:4019 via ge-0/0/0.0
                                        Session Id: 0
```

4.1.9 Positive Disaggregation

This section illustrates how south-reflection triggers the self-healing upon link-failure via positive disaggregation.

With routing in the underlay two extreme options exist:

- Full visibility: When each node has full visibility, then for any kind of destination always the shortest path can be taken. This is achieved with OSPF/IS-IS single area/level design. However with larger scale, the burden of maintaining hundreds or even thousands of nodes requires a rather powerful control and forwarding-plane. Due to the scaling-implications of an IGP very often BGP is seen in the underlay.
- Default-route: When receiving just default-routes, the RIB/FIB stays at minimum, hence the burden of scale is very low. But with only receiving only default-routes, it is not any longer for any node to understand which of its default-routes finally provide the shortest path.

Positive disaggregation is the tool to only advertise on top of the default route the knowledge to still use the shortest path to any destination.

Both, positive disaggregation and south-reflection will be shown with the below example.

As the topic contains lots of CLI output, the following steps will be shown:

- Fabric is healthy state and all links up
- Only default-routes received from North
- Advertising more-specific routes to North
- Showing how south-reflection enables to learn neighborship of nodes within the same POD and the same level
- Introducing a link-failure
- South-reflection indicates neighbor-shop loss of the neighboring node
- This triggers positive-disaggregation to still allow using the shortest path for ALL destinations

4.1.9.1 Fabric in healthy state case

For easier explanation, this section considers a simple case (Figure 4.4) where only RIFT is used in the IP Fabric (no overlay) and two hosts are connected to it using a /126 subnet part of the IPv6 range fc00:1000::/64.

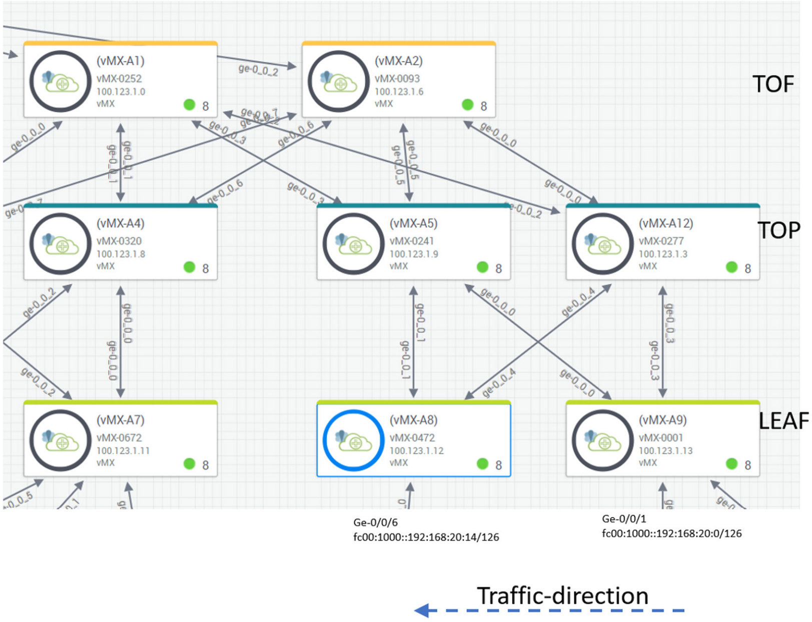


Figure 4.1 Fabric in healthy state

Let's assume traffic flow in Figure 4.1 is from LEAF vMX-A9 towards LEAF vMX-A8. When fabric is in perfect health, the following is expected:

- TOF-nodes vMX-A1 and vMX-A2 are aware of any prefix underneath.
- TOP-nodes vMX-A5 and vMX-A12 are receiving default routes from both their TOF's.
- TOP-nodes vMXA5-and vMX-A12 are also aware of any more-specific routes southwards.
- LEAF-Nodes receives default-routes vMX-A5 and vMX-A12 are only receiving IPv4 and IPv6 default-routes from the nodes northbound (TOP).
- LEAF-Nodes are aware of the routes northbound and the Direct access networks in the range fc00:1000::/64 which are exported/redistributed northbound.
- With current Junos, any received RIFT-routes are shown as [Static].

First let's examine the forwarding state in the fabric while it is in a healthy state, say route distribution as done by vMX-A8.

In this next example, prefix fc00:1000::192:168:20:14/126 of interface ge-0/0/6 gets advertised into the RIFT:

```
jcluser@vMX-A8# top show interfaces ge-0/0/6
unit 0 {
    family inet6 {
        address fc00:1000::192:168:20:15/126;
    }
}
```

It might be a good idea to have a dedicated range for any revenue interfaces (a.k.a., *access interfaces* or *user ports*) to ease its redistribution. In this example this range is fc00:1000::/64.

Here's a routing policy to match the `revenue_interfaces` range of fc00:1000::/64 with the `prefix-length-range` option used to match only on /126 routes:

```
policy-options {
replace:
    policy-statement extending_underlay {
        term revenue_interfaces {
            from {
                route-filter fc00:1000::/64 prefix-length-range /126-/126;
            }
            then accept;
        }
        term last {
            then reject;
        }
    }
}
```

Finally, those routes need to be exported into RIFT. Note the direction of `northbound`:

```
protocols {
    rift {
        export {
            northbound {
                extending_underlay;
            }
        }
    }
}
```

And know that vMX-A9 is doing the same but for prefix fc00:1000::192:168:20:0/126. RIFT-routes vMX-A8.

Using the `show rift routes content` command indicates that IPv4 and IPv6 default routes are installed by RIFT. The vMX-A8 does not require any more specific RIFT learned prefixes to reach any other destination:

```
jcluser@vMX-A8# run show rift routes content

Prefix                          Active Metric N-Hop    All Present
-------------------------------+-------+------+--------+-------------------
0.0.0.0/0                       S           2 800038c8 S
::/0                            S           2 800038c8 S
```

To get more insight into RIFT DB execute the `show rift topology nodes`" command. Right now, we're interested in finding out what system-ID vMX8 has, which is here listed in the `Originator` tab as `002c6bf5a931c000`:

```
jcluser@vMX-A8# run show rift topology nodes

.                                  +------ Links ------+--- TIEs ----+- Prefixs -+
Lvl Name      Originator      Ovld Dir|3way|Mscb|Sec |BFD | Auth | Non  | V4  | V6  |Newest TIE Issued
---+-----------+----------------+----+---+----+----+----+----+------+------+-----+-----+--------
 23 vMX-A12    002c6bf51ef8c000          3    1        3           3    1     1 2020/05/11 10:54:32
 23 vMX-A5     002c6bf525aac000          4             4          2            2020/05/08 21:57:11
 23 vMX-A5     002c6bf5c460c000      N    4            4           5    1     2 2020/05/11 10:54:07
 22 vMX-A8     002c6bf5a931c000      N    1            1           3          1 2020/05/11 10:54:06
```

By mapping the Originator to the hostname the database is more easily readable. The vMX-A8 is adding three different TIE's into the DB - one of them being `External`:

```
jcluser@vMX-A8# run show rift database content

Dir Originator       Type      ID        SeqNr         Lifetime   Origin Creation Time   Origin   Content   Key ID
---+----------------+----------+--------+----------------+--------+---------------------+-Lifetime-+--
Size-+--------
S   002c6bf51ef8c000 Node      10000002    5eb91eec3eda   599177   2020/05/11 10:54:32    604800    384     0
S   002c6bf51ef8c000 Prefix    20000051    5eb91ef34e69   595038   2020/05/11 09:46:27    604800    171     0
S   002c6bf51ef8c000 Prefix    2000006f    5eb91ef3fca5   595038   2020/05/11 09:46:27    604800    171     0
S   002c6bf525aac000 Node      10000000    5eb51f03b364   379684   2020/05/08 21:56:53    604800    249     0
S   002c6bf525aac000 Node      10000002    5eb5d56320bc   379684   2020/05/08 21:57:11    604800    371     0
S   002c6bf5a931c000 Node      10000002    5eb916c2869d   599160   2020/05/11 10:54:06    604800    None
S   002c6bf5c460c000 Node      10000000    5eb5e184ff7e   599113   2020/05/11 10:53:20    604800    249     0
S   002c6bf5c460c000 Node      10000002    5eb5e178460f   416044   2020/05/09 08:02:32    604800    371     0
S   002c6bf5c460c000 Prefix    20000001    5eb5e184accc   416044   2020/05/09 08:02:32    604800    171     0
S   002c6bf5c460c000 Prefix    2000003f    5eb5e1844b89   416044   2020/05/09 08:02:32    604800    171    0
S   002c6bf5c460c000 PosExt    7000005c    5eb928921d33   599160   2020/05/11 10:54:07    604800    183    0
N   002c6bf5a931c000 Node      10000002    5eb916c2485c   599160   2020/05/11 10:54:06    604800    249     0

N   002c6bf5a931c000 External  60000025    5eb91723c3c2   599081   2020/05/11 10:52:47    604800    191     0
```

Node vMX-A8 is populating the RIFT's DB with one external TIE. Note the direction of this TIE - (N)orthbound.

The command to see the TIE details seems complex, but after having used it a few times, it becomes more convenient:

```
jcluser@vMX-A8# run show rift tie ?
Possible completions:
  <tie>              Show RIFT TIE information for <node-hex|node-name>/<North|South>/<node|prefix|pos
itive|negative|key-value|external|ex-disaggregate>/<TIE-number-hex>
```

While today every element needs to be provided, Juniper is working on implementing wildcards for the `show rift tie` command in future releases.

We are interested into the external TIE from vMX8, which is Originator: 002c6bf5a931c000. The TIE is sent northbound, so the direction is: N. The TIE contains external prefix: `External`.

And, as there might be, many TIE's containing external prefixes, so we defined which TIE was of interest: `60000025` in this case:

```
jcluser@vMX-A8# run show rift tie 002c6bf5a931c000/N/External/60000025
TIE ID: 002c6bf5a931c000/N/External/60000025

 Prefix                          Metric LPB ATT On Link
 -------------------------------+------+---+---+-------
  fc00:1000::192:168:20:14/126        1      yes
```

As a result of northbound flooding of prefix `fc00:1000::192:168:20:14/126`, both vMX-A5 and vMX-A12 are aware of the southbound prefixes from both vMX-A8 and vMX-A9. Keep in mind that vMX-A9 is doing the same export for its connected prefix `fc00:1000::192:168:20:0/126`.

```
jcluser@vMX-A12> show rift routes content

Prefix                          Active Metric N-Hop    All Present
-------------------------------+------+------+--------+-------------------
0.0.0.0/0                       S           2 8000384e S
::/0                            S           2 8000384e S
fc00:1000::192:168:20:0/126     NExt        2 80003849 NExt
fc00:1000::192:168:20:14/126    NExt        2 80003842 NExt


jcluser@vMX-A5> show rift routes content

Prefix                          Active Metric N-Hop    All Present
-------------------------------+------+------+--------+-------------------
0.0.0.0/0                       S           2 8000e095 S
::/0                            S           2 8000e095 S
fc00:1000::192:168:20:0/126     NExt        2 8000e094 NExt
fc00:1000::192:168:20:14/126    NExt        2 8000e099 NExt
```

At this stage it is important to look into the detail of the forwarding state for vMX-A9. It looks as ifd vMX-8 and vMX-A9 are aware of the prefix they are exporting into RIFT and for anything else, the default-routes just work fine:

```
jcluser@vMX-A9# run show rift routes content

Prefix                          Active Metric N-Hop    All Present
-------------------------------+------+------+--------+-------------------
0.0.0.0/0                       S           2 80001207 S
::/0                            S           2 80001207 S
```

Assuming that load balancing is enabled on vMX-A9:

```
jcluser@vMX-A9# top set policy-options policy-statement plb then load-balance per-packet

[edit]
jcluser@vMX-A9# top set routing-options forwarding-table export plb

[edit]
jcluser@vMX-A9# commit
commit complete
```

The vMX-A9 can use the ECMP default route to reach any prefix served by vMX-A8:

```
jcluser@vMX-A9# run show route forwarding-table destination fc00:1000::192:168:20:14/126
Routing table: default.inet6
Internet6:
Destination        Type RtRef Next hop           Type Index    NhRef Netif
default            user     0                    indr  1048576     2
                                                 ulst  1048574     2
                              0:50:56:a2:2a:af   ucst      617     4 ge-0/0/0.0
                              0:50:56:a2:81:94   ucst      614     4 ge-0/0/3.0
default            perm     0                    rjct       44     5
```

NOTE With positive disaggregation in place, this would be different: a more specific route to reach prefix fc00:1000::192:168:20:14/126 would be seen.

Finally, the ToF nodes needs not only the fattest pipes but they require them to be more powerful on both the control and forwarding plane as more-specific prefixes are known there:

```
jcluser@vMX-A1> show rift routes content

Prefix                          Active Metric N-Hop    All Present
-------------------------------+------+------+--------+-------------------
0.0.0.0/0                       Disc                   Disc
::/0                            Disc                   Disc
fc00:1000::192:168:20:0/126     NExt        3 8000d10a NExt
fc00:1000::192:168:20:4/126     NExt        3 8000d135 NExt
fc00:1000::192:168:20:8/126     NExt        3 8000d135 NExt
fc00:1000::192:168:20:14/126    NExt        3 8000d10a NExt
```

NOTE The advertisement of the default route is covered in Section 2.2.5.

The `Disc` means *discard* and just means that the default route gets injected by default into RIFT by the ToF nodes. This means that any traffic destined to an IP address covered by the default route (no specific route) would be discarded.

Before introducing the link failure to trigger positive disaggregation, the south reflection should be explained, so let's look into the RIFT DB as it is seen by vMX-A5.

The vMX-A5 has two entries from vMX-A12, which is in the same PoD and the same level. While both nodes do not have a direct link, vMX-A5 does have knowledge about the two Node-TIE's from vMX-A12:

```
jcluser@vMX-A5> show rift topology nodes | match "lv|vMX-A12"
Lvl Name      Originator        Ovld Dir|3way|Mscb|Sec |BFD | Auth | Non  | V4  | V6  |Newest TIE Issued
 23 vMX-A12    002c6bf51ef8c000             4            4            2           2020/05/11 13:07:34

jcluser@vMX-A5> show rift database content | match "Dir|-|002c6bf51ef8c000"
```

```
Dir Originator       Type      ID        SeqNr         Lifetime   Origin Creation Time   Origin   Content   Key ID
---+----------------+---------+--------+----------------+--------+---------------------+-Lifetime-+--
Size-+--------
S   002c6bf51ef8c000 Node      10000000     5eb94dc54cfc   604121    2020/05/11 13:07:34    604800     250       0
S   002c6bf51ef8c000 Node      10000002     5eb91eec3edb   604040    2020/05/11 13:06:13    604800     372       0
```

This is because of south-reflection, performed by both vMX-A8 and vMX-A9. Only the NODE-TIE, which describes the adjacencies established are south-reflected. By doing so, vMX-A5 is aware of the vMX-A12 established neighborships. Figure 4.2 illustrates how vMX-A8 and vMX-A9 both perform south reflection.

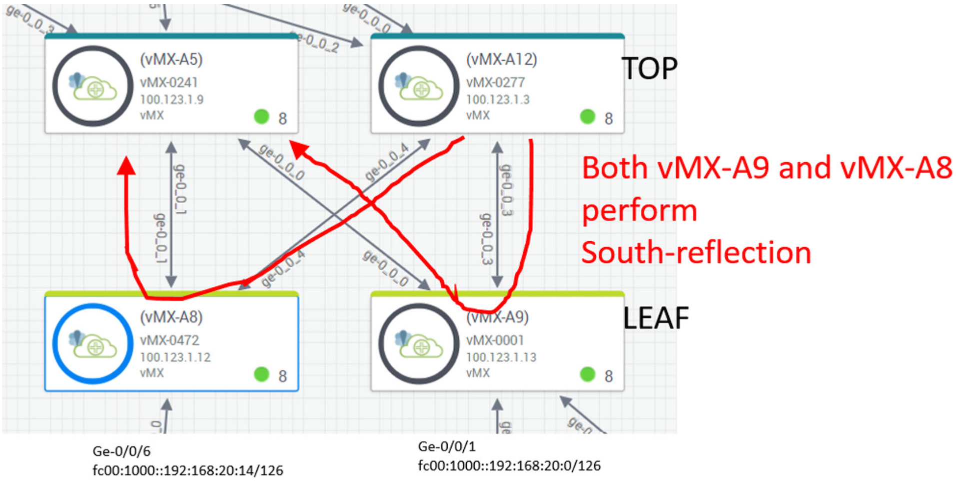


Figure 4.2 vMX-A8 and vMX-A9 Both Perform South Reflection

vMX-A5 is aware that vMX-A12 runs at the same level which enables RIFT to detect mis-cablings and perform more complex computations like determining necessary positive disaggregation. Furthermore, it is aware that vMX-A12 has the following neighbors as indicated by the next `show rift tie` command:

```
jcluser@vMX-A5> show rift tie 002c6bf51ef8c000/S/Node/10000000
TIE ID: 002c6bf51ef8c000/S/Node____/10000000
Name: vMX-A12, Level: 23
Capabilities: protocol_minor_version=0, flood_reduction=True, Flags: overload=False

 Neighbor           Lvl Cost Bandwidth
 -----------------+---+----+---------
  002c6bf5a931c000  22  1    1000

   Local ID Remote ID Intf Ndx Intf Name    Outer Key BFD
   --------+---------+--------+------------+---------+---
    261      261       336                              yes
```

```
jcluser@vMX-A5> show rift tie 002c6bf51ef8c000/S/Node/10000002
TIE ID: 002c6bf51ef8c000/S/Node____/10000002
Name: vMX-A12, Level: 23
Capabilities: protocol_minor_version=0, flood_reduction=True, Flags: overload=False

 Neighbor          Lvl Cost Bandwidth
-----------------+---+----+---------
  002c6bf59c18c000  22  1    1000

   Local ID Remote ID Intf Ndx Intf Name    Outer Key BFD
   --------+---------+--------+------------+---------+---
    260      260       335                             yes

 Neighbor          Lvl Cost Bandwidth
-----------------+---+----+---------
  002c6bf50ec2c000  24  1    1000

   Local ID Remote ID Intf Ndx Intf Name    Outer Key BFD
   --------+---------+--------+------------+---------+---
    257      257       332                             yes

 Neighbor          Lvl Cost Bandwidth
-----------------+---+----+---------
  002c6bf567cbc000  24  1    1000

   Local ID Remote ID Intf Ndx Intf Name    Outer Key BFD
   --------+---------+--------+------------+---------+---
    259      259       334                             yes
```

One may ask how the TIE's are composed and sorted, as shown in the output with one TIE containing one neighbor, while the other TIE contains three neighbors. It's the result of an intelligent hashing for efficiency and scale. Unlike OSPF or IS-IS protocols that are limited by a *dedicated* fragment or LSA in the amount of neighbors they can support (albeit there are complex protocol extensions to improve the situation), RIFT natively splits the node information across many node TIEs to allow for arbitrary numbers.

The system ID highlighted in the commands output below are related to the following nodes:

- `002c6bf5a931c000` (LEAF vMX-A8)
- `002c6bf59c18c000` (LEAF vMX-A9)
- `002c6bf50ec2c000` (TOF vMX-A2)
- `002c6bf567cbc000` (TOF vMX-A1)

4.1.9.2 Fabric in Recovery State

Now it's time to introduce a link-failure between vMX-A8 and vMX-A12 (see Figure 4.3) and look at the associated impact to RIFT.

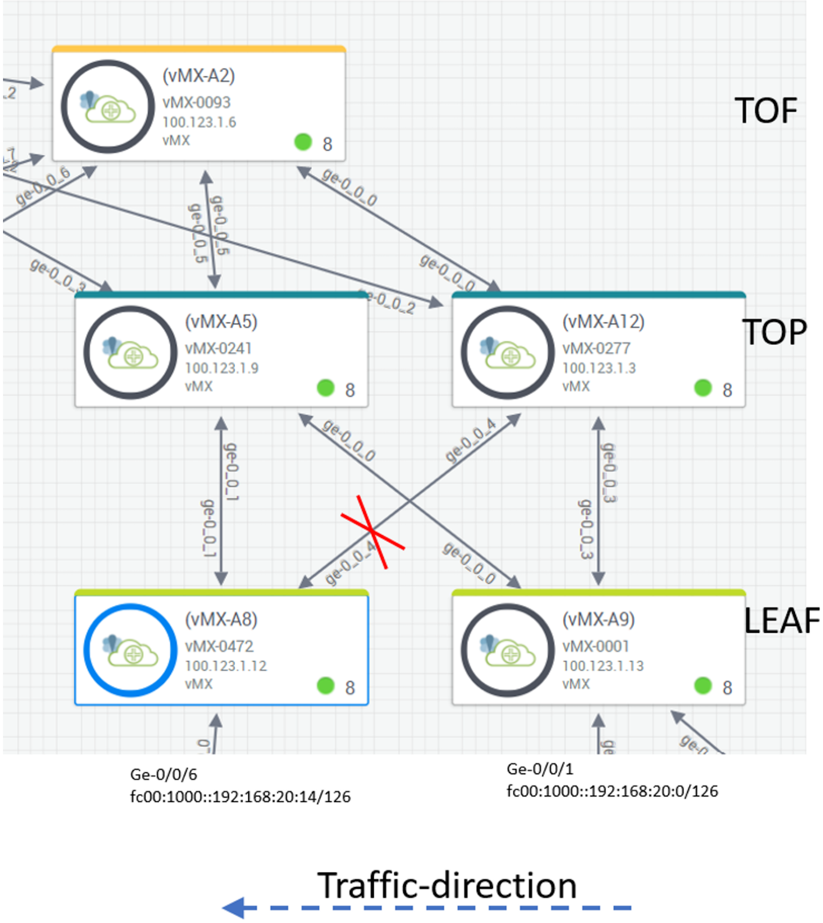


Figure 4.3 Link-failure Between vMX-A8 and vMX-A1

Via south-reflection, as soon as possible vMX-A5 is aware that vMX-A12 has lost one node. Here receiving node-ties:

```
May 11 13:34:38.779 DEBG tie: after rcvd TIE: North/12503601825038336/node/$10000002/
#R104149043791967/604800/0.000/true result ack: Some(TIE: North/12503601825038336/node/$10000002/
#R104149043791967/604800/0.000/true) tx: None, subsystem: flood, peer: ge-0/0/1.0, nodename: vMX-A5,
pid: 28562
```

Here triggering SPF-calculation to see if more specific needs to be advertised:

```
May 11 13:34:38.781 DEBG starting SPO computation, subsystem: southbound_
prefixes, nodename: vMX-A5, pid: 28562
May 11 13:34:38.781 DEBG computing SPFs on behalf of same level nodes [
    12503599506046976,
```

The result of the SPF-Calc revels that disaggregation is required:

```
May 11 13:34:38.782 DEBG system IDs needing disaggregation {12503601825038336, 12503602281103360}, 
subsystem: southbound_prefixes, nodename: vMX-A5, pid: 28562
```

And flooding the positive disaggregate towards vMX-A9:

```
May 11 13:34:38.784 DEBG flooding TIES output: Ok(()) upper: 51 len: [0, 0, 1], subsystem: flood, peer: 
ge-0/0/0.0, nodename: vMX-A5, pid: 28562
```

So what's the result of the disaggregation. First, an additional prefix is now received via RIFT. Prefix fc00:1000::192:168:20:14/126 is in the same PoD. Here's before the link break:

```
jcluser@vMX-A9# run show rift routes content

Prefix                          Active Metric N-Hop    All Present
-------------------------------+------+------+--------+-------------------
0.0.0.0/0                       S           2 80001207 S
::/0                            S           2 80001207 S
```

And after the link break:

```
jcluser@vMX-A9# run show rift routes content

Prefix                          Active Metric N-Hop    All Present
-------------------------------+------+------+--------+-------------------
0.0.0.0/0                       S           2 80001207 S
::/0                            S           2 80001207 S
fc00:1000::192:168:20:14/126    SExt        3 80001200 SExt
```

Prefix fc00:1000::192:168:20:14/126 now points to ge-0/0/0:

```
jcluser@vMX-A9# run show route fc00:1000::192:168:20:14/126

inet6.0: 14 destinations, 14 routes (14 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

fc00:1000::192:168:20:14/126
                   *[Static/200/100] 00:01:02, metric2 0
                    >  to fe80::250:56ff:fea2:2aaf via ge-0/0/0.0
```

As a recovery mechanism, vMX-A5 performs positive disaggregation, to provide vMX-A9 with enough routing information to still reach all destinations via a shortest path (Figure 4.4).

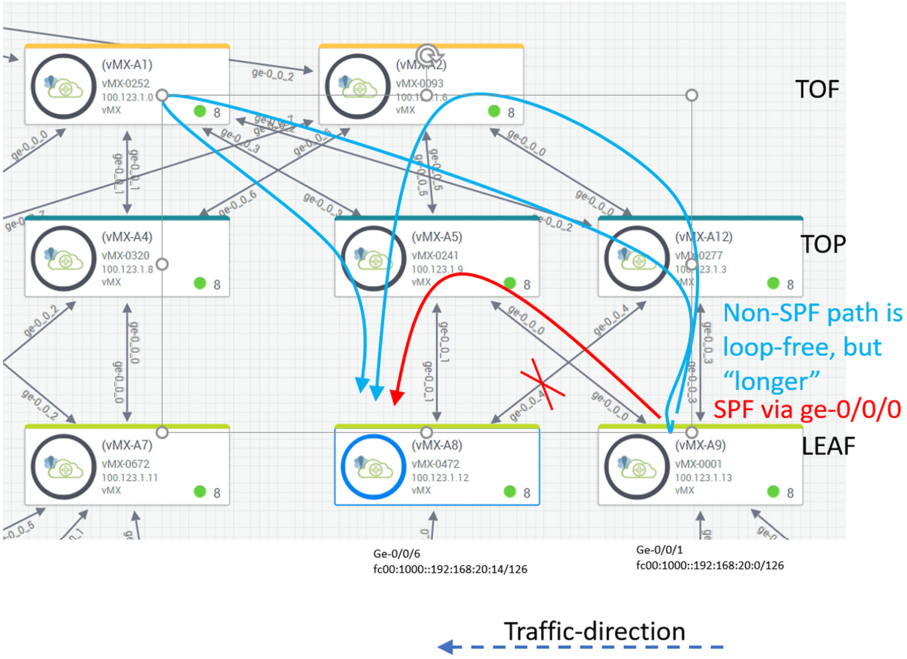


Figure 4.4 *Recovery Mechanism*

While vMX-A5 advertises the more specific route very fast, you might wonder what's the impact of not being fast? In this case, vMX-A9 might send its packets towards fc00:1000::192:168:20:14/126 via vMX-A12 and just take the longer path. vMX-A12 is aware that its link towards vMX-A8 is broken, hence it just relies on forwarding via default route.

Let's look into the RIFT DB and check on the positive disaggregation as done by vMX-A5:

```
[edit groups cg_extend_underlay]
jcluser@vMX-A9# run show rift topology nodes | match vMX-A5
 23 vMX-A5     002c6bf5c2c8c000       N      4              4             5      1      2 2020/05/11 20:07:40

[edit groups cg_extend_underlay]
jcluser@vMX-A9# run show rift database content | match 002c6bf5c2c8c000
S   002c6bf5c2c8c000 Node     10000000    5eb9ad4bbef1   601955   2020/05/11 20:07:39    604800    249    0
S   002c6bf5c2c8c000 Node     10000002    5eb9ad4cf4e5   601955   2020/05/11 20:07:39    604800    371    0
S   002c6bf5c2c8c000 Prefix    2000000e    5eb9ad51c937   601956   2020/05/11 20:07:40    604800    171    0
S   002c6bf5c2c8c000 Prefix    20000030    5eb9ad51e56e   601956   2020/05/11 20:07:40    604800    171    0
S   002c6bf5c2c8c000 PosExt    70000015    5eb9ad511a30   601771   2020/05/11 20:05:21    604800    183    0
```

The two prefix TIE's are the IPv4 and IPv6 default-routes advertised south:

```
jcluser@vMX-A9# run show rift tie 002c6bf5c2c8c000/S/Prefix/2000000e
TIE ID: 002c6bf5c2c8c000/S/Prefix__/2000000e

 Prefix                          Metric LPB ATT On Link
 -------------------------------+------+---+---+-------
  ::/0                                1

[edit groups cg_extend_underlay]
jcluser@vMX-A9# run show rift tie 002c6bf5c2c8c000/S/Prefix/20000030
TIE ID: 002c6bf5c2c8c000/S/Prefix__/20000030

 Prefix                          Metric LPB ATT On Link
 -------------------------------+------+---+---+-------
  0.0.0.0/0                           1
```

The remaining `PosExt` is the positive disaggregation of external prefixes:

```
jcluser@vMX-A9> show rift tie 002c6bf5c2c8c000/s/ex-/70000015
TIE ID: 002c6bf5c2c8c000/S/PosExt__/70000015

 Prefix                          Metric LPB ATT On Link
 -------------------------------+------+---+---+-------
  fc00:1000::192:168:20:14/126        2
```

4.1.10 Load Balancing (unequal/equal cost)

Now let's look into RIFT's recursive capability to load balance northbound traffic based on accumulated ECMP bandwidth. Because it contains lots of CLI-output, the summary describes the load balancing done by RIFT and main RIFT load-balancing characteristics:

- There is no LAG required at all.
- Interfaces of different speed can be mixed.
- RIFT supports unequal cost load balancing natively by default.
- All interfaces based on accumulated BW are taken into consideration. For example, 3 * 1gbps link in parallel count internally as a single link with 3gbps

4.1.10.1 Summary (Northbound)

RIFT can go far beyond plain ECMP based upon available bandwidth to its upstreams. RIFT also takes the uplink capacity of each of its northbound nodes into consideration as well!

Each RIFT node is aware of the northbound nodes' uplinks, so a true recursive load balancing within RIFT happens.

In our case vMX-A6 doesn't only perform plain ECMP to both its northbound nodes (vMX-A3 and vMX-A4), but also, based on its own available capacity, to both nodes. Furthermore, vMX-A6 takes into consideration the available uplink capacity for both vMX-A3 and vMX-A4. If vMX-A4 loses its uplink towards the TOF vMX-A2, then vMX-A3 has twice the uplink capacity compared to vMX-A4.

RIFT takes this into consideration and the vMX-A6 will shift more traffic to vMX-A3 as opposed to vMX-A4 in case of the described uplink failure below on vMX-A4.

Recursive means that each node in the tree is performing the same consideration by honoring its northbound node uplink capacity. All of this automatically happens!

4.1.10.2 So how does it all work?

The fact that a node receives TIE's from its northbound neighbor is important in this chapter, because it allows you to incorporate the uplink capacity of the upstream into the load balancing scheme. Also, it's important to recall that in case of node TIEs both the south and north ties contain details of *all* adjacencies.

Let's assume that the topology in Figure 4.5 has all links being of equal speed and BW, and that vMX-A1 is the ToF and Ubuntu-A1 is a leaf.

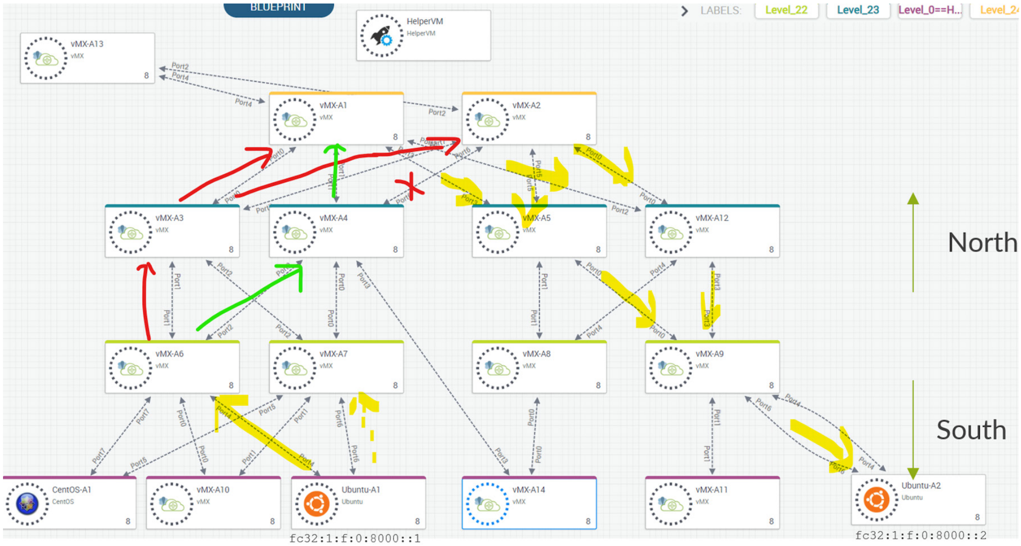


Figure 4.5 *vMX-A1 is the ToF and Ubuntu-A1 is a Leaf*

In nominal situations, vMX-A6 performs a 50%-%50% load balancing northbound:

```
jcluser@vMX-A6> show route table inet6.0 extensive

inet6.0: 12 destinations, 12 routes (12 active, 0 holddown, 0 hidden)
::/0 (1 entry, 1 announced)
TSI:
KRT in-kernel ::/0 -> {indirect(1048576) Flags NSR-incapable}
        *Static Preference: 20/100
                Next hop type: Indirect, Next hop index: 0
                Address: 0xc741d5c
                Next-hop reference count: 2
                Next hop type: Router, Next hop index: 0
                Next hop: fe80::250:56ff:fea2:d912 via ge-0/0/1.0 weight 0x1 balance 50%, selected
                Session Id: 0x0
                Next hop: fe80::250:56ff:fea2:db53 via ge-0/0/2.0 weight 0x1 balance 50%
                Session Id: 0x0
                Protocol next hop: fe80:100::1:0:54d5
                Indirect next hop: 0xc6bdd04 1048576 INH Session ID: 0x147 Weight 0x1
                State: <Active Int NSR-incapable Programmed>
                Age: 52:45      Metric2: 0
                Validation State: unverified
                Announcement bits (2): 0-KRT 1-Resolve tree 2
                AS path: I
                Indirect next hops: 1
                     Protocol next hop: fe80:100::1:0:54d5
                     Indirect next hop: 0xc6bdd04 1048576 INH Session ID: 0x147 Weight 0x1
                     Indirect path forwarding next hops: 2
                         Next hop type: Router
                         Next hop: fe80::250:56ff:fea2:d912 via ge-0/0/1.0 weight 0x1 balance 50%
                         Session Id: 0x0
                         Next hop: fe80::250:56ff:fea2:db53 via ge-0/0/2.0 weight 0x1 balance 50%
                         Session Id: 0x0
                         fe80:100::1:0:54d5/128 Originating RIB: inet6.3
                           Node path count: 1
                           Forwarding nexthops: 2
                                 Nexthop: fe80::250:56ff:fea2:d912 via ge-0/0/1.0
                                 Session Id: 0
                                 Nexthop: fe80::250:56ff:fea2:db53 via ge-0/0/2.0
                                 Session Id: 0
```

If distributed load balancing is enabled on the line card/performance forwarding engine (PFE), then the PFE reflects the load balancing as indicated by the RPD:

```
jcluser@vMX-A6> show route forwarding-table destination ::/0 extensive
Routing table: default.inet6 [Index 0]
Internet6:

Destination:  default
  Route type: user
  Route reference: 0                   Route interface-index: 0
  Multicast RPF nh index: 0
  P2mpidx: 0
  Flags: sent to PFE
  Next-hop type: indirect              Index: 1048576  Reference: 2
  Next-hop type: unilist               Index: 1048575  Reference: 2
  Nexthop: 0:50:56:a2:d9:12
  Next-hop type: unicast               Index: 609      Reference: 4
  Next-hop interface: ge-0/0/1.0    Weight: 0x1   Balance: 32768
  Nexthop: 0:50:56:a2:db:53
  Next-hop type: unicast               Index: 611      Reference: 4
  Next-hop interface: ge-0/0/2.0    Weight: 0x1   Balance: 65535
```

To understand the values better:

- The last interface balance is always 65535.
- ge-0/0/1 gets 32768/65535 balance = 50%
- ge-0/0/2 gets the remaining (65535-32768)/65k balance=50%. So finally both get 50% of the load.

Because vMX-A6 receives the node TIE's from vMX-A4, it is informed of the vMX-A4 northbound node uplink capacity:

```
Deriving  vMX-A4 Node-ID = 002c6bf56967c000

jcluser@vMX-A6> show rift topology nodes | match vMX-A4
 23 vMX-A4    002c6bf56967c000     N     4    1        4           4     1     1 2020/06/30 21:37:30
```

Checking the vMX-A4 NODE-TIE's:

```
jcluser@vMX-A6> show rift database content | match node| match 002c6bf56967c000
S   002c6bf56967c000 Node   10000000   5efa333549a4   599506   2020/06/30 20:10:52   604800   261  0
S   002c6bf56967c000 Node   10000002   5efa333424f4   604705   2020/06/30 21:37:30   604800   383  0
```

There are two TIEs (10000000 and 10000002). Decoding those indicates that vMX-A4 does have two (ToF) nodes northbound at Level 24:

```
jcluser@vMX-A6> show rift tie 002c6bf56967c000/s/node/10000000
TIE ID: 002c6bf56967c000/S/Node____/10000000
Name: vMX-A4, Level: 23
Capabilities: protocol_minor_version=0, flood_reduction=True, Flags: overload=False

 Neighbor          Lvl Cost Bandwidth
 -----------------+---+----+---------
  002c6bf58d4cc000  24  1    1000        <<< this is ToF vmx-A1

   Local ID Remote ID Intf Ndx Intf Name    Outer Key BFD
   ---------+---------+---------+------------+---------+---
    258      258       341                            yes
```

```
jcluser@vMX-A6> show rift tie 002c6bf56967c000/s/node/10000002
TIE ID: 002c6bf56967c000/S/Node____/10000002
Name: vMX-A4, Level: 23
Capabilities: protocol_minor_version=0, flood_reduction=True, Flags: overload=False

 Neighbor          Lvl Cost Bandwidth
 -----------------+---+----+---------
  002c6bf5d798c000  24  1    1000     <<< this is ToF vmx-A2

… (skipping more other Neighbors)
```

The database clearly indicates that vMX-A4 does have 2 * 1000mbit uplinks northbound (to Level 24). Let's trigger the failure, by bringing down the uplink of vMX-A4 towards vMX-A2:

```
jcluser@vMX-A4# top set interfaces ge-0/0/6 disable

[edit]
jcluser@vMX-A4# commit
```

Upon commit, TIEs get immediately flushed – this can be observed on the updated timestamp `21:47:05`:

```
jcluser@vMX-A6> show rift database content | match node| match 002c6bf56967c000
S   002c6bf56967c000 Node   10000000   5efa333549a4   598979   2020/06/30 20:10:52   604800   261   0
S   002c6bf56967c000 Node   10000002   5efa333424f5   604752   2020/06/30 21:47:05   604800   322   0
```

Looking into the TIE reveals that the previously seen neighbor, vMX-A2 (`002c6bf5d798c000`) in Level 24, is no longer existing:

```
jcluser@vMX-A6> show rift tie 002c6bf56967c000/s/node/10000002
TIE ID: 002c6bf56967c000/S/Node____/10000002
Name: vMX-A4, Level: 23
Capabilities: protocol_minor_version=0, flood_reduction=True, Flags: overload=False

 Neighbor          Lvl Cost Bandwidth
 -----------------+---+----+---------
  002c6bf53fe8c000  22  1    1000

   Local ID Remote ID Intf Ndx Intf Name    Outer Key BFD
   ---------+---------+---------+------------+---------+---
    259      259       342                           yes

 Neighbor          Lvl Cost Bandwidth
 -----------------+---+----+---------
  002c6bf57b95c000  22  1    1000

   Local ID Remote ID Intf Ndx Intf Name    Outer Key BFD
   ---------+---------+---------+------------+---------+---
    257      257       340                           yes
```

The vMX-A6 recalculates the desired load balancing ratio to a 60/40 ratio to accommodate vMX-A4's reduced uplink capacity:

```
jcluser@vMX-A6> show route detail ::/0

inet6.0: 12 destinations, 12 routes (12 active, 0 holddown, 0 hidden)
::/0 (1 entry, 1 announced)
        *Static Preference: 20/100
                Next hop type: Indirect, Next hop index: 0
                Address: 0xc741d5c
                Next-hop reference count: 2
                Next hop type: Router, Next hop index: 0
                Next hop: fe80::250:56ff:fea2:d912 via ge-0/0/1.0 weight 0x1 balance 60%, selected
                Session Id: 0x0
                Next hop: fe80::250:56ff:fea2:db53 via ge-0/0/2.0 weight 0x1 balance 40%
                Session Id: 0x0
                Protocol next hop: fe80:100::1:0:548d
                Indirect next hop: 0xc6bde84 1048578 INH Session ID: 0x14b Weight 0x1
                State: <Active Int NSR-incapable Programmed>
                Age: 5:09       Metric2: 0
                Validation State: unverified
                Announcement bits (2): 0-KRT 1-Resolve tree 2
                AS path: I
```

The 60/40 ratio is reflected in the PFE as well:

```
jcluser@vMX-A6> show route forwarding-table destination ::/0 extensive
Routing table: default.inet6 [Index 0]
Internet6:

Destination:  default
  Route type: user
  Route reference: 0                   Route interface-index: 0
  Multicast RPF nh index: 0
  P2mpidx: 0
  Flags: sent to PFE
  Next-hop type: indirect              Index: 1048578  Reference: 2
  Next-hop type: unilist               Index: 1048577  Reference: 2
  Nexthop: 0:50:56:a2:d9:12
  Next-hop type: unicast               Index: 609      Reference: 4
  Next-hop interface: ge-0/0/1.0    Weight: 0x1   Balance: 39321
  Nexthop: 0:50:56:a2:db:53
  Next-hop type: unicast               Index: 611      Reference: 4
  Next-hop interface: ge-0/0/2.0    Weight: 0x1   Balance: 65535

ge-0/0/1: 39321/65535 = 60%
ge-0/0/2: (65535-39321)/65535= 40%
```

How was the 60/40 derived? RIFT is adding the bandwidth as stated in the TIEs, and because in our demo topology each link is 1000mbit the following math applies:

RED-path: 1 * 1000mbps + 2 * 1000 mbps = 3000mbps

Green-path: 1 * 1000mbps + 1 * 1000mbps = 2000mbps

3000/2000=60/40

The bandwidth-ratio of 3000 / 2000 =60/40=1.5

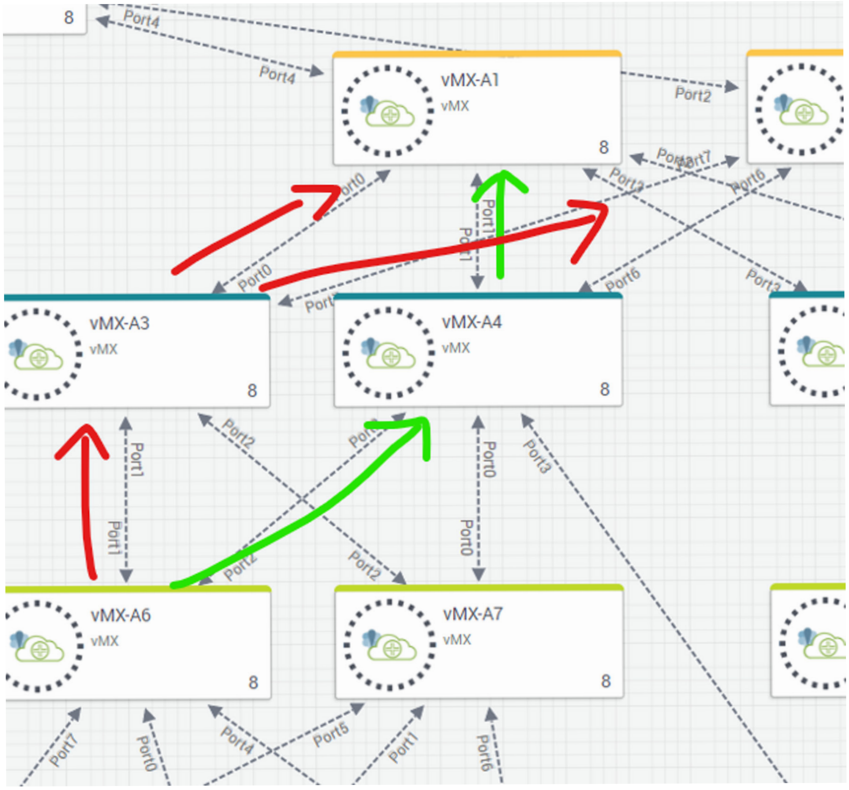


Figure 4.6 *Load Balancing in Depth*

We just explained in depth, which also took into account the northbound nodes uplink capacity to determine the load split.

Load balancing in the south direction is different. Southbound is just load balanced on the local ECMP paths available without involving downstream link capacity. With southbound, the tree becomes broader, which is different compared to the northbound where the tree becomes more narrowed.

So far, there is not yet any suitable algorithm available for taking into account southbound nodes' link capacity, especially when some links fail.

4.2 Troubleshooting

Here's a quick review of some RIFT troubleshooting techniques.

4.2.1 RIFT Debugging and Problem Reporting

RIFT does not produce cores except in very extreme cases. It reports every failure by extensive logging and sometimes backtraces on exit. To report a problem, the topologies used together with invocations and resulting output are needed. Depending on the case more detailed runs with debug logs could be needed as well.

To enable debug traces:

1. Flip on RIFT proxy tracing (this is the daemon that deals with config that really forks out riftd):

```
set protocols rift proxy-process traceoptions file rift-proxyd
set protocols rift proxy-process traceoptions file size 10m
set protocols rift proxy-process traceoptions file files 2
set protocols rift proxy-process traceoptions level all
set protocols rift proxy-process traceoptions flag all
```

2. Set RIFT tracing (that's `riftd`):

```
set protocols rift traceoptions file riftd size 10m files 2
set protocols rift traceoptions level verbose
set protocols rift traceoptions flag node
```

For deeper investigation of the interaction between RIFT and pRPD, pRPD traceoptions can be enabled. As an example, RIFT logs with `level-info` and `prpd traceoptions` are enabled. The corresponding log files are defined and stored in the /var/log/ directory:

```
routing-options {
    programmable-rpd {
        traceoptions {
            file prpd.log size 10m;
            flag all;
        }
    }
}
protocols {
    rift {
        traceoptions {
            file rift.log;
            level info;
            flag rib;
            flag fib;
            flag interface-manager;
```

4.2.2 Common RIFT Errors and Failure Scenarios

Q: I see an adjacency flapping up/down showing rejects due to Multiple Neighbors or Remote Uses Our Own SystemID

A: RIFT does not support more than two neighbors on an Ethernet link forming a p2p adjacency, or a node's own interfaces looped back. Possibly incorrect cabling.

Q: All my switches show undefined level and do not form ThreeWay adjacencies albeit I see LIEs being sent and received.

*A: Possibly there is no ToF level configuration. *All* top of fabric switches MUST be configured with* `level top-of-fabric` *to provide an anchor for ZTP.*

4.2.3. Restarting RIFT

RIFT relies on two major processes: `rift-proxyd` and `riftd`.

Rift proxyd is responsible for configuration and startup while riftd runs the protocol. For efficiency, it is highly multi-threaded as you can see here:

```
jcluser@vMX-A6> show system processes extensive |match rift
  PID USERNAME PRI NICE   SIZE RES STATE TIME WCPU COMMAND
11886 root   20 0 98980K 60424K kqread   0:12   0.00% riftd{ge-0/0/1.0}
11886 root   20 0 98980K 60424K kqread   0:11   0.00% riftd{ge-0/0/2.0}
11886 root   20 0 98980K 60424K kqread   0:07   0.00% riftd{ge-0/0/4.0}
11886 root   20 0 98980K 60424K kqread   0:07   0.00% riftd{ge-0/0/0.0}
11886 root   20 0 98980K 60424K kqread   0:07   0.00% riftd{ge-0/0/7.0}
11886 root   20 0 98980K 60424K kqread   0:06   0.00% riftd{NODE}
11886 root   20 0 98980K 60424K uwait    0:05   0.00% riftd{riftd}
11886 root   20 0 98980K 60424K uwait    0:05   0.00% riftd{riftd}
11886 root   20 0 98980K 60424K uwait    0:05   0.00% riftd{riftd}
11886 root   20 0 98980K 60424K uwait    0:05   0.00% riftd{riftd}
11886 root   20 0 98980K 60424K uwait    0:05   0.00% riftd{riftd}
11886 root   20 0 98980K 60424K uwait    0:05   0.00% riftd{riftd}
11886 root   20 0 98980K 60424K uwait    0:05   0.00% riftd{riftd}
11886 root   20 0 98980K 60424K RUN      0:05   0.00% riftd{riftd}
11886 root   20 0 98980K 60424K kqread   0:05   0.00% riftd{ge-0/0/6.0}
11886 root   20 0 98980K 60424K kqread   0:05   0.00% riftd{ZTP}
11886 root   20 0 98980K 60424K kqread   0:04   0.00% riftd{ge-0/0/3.0}
11886 root   20 0 98980K 60424K kqread   0:04   0.00% riftd{ge-0/0/5.0}
11886 root   20 0 98980K 60424K kqread   0:04   0.00% riftd{RIBREDIS}
11886 root   20 0 98980K 60424K uwait    0:04   0.00% riftd{riftd}
11886 root   20 0 98980K 60424K uwait    0:03   0.00% riftd{riftd}
11886 root   20 0 98980K 60424K RUN      0:02   0.00% riftd{LSDBREDIS}
11884 root   20 0   871M 16592K select   0:02   0.00% rift-proxyd
11886 root   20 0 98980K 60424K uwait    0:01   0.00% riftd{riftd}
11886 root   20 0 98980K 60424K kqread   0:01   0.00% riftd{LSDB}
11886 root   20 0 98980K 60424K uwait    0:01   0.00% riftd{riftd}
11886 root   20 0 98980K 60424K kqread   0:01   0.00% riftd{RIB}
11886 root   20 0 98980K 60424K uwait    0:01   0.00% riftd{riftd}
11886 root   20 0 98980K 60424K uwait    0:01   0.00% riftd{riftd}
```

```
11886 root   20 0 98980K 60424K uwait    0:01   0.00% riftd{riftd}
11886 root   20 0 98980K 60424K uwait    0:00   0.00% riftd{riftd}
11886 root   20 0 98980K 60424K kqread   0:00   0.00% riftd{SPF}
11886 root   20 0 98980K 60424K uwait    0:00   0.00% riftd{riftd}
11886 root   20 0 98980K 60424K kqread   0:00   0.00% riftd{SPO}
11886 root   20 0 98980K 60424K kqread   0:00   0.00% riftd{REDIS:ge-0/0/1.0}
11886 root   20 0 98980K 60424K kqread   0:00   0.00% riftd{REDIS:ge-0/0/4.0}
11886 root   20 0 98980K 60424K kqread   0:00   0.00% riftd{REDIS:ge-0/0/0.0}
11886 root   20 0 98980K 60424K kqread   0:00   0.00% riftd{REDIS:ge-0/0/2.0}
11886 root   20 0 98980K 60424K kqread   0:00   0.00% riftd{REDIS:ge-0/0/7.0}
11886 root   20 0 98980K 60424K kqread   0:00   0.00% riftd{FRE}
11886 root   20 0 98980K 60424K uwait    0:00   0.00% riftd{riftd}
11886 root   22 0 98980K 60424K uwait    0:00   0.00% riftd{riftd}
11886 root   20 0 98980K 60424K uwait    0:00   0.00% riftd{riftd}
11886 root   20 0 98980K 60424K uwait    0:00   0.00% riftd{JET GRPC "N"}
11886 root   20 0 98980K 60424K kqread   0:00   0.00% riftd{IFM}
11886 root   20 0 98980K 60424K kqread   0:00   0.00% riftd{FIBFLUSH}
11886 root   20 0 98980K 60424K kqread   0:00   0.00% riftd{FIB}
11886 root   20 0 98980K 60424K uwait    0:00   0.00% riftd{JET GRPC "S"}
11886 root   20 0 98980K 60424K uwait    0:00   0.00% riftd{JET/IFNOTIFY}
11886 root   20 0 98980K 60424K kqread   0:00   0.00% riftd{riftd}
11886 root   20 0 98980K 60424K uwait    0:00   0.00% riftd{JET/BFDNOTIFY}
11886 root   20 0 98980K 60424K kqread   0:00   0.00% riftd{riftd}
11886 root   20 0 98980K 60424K kqread   0:00   0.00% riftd{REDIS:ge-0/0/5.0}
11886 root   20 0 98980K 60424K kqread   0:00   0.00% riftd{REDIS:ge-0/0/6.0}
11886 root   22 0 98980K 60424K uwait    0:00   0.00% riftd{riftd}
11886 root   20 0 98980K 60424K kqread   0:00   0.00% riftd{REDIS:ge-0/0/3.0}
11886 root   20 0 98980K 60424K uwait    0:00   0.00% riftd{riftd}
11886 root   20 0 98980K 60424K uwait    0:00   0.00% riftd{riftd}
11886 root   20 0 98980K 60424K uwait    0:00   0.00% riftd{riftd}

jcluser@vMX-A6>
```

RIFT can be restarted using the `restart rift-proxyd` command.

Before the restart:

```
jcluser@vMX-A6> show rift node statistics
Starttime: 2020/05/15 16:12:07.249
Service Requests: 37, Failed Requests: 0
```

Restarting RIFT:

```
jcluser@vMX-A6> restart rift-proxyd
Routing In FAT Trees Protocol Proxy started, pid 13648
```

After the restart:

```
jcluser@vMX-A6> show rift node statistics
Starttime: 2020/05/15 16:20:02.615
Service Requests: 37, Failed Requests: 0

jcluser@vMX-A6>
```

You can verify that RIFT has been restarted using the RIFT start time displayed by the `show rift node statistics` command.

Chapter 5

Wireshark RIFT Dissector

The RIFT dissector was designed as a Wireshark plugin packet dissector and implemented in C language following the API that this tool provides. In addition, there is a partial dissector for the RIFT outer security envelope header in Lua language. The implementation is open source and available at https://gitlab.com/fing-mina/datacenters/rift-dissector.

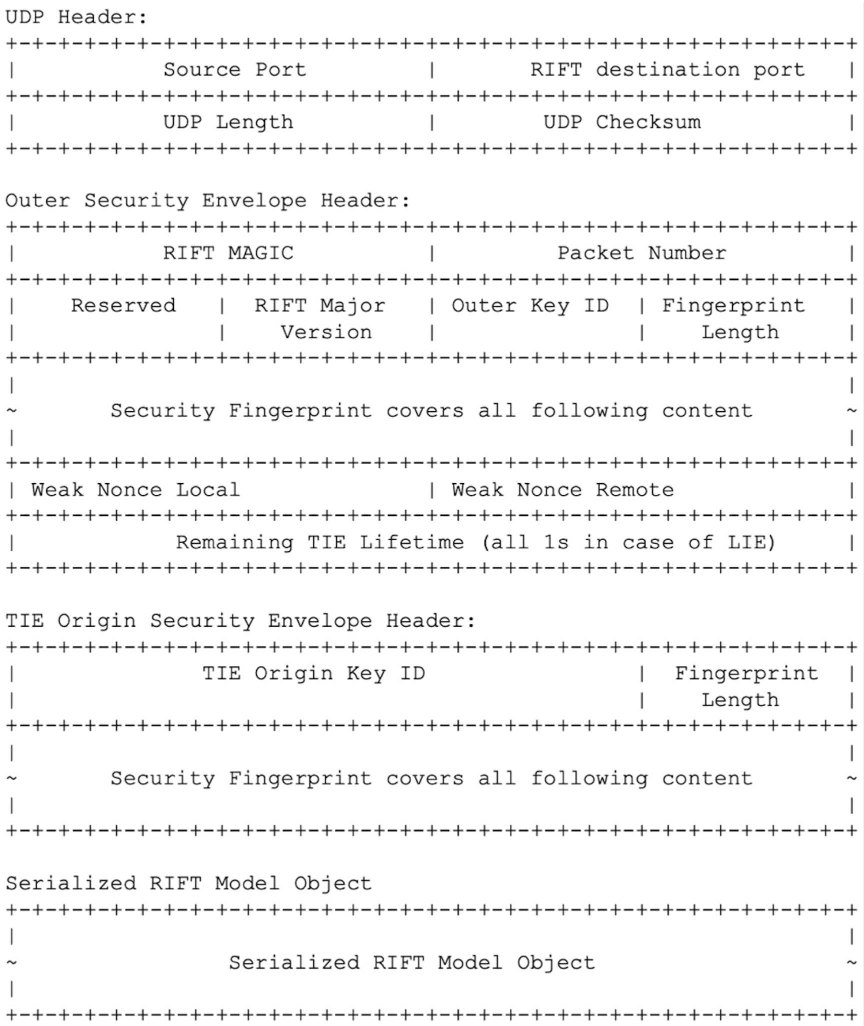
5.1. Design

In Figure 5.1, you'll see the envelope for a RIFT packet that is being transported inside the payload of an UDP packet. The dissector design follows the following three stages:

1. Outer Security Envelope Header: At this stage the dissector must recognize a set of specified and static fields.

2. TIE Origin Security Envelope Header: At this point, along with the previous stage, the dissection must identify a set of specified and static fields. There is only one difference between this stage and the previous one, and it is that here this set of fields is present if and only if the type of the RIFT packet is a TIE.

3. Serialized RIFT Model Object: Finally, the dissector has to process a set of dynamic fields that follows the Thrift Binary protocol encoding (https://github.com/apache/thrift/blob/master/doc/specs/thrift-binary-protocol.md) model defined for RIFT.



```
UDP Header:
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Source Port         |    RIFT destination port      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           UDP Length          |        UDP Checksum           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

Outer Security Envelope Header:
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           RIFT MAGIC          |         Packet Number         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    Reserved   |  RIFT Major   | Outer Key ID  | Fingerprint   |
|               |    Version    |               |    Length     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
~       Security Fingerprint covers all following content       ~
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Weak Nonce Local              | Weak Nonce Remote             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|            Remaining TIE Lifetime (all 1s in case of LIE)     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

TIE Origin Security Envelope Header:
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              TIE Origin Key ID                |  Fingerprint  |
|                                               |    Length     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
~       Security Fingerprint covers all following content       ~
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

Serialized RIFT Model Object
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
~                Serialized RIFT Model Object                   ~
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 5.1 *Security Envelope (Exracted From* https://datatracker.ietf.org/doc/draft-ietf-rift-rift/ *)*

The stages mentioned previously are derived from the security envelope for the RIFT packets shown in the Figure 5.1. The first two stages are designed with a classical approach, therefore the dissector has to follow a static definition that assigns a range of bytes to a field. The last stage instead represents a change of paradigm. In consequence, provided this new approach, a RIFT packet has to be decoded knowing in advance how the data is specified in the Thrift definition files and how the data types and structures are encoded. It is key to mention that there is a Thrift compiler that generates a Thrift decoder based on a given model. Therefore, two options are identified to design the dissection of the Serialized RIFT Model Object: i) passing this part of the binary packet to a Thrift back-end; or, ii) write the C code based on the encoded defined in Thrift for the data types implicated. The latter must be done following the schema for information elements, whose Interface Definition Language (IDL) is Thrift.

Wireshark's user interface allows the user to highlight some specific fields in the decoded packet, as well as highlighting the corresponding bytes in the hex dump of the binary packet.

To that end, the Thrift decoder that is employed in Wireshark is needed to first know the precise sequence in which the fields were encoded in the binary message, that could potentially not be the same order as in the model; and secondly, know the mapping between the bytes in the binary message and the fields in the decoded message. Since the generated code by the Thrift compiler does not put together this information, the dissector presented follows the above option (i).

5.2 Implementation

At first, the dissection of the security envelope header, which is specified in the RIFT draft [https://datatracker.ietf.org/doc/draft-ietf-rift-rift/] and has the format shown in Figure 5.1, is processed. This implementation follows a static representation of the bits corresponding to each field, for example, the first four bytes represent the RIFT Magic value for the packet, continued by the other four bytes, which represent the Packet Number value.

Having in mind that the protocol was not identified with any particular range of ports at the time of the dissector development, it was implemented as a Wireshark heuristic dissector: in other words, the dissector identifies a packet as RIFT if the field RIFT Magic contains the proper value defined as the hexadecimal 0xA1F7 in the current draft of the protocol.

After recognizing the packets as belonging to the RIFT protocol, the dissector identifies the fields of the security envelope header and finally dissects the serialized RIFT Model Object that is encoded with Thrift. Inside this encoding is a structure with the content of a RIFT packet. The implemented dissector identifies and performs a complete dissection of all these types of RIFT packets: Link Information Element (LIE), Topology Information Element (TIE), Topology Information Description Element (TIDE), and Topology Information Request Element (TIRE).

Additionally, some specific fields were added to the dissector in order to simplify the filtering process, for example, one with the explicit type of the RIFT packet.

5.3 Set Up and Deployment

The set up and deployment of the dissector is an easy task. Basically you have to uninstall previous versions of Wireshark and recompile from source code the recommended one after injecting the code of the dissector.

There is an example for a Linux Debian system:

1. Uninstall previous versions of Wireshark.

2. Install additional dependencies (or CMake can fail): apt install libglib2.0-dev libgcrypt20-dev flex bison qtbase5-dev qttools5-dev qtmultimedia5-dev libqt5svg5-dev.

3. Download Wireshark 3.2.2.

4. Make new folder under $wireshark_dir$/plugins/epan/rift.

5. Copy the files of the dissector: packet-rift.c, CMakeLists.txt, AUTHORS and Readme.md into the new folder.

6. Modify file $wireshark_dir$/CMakeListsCustom.txt.example, rename to CMakeListsCustom.txt and edit line 16 of file this way:

```
# Fail CMake stage if any of these plugins are missing from source tree
set(CUSTOM_PLUGIN_SRC_DIR
#       private_plugins/foo
# or
   plugins/epan/rift
)
```

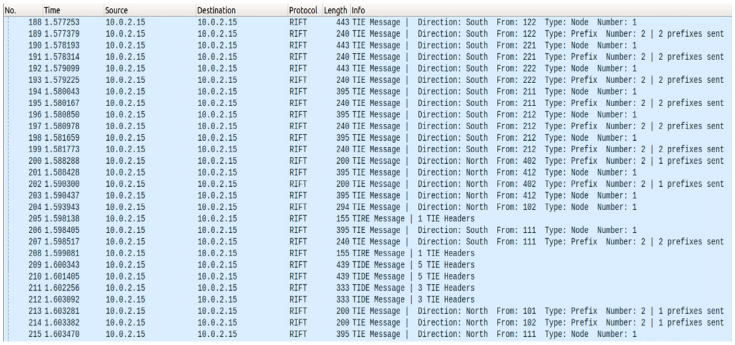
7. Create a build directory under $wireshark_dir$.

8. Inside the build directory you have to do the cmake, make, and make install as follow:

The configuration and set up is more detailed and maintained on the dissector source. There is also a docker image with all the set up and deployment already done.

5.4 Dissecting RIFT

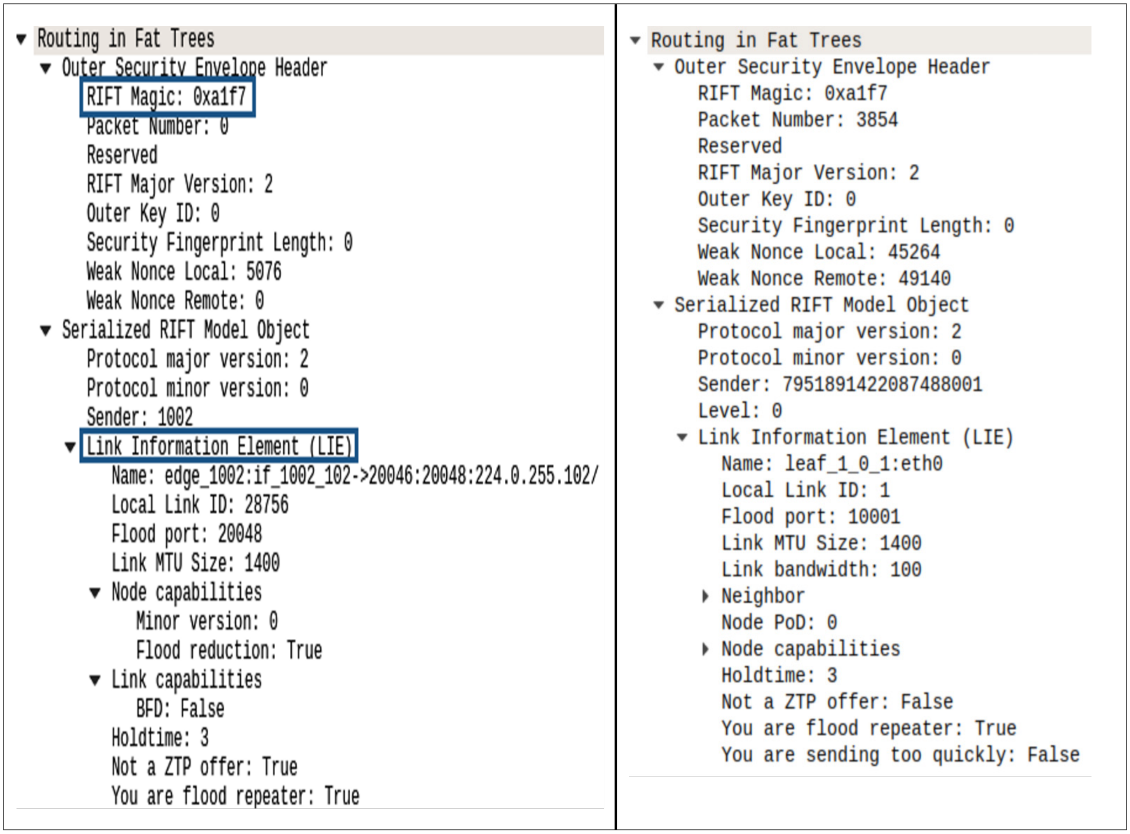
Figure 5.2 shows the Wireshark general view for a capture with RIFT traffic after the plugin is integrated. With this general view you can notice the protocol name and the type of packet captured, in the Protocol and Info columns respectively.



```
No.   Time      Source      Destination  Protocol Length Info
  188 1.577253  10.0.2.15   10.0.2.15    RIFT     443 TIE Message |  Direction: South  From: 122  Type: Node  Number: 1
  189 1.577379  10.0.2.15   10.0.2.15    RIFT     240 TIE Message |  Direction: South  From: 122  Type: Prefix  Number: 2 | 2 prefixes sent
  190 1.578193  10.0.2.15   10.0.2.15    RIFT     443 TIE Message |  Direction: South  From: 221  Type: Node  Number: 1
  191 1.578314  10.0.2.15   10.0.2.15    RIFT     240 TIE Message |  Direction: South  From: 221  Type: Prefix  Number: 2 | 2 prefixes sent
  192 1.579099  10.0.2.15   10.0.2.15    RIFT     443 TIE Message |  Direction: South  From: 222  Type: Node  Number: 1
  193 1.579225  10.0.2.15   10.0.2.15    RIFT     240 TIE Message |  Direction: South  From: 222  Type: Prefix  Number: 2 | 2 prefixes sent
  194 1.580043  10.0.2.15   10.0.2.15    RIFT     395 TIE Message |  Direction: South  From: 211  Type: Node  Number: 1
  195 1.580167  10.0.2.15   10.0.2.15    RIFT     240 TIE Message |  Direction: South  From: 211  Type: Prefix  Number: 2 | 2 prefixes sent
  196 1.580850  10.0.2.15   10.0.2.15    RIFT     395 TIE Message |  Direction: South  From: 212  Type: Node  Number: 1
  197 1.580978  10.0.2.15   10.0.2.15    RIFT     240 TIE Message |  Direction: South  From: 212  Type: Prefix  Number: 2 | 2 prefixes sent
  198 1.581659  10.0.2.15   10.0.2.15    RIFT     395 TIE Message |  Direction: South  From: 212  Type: Node  Number: 1
  199 1.581773  10.0.2.15   10.0.2.15    RIFT     240 TIE Message |  Direction: South  From: 212  Type: Prefix  Number: 2 | 2 prefixes sent
  200 1.588288  10.0.2.15   10.0.2.15    RIFT     200 TIE Message |  Direction: North  From: 402  Type: Prefix  Number: 2 | 1 prefixes sent
  201 1.588428  10.0.2.15   10.0.2.15    RIFT     395 TIE Message |  Direction: North  From: 412  Type: Node  Number: 1
  202 1.590300  10.0.2.15   10.0.2.15    RIFT     200 TIE Message |  Direction: North  From: 402  Type: Prefix  Number: 2 | 1 prefixes sent
  203 1.590437  10.0.2.15   10.0.2.15    RIFT     395 TIE Message |  Direction: North  From: 412  Type: Node  Number: 1
  204 1.593943  10.0.2.15   10.0.2.15    RIFT     294 TIE Message |  Direction: North  From: 102  Type: Node  Number: 1
  205 1.598138  10.0.2.15   10.0.2.15    RIFT     155 TIRE Message | 1 TIE Headers
  206 1.598405  10.0.2.15   10.0.2.15    RIFT     395 TIE Message |  Direction: South  From: 111  Type: Node  Number: 1
  207 1.598517  10.0.2.15   10.0.2.15    RIFT     240 TIE Message |  Direction: South  From: 111  Type: Prefix  Number: 2 | 2 prefixes sent
  208 1.599081  10.0.2.15   10.0.2.15    RIFT     155 TIRE Message | 1 TIE Headers
  209 1.600343  10.0.2.15   10.0.2.15    RIFT     439 TIDE Message | 5 TIE Headers
  210 1.601405  10.0.2.15   10.0.2.15    RIFT     439 TIDE Message | 5 TIE Headers
  211 1.602256  10.0.2.15   10.0.2.15    RIFT     333 TIDE Message | 3 TIE Headers
  212 1.603092  10.0.2.15   10.0.2.15    RIFT     333 TIDE Message | 3 TIE Headers
  213 1.603281  10.0.2.15   10.0.2.15    RIFT     200 TIE Message |  Direction: North  From: 101  Type: Prefix  Number: 2 | 1 prefixes sent
  214 1.603382  10.0.2.15   10.0.2.15    RIFT     200 TIE Message |  Direction: North  From: 102  Type: Prefix  Number: 2 | 1 prefixes sent
  215 1.603470  10.0.2.15   10.0.2.15    RIFT     395 TIE Message |  Direction: North  From: 111  Type: Node  Number: 1
```

Figure 5.2 Wireshark View with RIFT Dissector Integrated

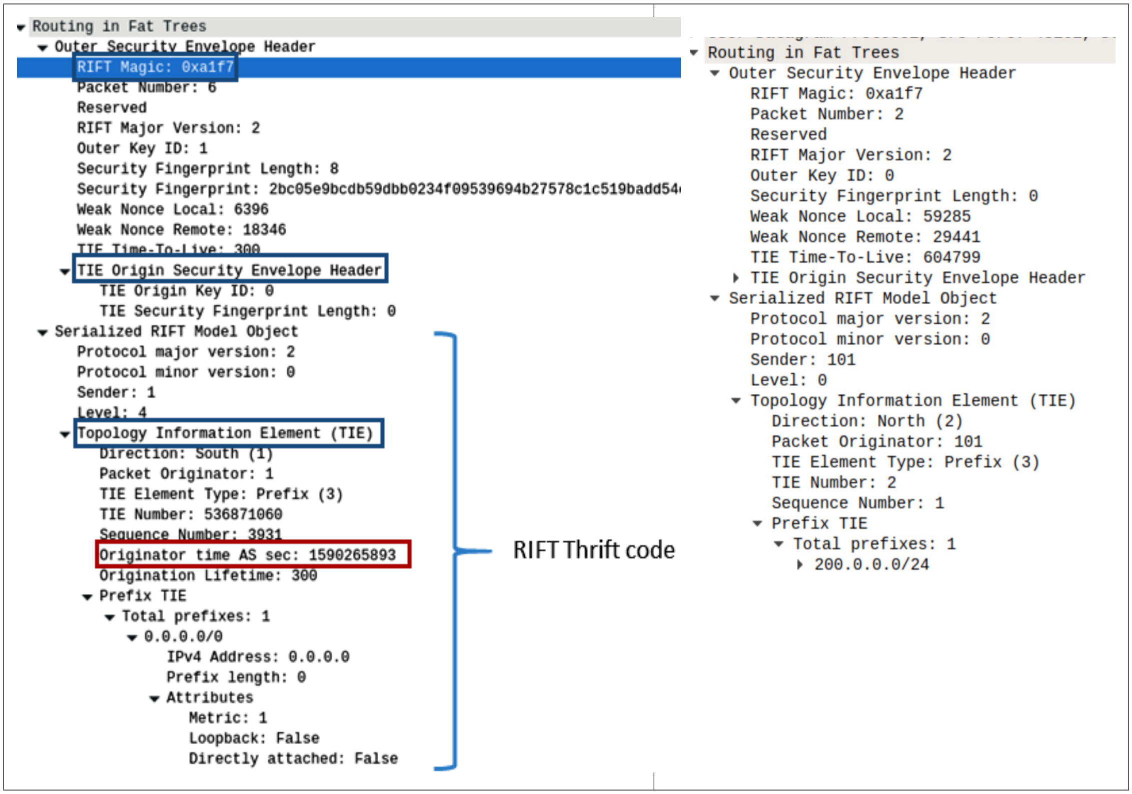
As shown next in Figure 5.3, you can get a complete dissection of the LIE packets. The figure shows the dissection of two LIE packets, one per each implementation available. On the right we can observe the dissection of a LIE packet generated by the RIFT Juniper implementation and on the left the generated by the RIFT-python one. Figures 5.3, 5.4, 5.5, and 5.6 show the same comparison between these two implementations for the LIE, TIE, TIRE, and TIDE packets respectively.



```
▼ Routing in Fat Trees
  ▼ Outer Security Envelope Header
      RIFT Magic: 0xa1f7
      Packet Number: 0
      Reserved
      RIFT Major Version: 2
      Outer Key ID: 0
      Security Fingerprint Length: 0
      Weak Nonce Local: 5076
      Weak Nonce Remote: 0
  ▼ Serialized RIFT Model Object
      Protocol major version: 2
      Protocol minor version: 0
      Sender: 1002
    ▼ Link Information Element (LIE)
        Name: edge_1002:if_1002_102->20046:20048:224.0.255.102/
        Local Link ID: 28756
        Flood port: 20048
        Link MTU Size: 1400
      ▼ Node capabilities
          Minor version: 0
          Flood reduction: True
      ▼ Link capabilities
          BFD: False
        Holdtime: 3
        Not a ZTP offer: True
        You are flood repeater: True
```

```
▼ Routing in Fat Trees
  ▼ Outer Security Envelope Header
      RIFT Magic: 0xa1f7
      Packet Number: 3854
      Reserved
      RIFT Major Version: 2
      Outer Key ID: 0
      Security Fingerprint Length: 0
      Weak Nonce Local: 45264
      Weak Nonce Remote: 49140
  ▼ Serialized RIFT Model Object
      Protocol major version: 2
      Protocol minor version: 0
      Sender: 7951891422087488001
      Level: 0
    ▼ Link Information Element (LIE)
        Name: leaf_1_0_1:eth0
        Local Link ID: 1
        Flood port: 10001
        Link MTU Size: 1400
        Link bandwidth: 100
      ▶ Neighbor
        Node PoD: 0
      ▶ Node capabilities
        Holdtime: 3
        Not a ZTP offer: False
        You are flood repeater: True
        You are sending too quickly: False
```

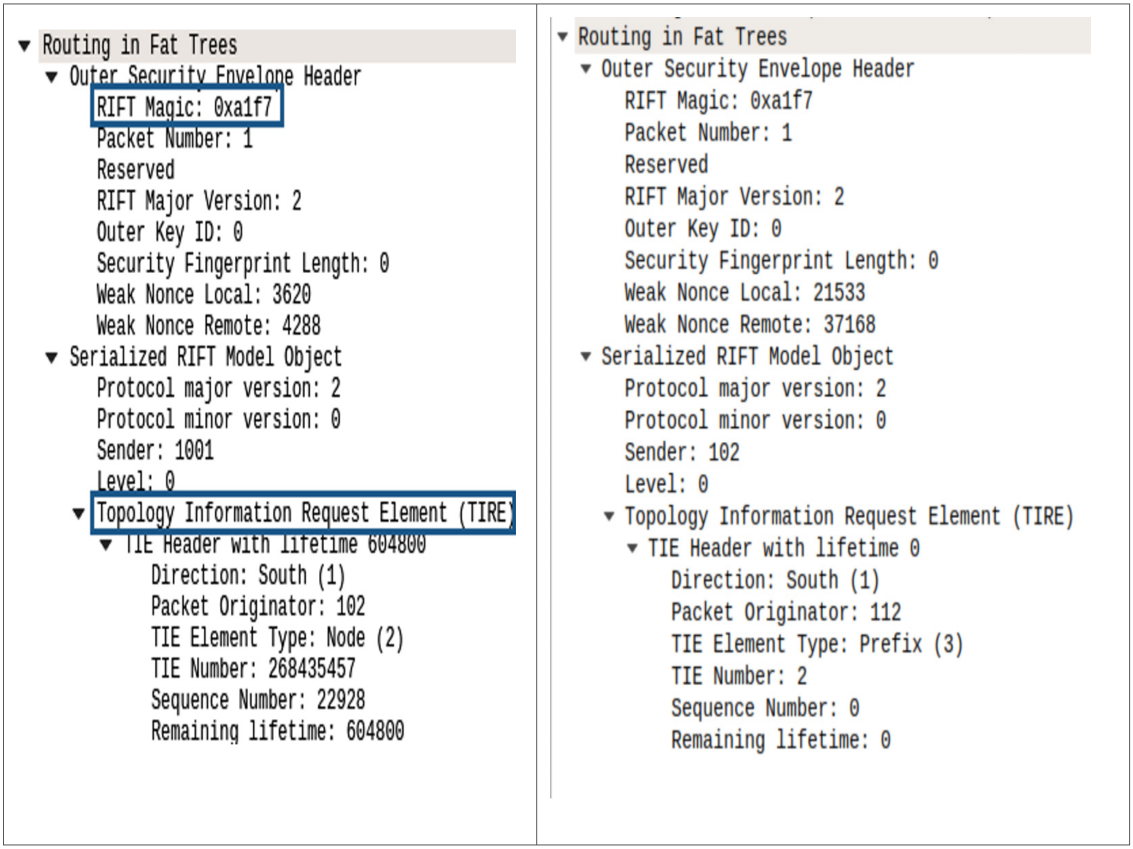
Figure 5.3 LIE Packets



```
▾ Routing in Fat Trees
  ▾ Outer Security Envelope Header
      RIFT Magic: 0xa1f7
      Packet Number: 6
      Reserved
      RIFT Major Version: 2
      Outer Key ID: 1
      Security Fingerprint Length: 8
      Security Fingerprint: 2bc05e9bcdb59dbb0234f09539694b27578c1c519badd54
      Weak Nonce Local: 6396
      Weak Nonce Remote: 18346
      TIE Time-To-Live: 300
    ▾ TIE Origin Security Envelope Header
        TIE Origin Key ID: 0
        TIE Security Fingerprint Length: 0
  ▾ Serialized RIFT Model Object
      Protocol major version: 2
      Protocol minor version: 0
      Sender: 1
      Level: 4
    ▾ Topology Information Element (TIE)
        Direction: South (1)
        Packet Originator: 1
        TIE Element Type: Prefix (3)
        TIE Number: 536871060
        Sequence Number: 3931
        Originator time AS sec: 1590265893
        Origination Lifetime: 300
      ▾ Prefix TIE
        ▾ Total prefixes: 1
          ▾ 0.0.0.0/0
              IPv4 Address: 0.0.0.0
              Prefix length: 0
            ▾ Attributes
                Metric: 1
                Loopback: False
                Directly attached: False
```

```
▾ Routing in Fat Trees
  ▾ Outer Security Envelope Header
      RIFT Magic: 0xa1f7
      Packet Number: 2
      Reserved
      RIFT Major Version: 2
      Outer Key ID: 0
      Security Fingerprint Length: 0
      Weak Nonce Local: 59285
      Weak Nonce Remote: 29441
      TIE Time-To-Live: 604799
    ▸ TIE Origin Security Envelope Header
  ▾ Serialized RIFT Model Object
      Protocol major version: 2
      Protocol minor version: 0
      Sender: 101
      Level: 0
    ▾ Topology Information Element (TIE)
        Direction: North (2)
        Packet Originator: 101
        TIE Element Type: Prefix (3)
        TIE Number: 2
        Sequence Number: 1
      ▾ Prefix TIE
        ▾ Total prefixes: 1
          ▸ 200.0.0.0/24
```

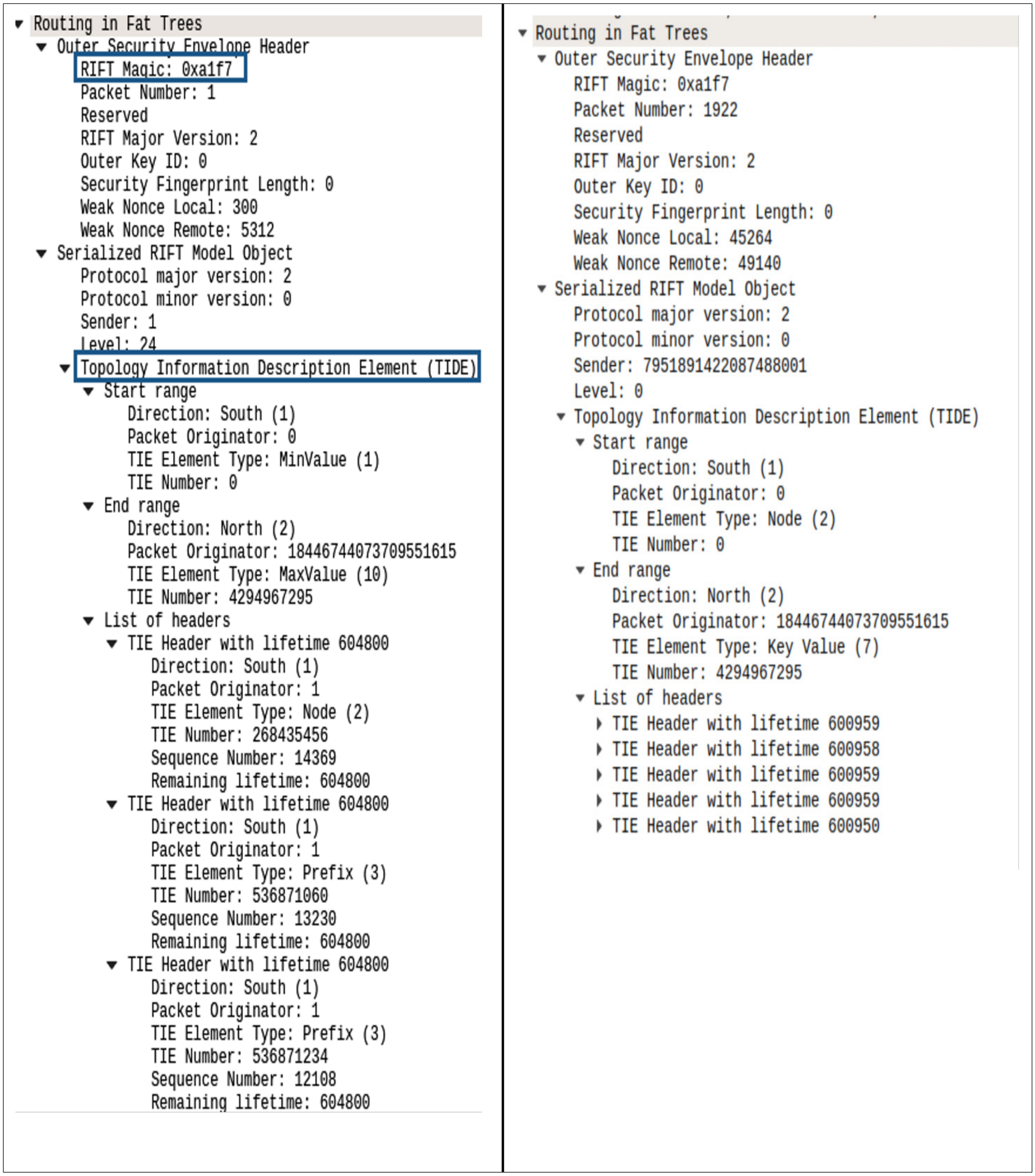
Figure 5.4 TIE Packets



```
▼ Routing in Fat Trees
  ▼ Outer Security Envelope Header
      RIFT Magic: 0xa1f7
      Packet Number: 1
      Reserved
      RIFT Major Version: 2
      Outer Key ID: 0
      Security Fingerprint Length: 0
      Weak Nonce Local: 3620
      Weak Nonce Remote: 4288
  ▼ Serialized RIFT Model Object
      Protocol major version: 2
      Protocol minor version: 0
      Sender: 1001
      Level: 0
    ▼ Topology Information Request Element (TIRE)
      ▼ TIE Header with lifetime 604800
          Direction: South (1)
          Packet Originator: 102
          TIE Element Type: Node (2)
          TIE Number: 268435457
          Sequence Number: 22928
          Remaining lifetime: 604800
```

```
▼ Routing in Fat Trees
  ▼ Outer Security Envelope Header
      RIFT Magic: 0xa1f7
      Packet Number: 1
      Reserved
      RIFT Major Version: 2
      Outer Key ID: 0
      Security Fingerprint Length: 0
      Weak Nonce Local: 21533
      Weak Nonce Remote: 37168
  ▼ Serialized RIFT Model Object
      Protocol major version: 2
      Protocol minor version: 0
      Sender: 102
      Level: 0
    ▼ Topology Information Request Element (TIRE)
      ▼ TIE Header with lifetime 0
          Direction: South (1)
          Packet Originator: 112
          TIE Element Type: Prefix (3)
          TIE Number: 2
          Sequence Number: 0
          Remaining lifetime: 0
```

Figure 5.5 TIRE Packets



```
▾ Routing in Fat Trees
  ▾ Outer Security Envelope Header
      RIFT Magic: 0xa1f7
      Packet Number: 1
      Reserved
      RIFT Major Version: 2
      Outer Key ID: 0
      Security Fingerprint Length: 0
      Weak Nonce Local: 300
      Weak Nonce Remote: 5312
  ▾ Serialized RIFT Model Object
      Protocol major version: 2
      Protocol minor version: 0
      Sender: 1
      Level: 24
    ▾ Topology Information Description Element (TIDE)
      ▾ Start range
          Direction: South (1)
          Packet Originator: 0
          TIE Element Type: MinValue (1)
          TIE Number: 0
      ▾ End range
          Direction: North (2)
          Packet Originator: 18446744073709551615
          TIE Element Type: MaxValue (10)
          TIE Number: 4294967295
      ▾ List of headers
        ▾ TIE Header with lifetime 604800
            Direction: South (1)
            Packet Originator: 1
            TIE Element Type: Node (2)
            TIE Number: 268435456
            Sequence Number: 14369
            Remaining lifetime: 604800
        ▾ TIE Header with lifetime 604800
            Direction: South (1)
            Packet Originator: 1
            TIE Element Type: Prefix (3)
            TIE Number: 536871060
            Sequence Number: 13230
            Remaining lifetime: 604800
        ▾ TIE Header with lifetime 604800
            Direction: South (1)
            Packet Originator: 1
            TIE Element Type: Prefix (3)
            TIE Number: 536871234
            Sequence Number: 12108
            Remaining lifetime: 604800
```

```
▾ Routing in Fat Trees
  ▾ Outer Security Envelope Header
      RIFT Magic: 0xa1f7
      Packet Number: 1922
      Reserved
      RIFT Major Version: 2
      Outer Key ID: 0
      Security Fingerprint Length: 0
      Weak Nonce Local: 45264
      Weak Nonce Remote: 49140
  ▾ Serialized RIFT Model Object
      Protocol major version: 2
      Protocol minor version: 0
      Sender: 7951891422087488001
      Level: 0
    ▾ Topology Information Description Element (TIDE)
      ▾ Start range
          Direction: South (1)
          Packet Originator: 0
          TIE Element Type: Node (2)
          TIE Number: 0
      ▾ End range
          Direction: North (2)
          Packet Originator: 18446744073709551615
          TIE Element Type: Key Value (7)
          TIE Number: 4294967295
      ▾ List of headers
        ▸ TIE Header with lifetime 600959
        ▸ TIE Header with lifetime 600958
        ▸ TIE Header with lifetime 600959
        ▸ TIE Header with lifetime 600959
        ▸ TIE Header with lifetime 600950
```

Figure 5.6 TIDE Packets

Chapter 6

Open Source RIFT Implementation

6.1 History and Current State

RIFT-Python is an open source implementation of the RIFT protocol released under the permissive Apache 2.0 license (https://www.apache.org/licenses/LICENSE-2.0). The source code is publicly available in the GitHub repository https://github.com/brunorijsman/rift-python.

The project has its roots in a hackathon that took place at the 101st meeting of the Internet Engineering Task Force (IETF) in July of 2018 in Montreal. At that time RIFT was still in the early stages of being standardized. Eight of us got together and spent a weekend hacking together a Python implementation of the link information element (LIE) finite state machine (FSM) in RIFT. Our goal was to find out whether the RIFT specification was clear enough to allow two independent teams of developers to implement interoperable versions of RIFT, or at least two interoperable versions of the LIE FSM.

Once we finished the code, we did interoperability testing with an early prototype of the Juniper RIFT code, which was also publicly available for alpha testing at the time. The hackathon was a lot of fun (you should definitely attend a hackathon if you haven't already) and it achieved its goals: not only were we able to bring up an adjacency to state three-way, thus proving interoperability between the Juniper and the RIFT-Python code, but we also learned several useful lessons.

One lesson learned was that using a Thrift model to model the protocol messages really paid off. It took us just a few hours to write the code to encode and decode RIFT protocol messages. That's because the Thrift compiler did most of the heavy lifting for us by generating all the Python data structures and the Python code to encode and decode those data structures to on-the-wire binary messages.

Not only did using a Thrift model save us a lot of work, it also made it much more likely that the encoding and decoding functions were actually correct. In traditional routing protocols, where the message format is specified using English text and ASCII diagrams, the encoding and decoding functions are often a rich source of bugs and security issues (see, for example, CVE-2018-5381 (https://nvd.nist.gov/vuln/detail/CVE-2018-5381)).

We did find a few issues during the interoperability testing. One issue was caused by the lack of unsigned integers in Thrift combined with differences in how different programming languages represent large integers. Another issue we discovered was that having three RIFT routers connected to the same Ethernet broadcast domain (something that is explicitly not allowed but could accidentally happen anyway) caused the LIE FSM to get stuck in a loop sending RIFT LIE messages at line rate. This was fixed by introducing a new MultipleNeighborsWait state to the FSM.

A detailed report of this first hackathon can be found in the hackathon PowerPoint presentation and report https://github.com/brunorijsman/rift-python/tree/master/ietf-102/.

Ever since that hackathon in 2018, I (Bruno Rijsman) continued working on RIFT-Python on-and-off as a personal side project. I was fascinated by the new ideas in RIFT such as combining link-state with distance vector, using model-based encoding, and negative disaggregation. Also, I was looking for a hands-on project that would keep both my coding skills and my networking skills from atrophying. At the time I was on a long sabbatical, taking a break from the corporate world and spending most of my time hiking in remote mountains (https://hikingandcoding.wordpress.com/).

Figure 6.1 Himlung Base Camp, Himalaya, Nepal

The teams working on the IETF specification and on the Juniper implementation were keenly interested in a second implementation to improve the quality of the IETF draft and to discover potential interoperability issues early on.

My original goal was not to produce a commercial-grade implementation of RIFT that could be used on routers in large scale data centers. Instead, the main goals were:

1. To prove that IETF specification was sufficiently clear to allow independent software teams (in this case Juniper and I) to develop interoperable implementations.

2. To create a freely available open source reference implementation that can be used by anyone (for example, academics) to gain understanding of and experience with RIFT.

The RIFT-Python implementation has neither been designed for nor tested in very large topologies with hundreds or thousands of routers.

In retrospect, the RIFT-Python project has gone much further than I ever imagined. By now (late 2020) the code has evolved to the point that it is around 90% feature complete. RIFT-Python is now usable as a host-based RIFT leaf router or even a non-leaf RIFT router in small to medium topologies.

At this point some features such as mobility, anycast, policy, key-value processing, and SSH support have not yet been implemented. See the RIFT-Python features list (https://github.com/brunorijsman/rift-python/blob/master/doc/features.md) and the issue list on GitHub (https://github.com/brunorijsman/rift-python/issues) for a more complete and up-to-date view of what's present and what's missing. And certainly a lot more testing would be needed before it could be deployed in production.

In early 2019 there were even some discussions about porting RIFT-Python to the C programming language and integrating it into Free Range Routing (FRR) (https://frrouting.org/), but that project has not materialized for various reasons.

6.2 Installation

RIFT-Python has been tested on Ubuntu 16.04 LTS (Xenial), Ubuntu 18.04 LTS (Bionic), Ubuntu 20.04 LTS (Focal), macOS 10.14 (Mojave), and macOS 10.15 (Catalina). It should be possible to run RIFT-Python on versions of Linux or macOS or even on other platforms (such as Windows) with little or no porting, but we have not tested that. Most code is very portable, except for the code that deals with sending multicast and IPv6 UDP packets over sockets, which is infuriatingly platform dependent in subtle ways.

As the name suggests, RIFT-Python has been implemented in Python. It has been tested on Python 3.5, 3.6, 3.7 and 3.8. It cannot run on any version of Python 2.

Here we describe how to install RIFT-Python on an Amazon Web Services (AWS) Elastic Compute Cloud (EC2) instance running Ubuntu 20.04 LTS. These instructions should also work for an Ubuntu 20.04 LTS server running on bare metal or in a locally hosted virtual machine or in a container.

Using the AWS console (or CLI or API) create an EC2 instance:

- Choose Amazon Machine Image (AMI) "Ubuntu Server 20.04 LTS (HVM), SSD Volume Type, 64-bit (x86)".
- Instance type t2.micro (which is eligible for the AWS free tier) is large enough for small topologies. Large topologies require an instance type with more CPU and memory.
- Accept the default values for all other configuration parameters.
- Make sure you download the private key for the EC2 instance and store it locally (these instructions assume you store it in `~/.ssh/private-key-file.pem`).
- Make a note of the IP address `vm-ip-address` of your newly created instance, which is reported in the AWS console.

Log in to Ubuntu, using user name `ubuntu` and using the private key that you downloaded from AWS:

```
$ ssh -i ~/.ssh/private-key-file.pem ubuntu@vm-ip-address
```

The above command assumes you are logging in from a platform (such as Linux or macOS) that supports a command-line SSH client. If you are logging in from Windows you may have to download a Windows SSH client such as Putty.

Once logged in to the EC2 instance, install the latest security patches on your EC2 instance by doing an update:

```
$ sudo apt-get update
```

The AWS Ubuntu 20.04 AMI comes pre-installed with Python 3.8:

```
$ python3 --version
Python 3.8.5
```

However, if you need to install Python 3 yourself you can do so as follows:

```
$ sudo apt-get install -y python3    >>> This step is not needed on AWS
```

RIFT-Python itself is written entirely in Python and does not contain any C or C++ code. However, you must install a C compiler because it is needed for the `pytricia` dependency, which is partly written in C:

```
$ sudo apt-get install -y build-essential
```

The pytricia dependency also needs the header files for the Python 3 source code:

```
$ sudo apt-get install -y python3-dev
```

The RIFT-Python code is stored in GitHub, so you need `git` to clone the repository. Git comes pre-installed on the AWS Ubuntu AMI:

```
$ git --version
git version 2.25.1
```

However, if you need to install git yourself you can do so as follows:

```
$ sudo apt-get install -y git    >>> This step is not needed on AWS
```

Use `git` to clone the RIFT-Python repository from GitHub into the Ubuntu instance. These instructions assume that you run the clone command from your home directory:

```
$ git clone https://github.com/brunorijsman/rift-python.git
```

If all went well, you should now have directory `~/rift-python` that contains the RIFT-Python code:

```
$ find rift-python
rift-python
rift-python/tests
rift-python/tests/test_rib_fib.py
rift-python/tests/test_visualize_log.py
rift-python/tests/test_sys_2n_un_l0.py
rift-python/tests/test_sys_2n_l0_l2.py
...
```

Enter the directory that contains the RIFT-Python code:

```
$ cd rift-python
```

If you want to be 100% sure that all of the examples given later in this chapter work exactly as described, you must check-out the version of the RIFT-Python code as it was in November 2020 when this book was written:

```
$ git checkout tags/day-one-book-20201102   >>> Skip this if you want latest code
```

On the other hand, if you want the latest and greatest version of RIFT-Python with the most recent bug fixes and features, then skip the above command. But then the CLI commands and output might be slightly different from what is described in this book (the online documentation will be up to date).

A Python virtual environment is a mechanism to keep all project dependencies together and isolated from the dependencies of other projects you may be working on to avoid conflicts.

Install `virtualenv` into the Ubuntu instance:

```
$ sudo apt-get install -y virtualenv
```

Create a Python 3 virtual environment called `env` (make sure you are still in the `~/rift-python` directory):

```
$ virtualenv env --python=python3
```

Activate the Python virtual environment. You will know that the virtual environment is active because your command line prompt contains the word (env):

```
$ source env/bin/activate
(env) $
```

In AWS the command line prompt is actually a bit longer, but it's abbreviated here to improve readability of these instructions. This is what the actual prompt looks like (except that the IP address will be different):

```
(env) ubuntu@ip-172-31-30-251:~/rift-python$
```

Use the package installer for Python (pip) to install the dependencies for RIFT-Python. Make sure you have activated the virtual environment as described in the previous step before you install these dependencies:

```
(env) $ pip install -r requirements-3-8.txt
```

If you are using Python 3.5, 3.6 or 3.7 instead of Python 3.8 (which is the default for Ubuntu 20.04), you should use requirements file requirements-3-567.txt instead.

Congratulations! You have installed RIFT-Python. You can verify that the installations was successful using the following command:

```
(env) $ python rift -i topology/one.yaml
node1> exit
(env) $
```

If you see the node1> prompt it means that you have successfully started a RIFT router and connected to its command line interface (CLI). Use the exit command to exit the CLI and return to the Ubuntu shell.

Every time you login in on a new terminal session, you must re-activate your virtual environment using the source env/bin/activate command. If you forget to do so, you will see error messages similar to the following.

```
$ python rift -i topology/one.yaml
Command 'python' not found, did you mean:
  command 'python3' from deb python3
  command 'python' from deb python-is-python3
```

Or possibly something similar to:

```
$ python rift -i topology/one.yaml
Traceback (most recent call last):
  File "/usr/lib/python3.8/runpy.py", line 194, in _run_module_as_main
    return _run_code(code, main_globals, None,
  File "/usr/lib/python3.8/runpy.py", line 87, in _run_code
    exec(code, run_globals)
  File "rift/__main__.py", line 5, in <module>
    import config
  File "rift/config.py", line 6, in <module>
    import cerberus
ModuleNotFoundError: No module named 'cerberus'
```

If the Ubuntu shell prompt does not include the word (env) it means that you forgot to activate the virtual environment.

Recommend using the screen (https://www.howtoforge.com/linux_screen) command to keep your shell environment intact and possibly keep your RIFT topology running, even when your SSH session is disconnected from the AWS instance.

The following command creates a new screen session. In this example the name of the session is rift, but you can name it whatever you want.

```
$ screen -S rift
```

Since this starts a new shell, you have to reactivate the Python virtual environment:

```
$ cd ~/rift-python
$ source env/bin/activate
(env) $
```

Now you can start RIFT-Python in the screen session:

```
(env) $ python rift -i topology/one.yaml
node1>
```

Pressing Ctrl-a d to detach from the screen session; this brings us back to the shell in the AWS instance:

```
node1> Ctrl-a d
$
```

The RIFT-Python topology is still running in the screen session even though you don't see its output because we are detached from the screen session.

You can verify that the screen session still exists:

```
$ screen -ls
There is a screen on:
            8230.rift (06/11/2020 06:57:10 AM) (Detached)
1 Socket in /var/run/screen/S-ubuntu.
```

At this point it is safe to log out of the SSH session to the AWS instance. As long as the AWS instance keeps running, the screen session will keep running as well, even when you are not logged in.

The following command re-attaches to the running screen session:

```
$ screen -r -S rift
(env) $ python rift -i topology/one.yaml
node1>
```

After re-attach, you'll see the exact same screen that you left behind just before detaching. If RIFT-Python had produced any output while being detached, you would see it as well.

6.3 Starting a Topology

You can run RIFT-Python as an individual RIFT node, running on bare metal hardware or in a VM or in a container. RIFT-Python also supports running topologies with multiple RIFT nodes. This allows you to study how RIFT behaves in real-life scenarios in a very simple manner. Such multi-node topologies can even contain a mixture of RIFT-Python and Juniper RIFT nodes for interoperability testing (see section 6.6.3).

There are two different approaches for running multi-node topologies: one approach is simple but only works well for small topologies; the other approach is more complex and supports larger topologies. Table 6.1 describes the differences between the two approaches.

Table 6.1 Different Approaches for Running Multi-node Topologies

Approach	Single-process approach.	Multi-process approach (also known as network namespace per node).
Topology size	Small topology.	Large topology.
Python processes	One single Python process for all RIFT nodes combined.	Each RIFT node runs in its own separate Python process in its own network namespace.
Node-to-node links	All links run over one single physical interface, using different UDP ports and multicast addresses to separate the simulated links from one another.	Each node-to-node link is implemented as a virtual ethernet (veth) pair between network namespaces.
Configuration	You manually write the topology file that contains the configuration for each RIFT node and that describes how the nodes are connected to each other.	The `config_generator` tool takes a meta-topology (a high-level description of the shape of the fabric) as input and automatically generates a configuration file for each RIFT node in the topology.
Pros	Simple to use and debug. Works both on Linux and on macOS.	Scales up to large and complex topologies because each node runs in a separate Python process and can potentially run on a separate CPU core. More realistic. Chaos testing requires the multi-process approach.
Cons	Does not scale up to large topologies because only one CPU core is used for all nodes.	More complex to use and debug. Only works on Linux; does not work on macOS.

6.3.1 Starting a Small Topology: the Single-process Approach

For small topologies, it is convenient to run all RIFT-Python nodes in a single Python process. Because everything runs in a single Python process, it is operationally very simple to start and stop the whole topology, to access the command line interface of each RIFT node, and to attach a Python debugger. Also, this method works on both macOS and Linux. This allows for a rapid code/test/fix development cycle using only a laptop, even when disconnected from the Internet (a common situation for me while traveling).

The downside of the single-process approach is that the single Python process runs single-threaded. This means that all nodes in the topology share a single CPU core. If you have a powerful multi-core server, all the extra CPU capacity will go to waste. Hence, the single-process approach is not suitable for large topologies. That said, in practice, I have been able to run topologies with 10 to 20 RIFT nodes on a very under-powered 2016 MacBook Air using the single-process approach without any problems.

In the single-process approach, all the node-to-node links are simulated using only a single physical interface. Traffic on one simulated link is separated from traffic over another simulated link by using different UDP ports and different multicast addresses for each simulated link, as shown in Figure 6.2.

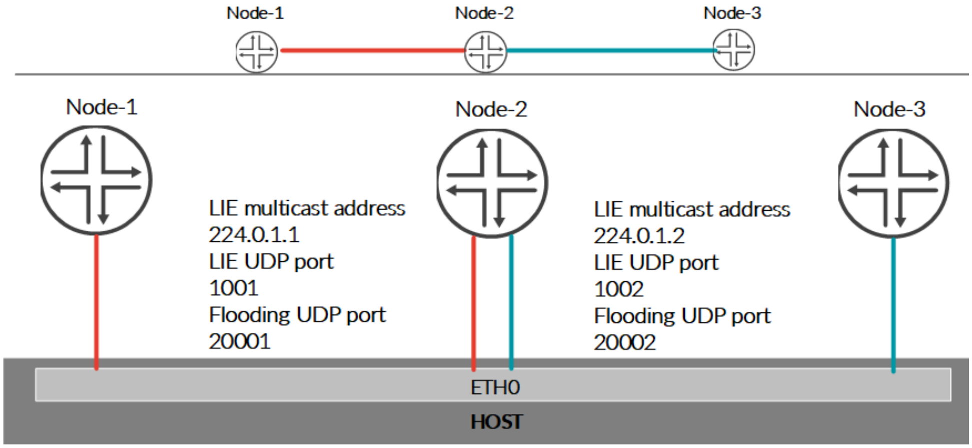


Figure 6.2 *Node-to-node Links Are Simulated Using Only a Single Physical Interface*

You can see the single-process approach uses different multicast addresses and UDP ports to logically separate multiple simulated links on a single physical interface.

This trick of using different UDP ports and multicast addresses is also useful for real deployments when running multiple instances of RIFT on a single device.

To start a topology you first need a configuration file, also known as a *topology file*, that describes which RIFT nodes are present in the topology, how they are connected to each other, and what the configuration parameters for each RIFT node are.

The configuration file for a single stand-alone RIFT node that runs in a real network looks exactly the same as a configuration file for a simulated topology with multiple RIFT nodes, except that:

1. It contains only a single RIFT node as opposed to multiple RIFT nodes.

2. It contains simulated interface names as opposed to real interface names.

The rest of this chapter will use the term *topology file* when a single RIFT-Python process runs multiple nodes (the single-process approach) and the term *configuration file* when each node runs in its own RIFT-Python process (the multi-process approach). But keep in mind that topology files and configuration files use the same syntax and are really the same thing used in different contexts.

We will describe how to create a topology file in the next section. For now, just use an existing example topology `topologies/2n_l0_l1.yaml`, which is an extremely simple topology with only two RIFT nodes: node1 which is a spine (level 1) node and node2 which is a leaf (level 0) node.

Once you have a topology file, use the following steps to start a RIFT-Python process (also known as a *RIFT-Python engine*) running that topology.

Make sure that you are in the root directory of the git repository:

```
$ cd ~/rift-python
```

Make sure that your virtual environment is activated: look for `(env)` in the shell prompt:

```
$ source env/bin/activate
(env) $
```

Run the `rift` Python module to start the Python process. This single Python process runs all the RIFT nodes in the topology:

```
(env) $ python rift --interactive topology/2n_l0_l1.yaml
node1>
```

The `node1>` prompt indicates that you are currently attached to the command line interface of the RIFT-Python process and that the current RIFT node is `node1`.

The `--interactive` option (which can be abbreviated to `-i`) means that you immediately attach to the command interface (CLI), as opposed to using Telnet to access the CLI.

If you want to run RIFT-Python as a daemon, leave out the `--interactive` option and run it in the background by adding an ampersand at the end:

```
(env) $ python rift topology/2n_l0_l1.yaml &
[1] 19225
Command Line Interface (CLI) available on port 34009
(env) $
```

RIFT-Python reports a port number that you can use to Telnet into the CLI. RIFT-Python currently does not support SSH. In this example the port number is 34009, which you can connect to using the following telnet command:

```
$ telnet localhost 34009
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
node1>
```

MORE? The full set of Python RIFT command line options is documented at https://github.com/brunorijsman/rift-python/blob/master/doc/command-line-options.md and summarized in the Table 6.2.

Table 6.2 A Summary of Python RIFT Commands

`--help (-h)`	Display help.

`--passive (-p)`
`--non-passive (-n)`
	Run only the nodes that are marked as passive (or non-passive) in the topology.

`--log-level (-l)` *`level`*	Sets the minimum log level for log messages. Allowed values are `debug`, `info`, `warning`, `error`, and `critical`. RIFT-Python writes all log messages from all nodes and all categories to a single log file `rift.log` in the directory where RIFT is started.
`--interactive (-i)`	Runs RIFT in interactive mode as explained above.
`--telnet-port-file` *`file-name`*	When RIFT is not run in interactive mode, this option writes the Telnet port to the specified file. This is used by the chaos testing scripts described in section 6.6.2, *Automated Chaos Testing*.
`--ipv4-multicast-loopback-disable` `--ipv6-multicast-loopback-disable`	In the multi-process approach, RIFT-Python maps multiple simulated interfaces to a single physical interface on the host operating system, as shown in Figure 6.2. By default, RIFT-Python enables multicast loopback on the UDP sockets to loopback the sent RIFT UDP packets. Some cheap consumer Wifi routers incorrectly send received IP multicast packets back to the source. In that case, to prevent RIFT-Python from receiving two copies of each sent RIFT packet, you must disable UDP socket loopback using these command-line options.

If you start a large RIFT topology, you may see the following error message because the host operating system runs out of file descriptors. This is especially common when running large topologies on macOS.

```
OSError: [Errno 24] Too many open files
```

You can fix this by using the `ulimit` command to increase the number of available file descriptors, for example:

```
(env) $ ulimit -S -n 1024
```

However, in my experience, macOS tends to become unstable (applications freeze, the kernel panics) when you increase the number of file descriptors by too much. I would not recommend going over 1024. For very large topologies, Linux is better.

Later on an overview of the operational commands is provided that are available in the CLI. For now, let's just issue a few simple commands to give you an idea of what it looks like and to make sure everything works as expected.

Use the `show nodes` command to see which RIFT nodes are present in the topology:

```
node1> show nodes
+-------+--------+---------+
| Node  | System | Running |
| Name  | ID     |         |
+-------+--------+---------+
| node1 | 1      | True    |
+-------+--------+---------+
| node2 | 2      | True    |
+-------+--------+---------+
```

Use the `show interfaces` command to display the interfaces of the current RIFT node (i.e. node1). In this case there is only one interface named if1 which is connected to node2:if1 (i.e. interface if1 on node2).

```
node1> show interfaces
+-----------+-----------+-----------+-----------+-------------------+-------+
| Interface | Neighbor  | Neighbor  | Neighbor  | Time in           | Flaps |
| Name      | Name      | System ID | State     | State             |       |
+-----------+-----------+-----------+-----------+-------------------+-------+
| if1       | node2:if1 | 2         | THREE_WAY | 0d 00h:00m:47.86s | 0     |
+-----------+-----------+-----------+-----------+-------------------+-------+
```

Use the `exit` command to disconnect from the command line interface but leave RIFT-Python running in the background:

```
node1> exit
Connection closed by foreign host.
(env) $
```

Or, use the `stop` command to disconnect from the command line interface and also terminate the RIFT-Python process.

```
node1> stop
Connection closed by foreign host.
(env) $
```

NOTE There is only a difference between `exit` and `stop` if you started RIFT-Python in the background using the non-interactive mode. In interactive mode, `exit` and `stop` both terminate the RIFT-Python process.

6.3.2 Starting a Large Topology: the Multi-process Approach

For large topologies, it is not feasible to run all RIFT nodes in a one single-threaded Python process. You need to run each RIFT node in its own Python process so that you make use of all available CPU cores. This is exactly what the multi-process approach does: it runs each RIFT node in its own separate Python process.

Furthermore, in the multi-process approach, each RIFT node process runs in a separate network namespace. Namespaces as a general concept are a Linux feature that allow you to create multiple isolated instances of particular subsystems in the Linux kernel. Namespaces are one of the foundational technologies that are used to implement containers in Linux. There are multiple types of namespaces. The network namespace (which RIFT-Python uses) allows you to create multiple instances of the TCP/IP stack that are isolated from each other, each with its own set of interfaces and its own route table. The Process ID (PID) namespace is another example of a namespace; each PID namespace has its own set of processes with corresponding PIDs, that are isolated from and hidden from other PID namespaces.

The RIFT nodes are connected to each other using virtual Ethernet (veth) interfaces, which are a special type of interface in Linux. Virtual Ethernet interfaces are always created in pairs, let's say veth0 and veth1. One veth interface in the pair is connected to the other using a virtual (i.e. 'fake') Ethernet connection. When a program sends an IP packet into interface veth0 it pops out of veth1, and vice versa.

Typically, one veth interface of the pair (say veth0) is placed in one network namespace, and the other veth interface of the pair (say veth1) is placed in another network namespace. This creates a virtual Ethernet link between the two network namespaces, as shown in Figure 6.3.

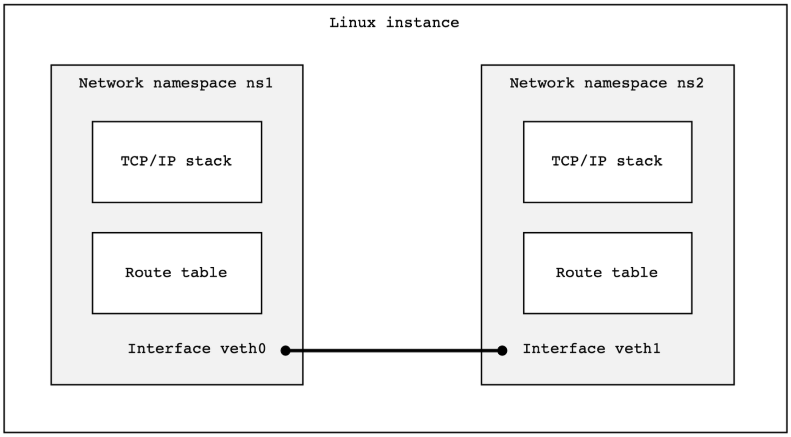


Figure 6.3 Network Namespaces and Virtual Ethernet (veth) Interfaces.

Network namespaces and virtual Ethernet interfaces are Linux features that are not natively available in macOS. As a result, the multi-process approach for running large topologies described in this chapter is only supported on Linux and not on macOS (unless you run Linux in a virtual machine or in a container on macOS).

In the single-process approach described in section 6.3.1 we had a single configuration file, known as the topology file, that describes all RIFT nodes in the entire topology. In the multi-process approach described in this chapter, there is a separate configuration file for each RIFT node in the topology.

Hypothetically, it would be possible to manually create the configuration file for each RIFT node in the topology, and to manually issue all the Linux shell commands to create the necessary network namespaces, to create the necessary virtual ethernet interfaces, to start the necessary processes, and so forth. In practice that is wildly impractical and error-prone, especially if there are many nodes, i.e. if the topology is large.

For that reason, RIFT-Python includes a configuration generation script called `config_generator.py`.

The input to the configuration generator is a so-called *meta-topology file*, which describes the entire fabric at a very high level of abstraction. We will describe the syntax of the meta-topology file in detail in section 6.4.2. For now, here is an example, just to give you an idea of what it looks like (notice how tiny it is):

```
nr-pods: 3
nr-leaf-nodes-per-pod: 3
nr-spine-nodes-per-pod: 3
nr-superspine-nodes: 6
nr-planes: 3
```

The configuration generator reads the meta-topology file as input and produces the following files as output:

1. A separate configuration file `node-name.yaml` for each RIFT router.

2. A `start.sh` shell script to start the entire topology. It creates all the network namespaces, creates all the veth interface pairs, assigns IP addresses to the veth interfaces, puts the veth interfaces into the right namespace, and starts the Python process for each RIFT node running in the background.

3. One `connect-node-name.sh` shell script for each RIFT node, to telnet into that node.

4. A `stop.sh` shell script to stop the entire topology. It stops all RIFT processes and cleans up all the veth interfaces and network namespaces.

5. A `check.sh` shell script, which verifies that everything has converged properly by doing a ping from every leaf node to every other leaf node. Note that the `config_generator.py` tool has a `--check` option that does a much more elaborate convergence test than this very simple `check.sh` script.

6. A `chaos.sh` shell script, which is used for chaos testing (see section 6.6.2). It randomly breaks and repairs 'stuff' (nodes, links, etc.) in the network and makes sure that RIFT reconverges properly at the end.

7. An `allocations.txt` file that describes which IP address has been assigned to which interface on which RIFT node.

Use the following command to invoke the configuration generator for the example meta-topology file `clos_3plane_3pod_3leaf_3spine_6super.yaml`. (This example file is included in the rift-python GitHub repository and its contents were shown above.)

```
$ cd ~/rift-python
$ source env/bin/activate
(env) $ tools/config_generator.py --graphics-file diagram.html --netns-per-node meta_topology/clos_3plane_3pod_3leaf_3s
pine_6super.yaml generated-config
```

`Generated-config` is the name of the directory into which all generated scripts are written. This directory must not already exist.

After the `config_generator` has run, directory `generated-config` will contain the following files:

```
(env) $ ls -1 generated-config
allocations.txt
chaos.sh
check.sh
connect-leaf-1-1.sh
connect-leaf-1-2.sh
...
connect-super-3-2.sh
leaf-1-1.yaml
leaf-1-2.yaml
...
```

```
spine-3-3.yaml
start.sh
stop.sh
super-1-1.yaml
super-1-2.yaml
...
super-3-2.yaml
```

The `--netns-per-node` option (`-n` for short) indicates that you want `config_generator` to generate scripts for the multi-process approach, therefore to create a separate network namespace for each RIFT node. If you omit this option, `config_generator` will generate a single configuration file to be used for the single-process approach described in section 6.3.1.

The `--graphics-file diagram.html` option (`-g` for short) writes a scalable vector graphics (SVG) file containing a diagram of the generated topology to the file `diagram.html`. You can view this file using any browser. On macOS you can use the following command to view the diagram:

```
$ open diagram.html
```

In this example the topology now looks like Figure 6.4

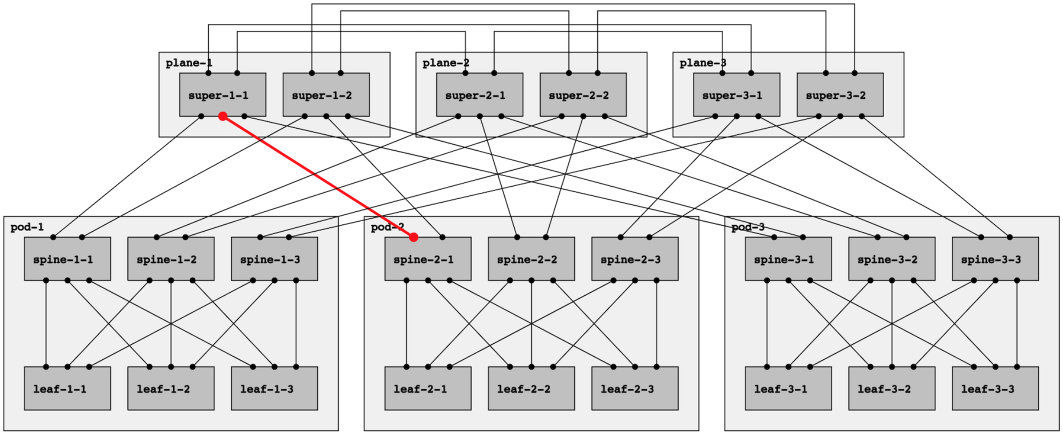


Figure 6.4 *Topology Diagram.*

You can hover over links or interfaces or nodes in the topology to highlight that particular element in red.

Notice that indeed there are three points of delivery (PoDs), each with three leaf nodes and three spine nodes. There are six superspine nodes, spread over three planes that are interconnected using east-west inter-plane links.

Before starting this topology you first need escalated privileges to create the network namespaces. The easiest way (although perhaps not the most security conscious way) to achieve this is to create a new bash shell as root:

```
(env) $ sudo bash
#
```

Since you are in a new shell, you have to reactivate the Python virtual environment:

```
# source env/bin/activate
(env) #
```

Now, at last, you can actually start the topology by invoking the `start.sh` script:

```
(env) # ./generated-config/start.sh
Create veth pair veth-1a-101d and veth-101d-1a for link from super-1-1:if-1a to spine-1-1:if-101d
Create veth pair veth-1b-104d and veth-104d-1b for link from super-1-1:if-1b to spine-2-1:if-104d
...
Create veth pair veth-1009c-109c and veth-109c-1009c for link from leaf-3-3:if-1009c to spine-3-3:if-109c
Create network namespace netns-1 for node super-1-1
Create network namespace netns-2 for node super-1-2
...
Create network namespace netns-109 for node spine-3-3
Start RIFT-Python engine for node super-1-1
Start RIFT-Python engine for node super-1-2
...
Start RIFT-Python engine for node spine-3-3
(env) #
```

After the `start.sh` script has completed, you are back in the Linux shell and all RIFT nodes are running in the background as separate processes.

Use one of the `connect-node-name.sh` scripts to Telnet to the CLI on the RIFT node with name *node-name*. For example, to connect to the CLI of superspine node super-1-1:

```
(env) #
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
super-1-1>
```

Once you are connected to node super-1-1 you can see that super-1-1 has three southbound interfaces to spine nodes and two east-west interfaces to other superspine routers for inter-plane links:

```
super-1-1> show interfaces
+--------------+------------------------+-----------+-----------+-------------------+-------+
| Interface    | Neighbor               | Neighbor  | Neighbor  | Time in           | Flaps |
| Name         | Name                   | System ID | State     | State             |       |
+--------------+------------------------+-----------+-----------+-------------------+-------+
| veth-1a-101d | spine-1-1:veth-101d-1a | 101       | THREE_WAY | 0d 00h:00m:14.89s | 0     |
+--------------+------------------------+-----------+-----------+-------------------+-------+
| veth-1b-104d | spine-2-1:veth-104d-1b | 104       | THREE_WAY | 0d 00h:00m:11.68s | 0     |
+--------------+------------------------+-----------+-----------+-------------------+-------+
| veth-1c-107d | spine-3-1:veth-107d-1c | 107       | THREE_WAY | 0d 00h:00m:09.87s | 0     |
+--------------+------------------------+-----------+-----------+-------------------+-------+
| veth-1d-3d   | super-2-1:veth-3d-1d   | 3         | THREE_WAY | 0d 00h:00m:20.89s | 0     |
+--------------+------------------------+-----------+-----------+-------------------+-------+
| veth-1e-5e   | super-3-1:veth-5e-1e   | 5         | THREE_WAY | 0d 00h:00m:19.84s | 0     |
+--------------+------------------------+-----------+-----------+-------------------+-------+
```

Use `exit` (not `stop`) to return to the Linux shell and leave the router running in the background:

```
super-1-1> exit
Connection closed by foreign host.
(env) #
```

Next, run the script check.sh to perform a sanity check to verify that RIFT has properly converged. This script pings every leaf router from every other leaf router:

```
(env)# ./generated-config/check.sh
*** ping ***
OK: ping leaf-1-1 88.0.1.1 -> leaf-1-2 88.0.2.1
OK: ping leaf-1-1 88.0.1.1 -> leaf-1-3 88.0.3.1
OK: ping leaf-1-1 88.0.1.1 -> leaf-2-1 88.0.4.1
...
OK: ping leaf-3-3 88.0.9.1 -> leaf-3-1 88.0.7.1
OK: ping leaf-3-3 88.0.9.1 -> leaf-3-2 88.0.8.1
Number of failures: 0
(env) #
```

Yay! Everything has converged properly. Later on, when we discuss chaos testing in section 6.6.2 we'll describe a much more sophisticated method for checking whether the network converged correctly.

To figure out which nodes the IP addresses in the above ping output correspond to, have a look at the generated file `allocations.txt`: it summarizes the names and IP addresses of all interfaces and nodes in the topology:

```
(env) # cat ./generated-config/allocations.txt
+-----------+----------+--------+------------+-----------+-------------------+-----------+-------------------+
| Node      | Loopback | System | Network    | Interface | Interface         | Neighbor  | Neighbor          |
| Name      | Address  | ID     | Namespace  | Name      | Address           | Node      | Address           |
+-----------+----------+--------+------------+-----------+-------------------+-----------+-------------------+
| leaf-1-1  | 88.0.1.1 | 1001   | netns-1001 | if-1001a  | 99.1.2.1/24       | spine-1-1 | 99.1.2.2/24       |
|           |          |        |            | if-1001b  | 99.3.4.3/24       | spine-1-2 | 99.3.4.4/24       |
|           |          |        |            | if-1001c  | 99.5.6.5/24       | spine-1-3 | 99.5.6.6/24       |
+-----------+----------+--------+------------+-----------+-------------------+-----------+-------------------+
.           .          .        .            .           .                   .           .                   .
+-----------+----------+--------+------------+-----------+-------------------+-----------+-------------------+
| super-3-2 | 88.2.6.1 | 6      | netns-6    | if-6a     | 99.85.86.85/24    | spine-1-3 | 99.85.86.86/24    |
|           |          |        |            | if-6b     | 99.87.88.87/24    | spine-2-3 | 99.87.88.88/24    |
|           |          |        |            | if-6c     | 99.89.90.89/24    | spine-3-3 | 99.89.90.90/24    |
|           |          |        |            | if-6d     | 99.99.100.100/24  | super-2-2 | 99.99.100.99/24   |
|           |          |        |            | if-6e     | 99.101.102.101/24 | super-1-2 | 99.101.102.102/24 |
+-----------+----------+--------+------------+-----------+-------------------+-----------+-------------------+
```

Finally, use the `stop.sh` script to stop all RIFT nodes and to clean up all network namespaces, veth interfaces, etc.:

```
(env)# ./generated-config/stop.sh
Stop RIFT-Python engine for node super-1-1
Delete interface veth-1a-101d for node super-1-1
Delete interface veth-1b-104d for node super-1-1
...
Delete network namespace netns-108 for node spine-3-2
Delete network namespace netns-109 for node spine-3-3
(env)#
```

NOTE For complete documentation of the `config_generator` script see https://github.com/brunorijsman/rift-python/blob/master/doc/configuration-file-generator.md.

6.4 Configuration

6.4.1 Configuration file (also known as topology file)

RIFT-Python is configured using a configuration file. It is currently not possible to configure nodes using the command line interface (CLI); the CLI is mainly used for operational (i.e. show) commands. If you want to reconfigure RIFT-Python, you must stop and restart the RIFT-Python process to force it to read the new configuration.

As you saw in section 6.3, there are two approaches for running a multi-node RIFT topology: the single-process approach and the multi-process approach.

In the single-process approach, shown in Figure 6.5, all RIFT nodes run in a single RIFT engine, i.e. in a single RIFT-Python process. There is one single configuration file for all nodes combined. In the single-process mode, we often use the term topology file instead of configuration file because it also describes how the nodes are connected to each other.

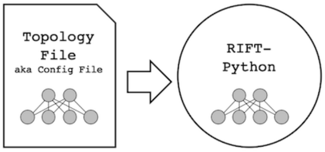


Figure 6.5 Topology File as Used in the Single-process Approach

In the multi-process approach, shown in Figure 6.6, each RIFT node runs in its own RIFT engine, i.e. in its own RIFT-Python process. There is a single so-called meta-configuration file that describes the shape of the fabric at a very high level of abstraction (e.g. how many PoDs, how many leaf and spine nodes per PoD, etc.) The `config_generator.py` tool takes the meta-topology file as input and produces a configuration file for each individual RIFT node as output. It also produces several scripts, for example to start and stop the topology.

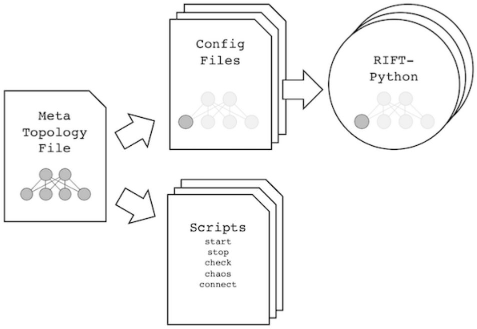


Figure 6.6 Topology File, as used in the Multi-process Approach

Regardless of whether the configuration file / topology file is used for the single-process approach or for the multi-process approach, it has the exact same syntax, namely a YAML (https://yaml.org/) syntax with the following structure.

Topology file / configuration file YAML schema:

```
shards:
  - id: int
    nodes:
      - name: string
        passive: bool
        level: int
        systemid: int
        rx_lie_mcast_address: ipv4addr
        tx_lie_mcast_address: ipv4addr
        rx_lie_v6_mcast_address: ipv6addr
        tx_lie_v6_mcast_address: ipv6addr
        rx_lie_port: portnr
        tx_lie_port: portnr
        rx_tie_port: portnr
        flooding_reduction: bool
        flooding_reduction_redundancy: int
        flooding_reduction_similarity: int
        kernel_routing_table: string
        active_authentication_key: int
        accept_authentication_keys: list of int
        active_origin_authentication_key: int
        accept_origin_authentication_keys: list of int
        interfaces:
          - name: string
            metric: int
            bandwidth: int
            rx_lie_mcast_address: ipv4addr
            tx_lie_mcast_address: ipv4addr
            rx_lie_v6_mcast_address: ipv6addr
            tx_lie_v6_mcast_address: ipv6addr
            rx_lie_port: portnr
            tx_lie_port: portnr
            rx_tie_port: portnr
            active_authentication_key: int
            accept_authentication_keys: list of int
        v4prefixes:
          - address: ipv4addr
            mask: int
            metric: int
            tags: list of int
        v6prefixes:
          - address: ipv6addr
            mask: int
            metric: int
            tags: list of int
authentication_keys:
  - id: int
    algorithm: string
    secret: string
```

At a high level, the configuration file consists of a list of nodes. The term node is simply a synonym for router or switch. In multi-process topology files, the list contains multiple nodes. In single-process configuration files, the list contains a single node. For each node, there is a list of interfaces, a list of IPv4 prefixes, and a list of IPv6 prefixes.

NOTE The concept of shards was introduced by the Juniper RIFT implementation and is intended to support running a very large topology across multiple servers connected by VXLAN tunnels. RIFT-Python does not support the concept of shards; it uses a different mechanism for supporting large topologies (namely one namespace per node).

Table 6.3 The Meaning of the Node Attributes

Attribute	Meaning	Mandatory / Optional
name	The name of the node.	Mandatory.
passive	Used to mark which RIFT implementation this node runs during interoperability tests. See section 6.6.3 for more details. Value is true or false.	Optional, default value false.
level	The level of the node. Value is one of the following: 0, 1, 2, undefined, leaf, spine, or top-of-fabric.	Optional. If level is not configured, zero touch provisioning (ZTP) is used to automatically select the level. However, it is mandatory to configure the level for Top of Fabric (ToF) nodes, which is typically set to value top-of-fabric. This is needed to make ZTP work correctly (ZTP bootstraps from the ToF level).
systemid	The system identifier of the node. Value is an integer >= 0.	Optional. If not configured, a unique system identifier is automatically generated.
rx_lie_mcast_address	The IPv4 multicast address used to receive IPv4 LIE packets.	Optional. Default value is 224.0.0.120.
tx_lie_mcast_address	The IPv4 multicast address used to send IPv4 LIE packets.	Optional. Default value is 224.0.0.120.
rx_lie_v6_mcast_address	The IPv6 multicast address used to receive IPv6 LIE packets.	Optional. Default value is FF02::A1F7.
tx_lie_v6_mcast_address	The IPv6 multicast address used to send IPv6 LIE packets.	Optional. Default value is FF02::A1F7.
rx_lie_port	The UDP port number for receiving LIE packets.	Optional. Default value is 914 when RIFT-Python is running with root privileges, or 10000 if not.
tx_lie_port	The UDP port number for receiving LIE packets.	Optional. Default value is the same as rx_lie_port.
rx_tie_port	The UDP port number for receiving TIE packets.	Optional. Default value is 915 when RIFT-Python is running as root, or 10001 if RIFT-Python is not running as root.
flooding_reduction	Is flooding reduction enabled? Value is true or false.	Optional. Default value is true.
flooding_reduction_redundancy	The minimum number of flooding paths (factor R in the RIFT specification). Value is an integer >=1.	Optional. Default value is 2.

flooding_reduction_similarity	How much is the grandparent count allowed to differ for two parent nodes to be put in the same similarity group for flood leader election in flooding reduction? Value is an integer >= 0.	Optional. Default value is 2.
kernel_routing_table	The Linux routing table into which the routes are installed. Value is one of the following: an integer between 0 and 255 inclusive, local, main, default, unspecified, or none.	Optional. If the RIFT-Python process is running a single RIFT router, the default value is main. If the RIFT-Python process is running a topology (i.e. multiple RIFT routers) the default value is the node number (or none if the node number > 250).
active_authentication_key	The key that is used to sign the outer header for all sent packets. Value is an integer and must match the id of one of the keys in the authentication_keys configuration (see below).	Optional. If not configured, sent outer headers are not signed.
accept_authentication_keys	A list of additional keys that can also be used to check the signature of the outer header of all received packets, above and beyond the active key. This is to support key roll-overs. Value is a list of integers, where each integer must match the id of one of the keys in the authentication_keys configuration (see below).	Optional. If not configured, no additional keys are accepted (i.e. only the active key is accepted).
active_origin_authentication_key	The key that is used to sign the origin header for all originated TIEs. Value is an integer and must match the id of one of the keys in the authentication_keys configuration (see below).	Optional. If not configured, the origin header in originated TIEs is not signed.
accept_origin_authentication_keys	A list of additional keys that can also be used to check the signature of the origin header of received TIEs, above and beyond the active key. This is to support key roll-overs. Value is a list of integers, where each integer must match the id of one of the keys in the authentication_keys configuration (see below).	Optional. If not configured, no additional origin keys are accepted (i.e. only the active origin key is accepted).

The meaning of the interface attributes are described in Table 6.4. Note that some attributes (e.g. multicast addresses, port number, and authentication keys) can be configured at both the node level and the interface level. Configuring the attributes at the node level establishes a default value for all the interfaces in that node.

Table 6.4 Meaning of the Interface Attributes

Attribute	Meaning	Mandatory / Optional
name	The name of the interface. For real RIFT routers this is the name of a real Linux interface. For multi-process topologies, this is the name of a veth interface. For single-process topologies this can be any name chosen by the user.	Mandatory.
metric	The metric for the interface. Value is an integer >= 1.	Optional. Default value is 1.
bandwidth	The bandwidth of the interface in megabits per second. Value is an integer >= 1.	Optional. Default value is the actual bandwidth of the interface if it can be determined (Linux only) or 100 Mbps otherwise.
rx_lie_mcast_address	See corresponding node attribute for details.	
tx_lie_mcast_address		
rx_lie_v6_mcast_address		
tx_lie_v6_mcast_address		
rx_lie_port		
tx_lie_port		
rx_tie_port		
active_authentication_key		
accept_authentication_keys		

The meaning of the v4prefix and v6prefix attributes are listed in Table 6.5.

Table 6.5 Meaning of the v4prefix and v6prefix Attributes

Attribute	Meaning	Mandatory / Optional
address	The IPv4 or IPv6 address. Value is a dotted decimal IPv4 address or a hex IPv6 address.	Mandatory.
mask	The prefix length. Value is an integer >= 0.	Mandatory.
metric	The metric of the prefix. Value is an integer >= 1.	Optional. Default value is 1.
tags	The tags for the prefix. Value is a list of unsigned 64-bit integers.	Optional. Default is no tags.

The meaning of the authentication_keys attributes are listed in Figure 6.6.

Table 6.6 Meaning of Authentication_keys Attributes

Attribute	Meaning	Mandatory / Optional
id	A number (>=1) uniquely identifying the key within the scope of this node.	Mandatory.
algorithm	The message signing algorithm: hmac-sha-1, hmac-sha-224, hmac-sha-256, hmac-sha-384, hmac-sha-512, sha-1, sha-224, sha-256, sha-384, or sha-512.	Mandatory.
secret	The message signing secret (also known as key). Value is a string.	Mandatory.

As mentioned before, the topology file not only describes the configuration of each node, but also how the nodes are connected to one another, i.e. the topology. Two node interfaces are connected to each other if the tx_lie_port of one interface matches the rx_lie_port of the other interface. This method of specifying the topology is rather opaque and cumbersome because it puts the burden of manually allocating port numbers on the user. It exists for historical reasons and will likely be replaced by a more direct and convenient mechanism that automatically allocated port numbers in the future.

You may have noticed that rx_tie_port can be configured, but tx_tie_port cannot. The value for tx_tie_port is inferred from the value on rx_tie_point on the remote side of the link.

The directory `topology` in the GitHub repository contains several example topology files, which are used for automated unit testing and system test (see section 6.6.1). For example, topology file `topology/3n_l0_l1_l2.yaml` describes a very simple topology with three nodes as shown in Figure 6.7.

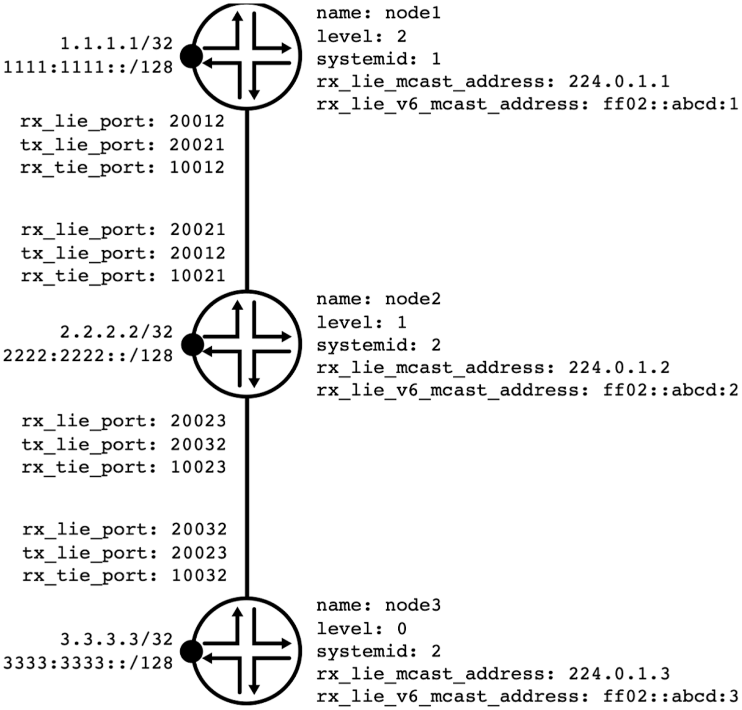


Figure 6.7 Topology/3n_l0_l1_l2.yaml Describes a Very Simple Topology with Three Nodes

```
# Example topology file topology/3n_l0_l1_l2.yaml
shards:
  - id: 0
    nodes:
      - name: node1
        level: 2
        systemid: 1
        rx_lie_mcast_address: 224.0.1.1
        rx_lie_v6_mcast_address: ff02::abcd:1
        interfaces:
          - name: if1 # Connected to node2-if1
            rx_lie_port: 20012
            tx_lie_port: 20021
            rx_tie_port: 10012
        v4prefixes:
          - address: 1.1.1.1
            mask: 32
            metric: 1
        v6prefixes:
          - address: "1111:1111::"
            mask: 128
            metric: 1
      - name: node2
        level: 1
        systemid: 2
        rx_lie_mcast_address: 224.0.1.2
        rx_lie_v6_mcast_address: ff02::abcd:2
        interfaces:
          - name: if1 # Connected to node1-if1
            rx_lie_port: 20021
            tx_lie_port: 20012
            rx_tie_port: 10021
          - name: if2 # Connected to node3-if1
            rx_lie_port: 20023
            tx_lie_port: 20032
            rx_tie_port: 10023
        v4prefixes:
          - address: 2.2.2.2
            mask: 32
            metric: 1
        v6prefixes:
          - address: "2222:2222::"
            mask: 128
            metric: 1
      - name: node3
        level: 0
        systemid: 3
        rx_lie_mcast_address: 224.0.1.3
        rx_lie_v6_mcast_address: ff02::abcd:3
        interfaces:
          - name: if1 # Connected to node2-if1
            rx_lie_port: 20032
            tx_lie_port: 20023
            rx_tie_port: 10032
        v4prefixes:
          - address: 3.3.3.3
            mask: 32
            metric: 1
        v6prefixes:
          - address: "3333:3333::"
            mask: 128
            metric: 1
```

6.4.2 Meta-topology Configuration File

As discussed in section 6.3.2, the meta-configuration file describes the topology of the fabric at a very high level of abstraction. The `config_generator.py` script takes the meta-topology file as input and produces a detailed configuration for each router and some associated scripts as output.

The meta-configuration file is a YAML file with the following structure:

```
nr-pods: int
nr-leaf-nodes-per-pod: int
nr-spine-nodes-per-pod: int
nr-superspine-nodes: int
nr-planes: int
inter-plane-east-west-links: bool
leafs:
  nr-ipv4-loopbacks: int
spines:
  nr-ipv4-loopbacks: int
superspines:
  nr-ipv4-loopbacks: int
leaf-spine-links:
  nr-parallel-links: int
spine-superspine-links:
  nr-parallel-links: int
inter-plane-links:
  nr-parallel-links: int
chaos:
  nr-link-events: int
  nr-node-events: int
  event-interval: float
  max-concurrent-events: int
```

The meaning of the meta-topology attributes are listed in Table 6.7.

Table 6.7 Meaning of Meta-topology Attributes

Attribute	Meaning	Mandatory / Optional
nr-pods	The number of points of deployment (PoDs). Value is an integer >= 1.	Optional, default value 1.
nr-leaf-nodes-per-pod	The number of leaf nodes per PoD. Value is an integer >= 1.	Mandatory.
nr-spine-nodes-per-pod	The number of spine nodes per PoD. Value is an integer >= 1.	Mandatory.
nr-superspine-nodes	The number of superspine nodes, also known as top-of-fabric (ToF) nodes. Value is an integer >= 1.	Optional. If not present, there are no superspine nodes (i.e. it is a two-level topology).
nr-planes	The number of planes in the superspine. Value is an integer >= 1.	Optional. If not present, there is a single plane.
inter-plane-east-west-links	In a multiple-plane topology, are there east-west inter-plane links between the top-of-fabric (i.e. superspine) nodes.	Optional, default value true.

Table 6.8 Meaning of Leaf, Spine, and Superspine Attributes

Attribute	Meaning	Mandatory / Optional
nr-ipv4-loopbacks	The number of IPv4 loopback interfaces on each node at the specified node level.	Optional, default value 1.

The attributes for the leaf-spine-links, spine-superspine-links, and inter-plane-links are listed in Table 6.9.

Table 6.9 *Meaning of Leaf-spine-link, spine-superspine-link, and inter-plane-link attributes*

Attribute	Meaning	Mandatory / Optional
nr-parallel-links	The number of parallel links between each pair of nodes for the specified type of link.	Optional, default value 1.

The chaos group of attributes will be explained in section 6.6.2 where we discuss automated chaos testing.

The `config_generator` currently only supports 2-level (= 3 stage Clos) or 3-level (= 5 stage Clos) topologies.

6.5 Operational Commands

So far, we have learned how to configure and how to start a stand-alone RIFT node or a multi-node RIFT topology. This section discusses how to use the command line interface (CLI) to issue operational commands (mostly `show` commands) to monitor the operation of RIFT.

As a reminder: it is not possible to dynamically change the configuration of RIFT-Python using the CLI. If you want to change the configuration you must stop RIFT-Python, edit the configuration file, and restart RIFT-Python. That said, there are a small number of `set` commands in the CLI to change the behavior, for example to simulate a failure or repair of an interface. This allows you to study how RIFT behaves and reconverges after failures in a very simple manner (e.g. in automated unit tests).

Use the `--interactive` command line option or use Telnet to attach to the RIFT CLI:

```
(env) $ python rift --interactive topology/two_by_two_by_two.yaml
agg_101>
```

The help command lists all available operational commands:

```
agg_101> help
clear engine statistics
clear interface <interface> statistics
clear node statistics
exit
help
set interface <interface> failure <failure>
set level <level>
set node <node>
```

```
show bandwidth-balancing
show disaggregation
show engine
show engine statistics
show engine statistics exclude-zero
show flooding-reduction
show forwarding
show forwarding family <family>
show forwarding prefix <prefix>
show fsm lie
show fsm ztp
show interface <interface>
show interface <interface> fsm history
show interface <interface> fsm verbose-history
show interface <interface> packets
show interface <interface> queues
show interface <interface> security
show interface <interface> sockets
show interface <interface> statistics
show interface <interface> statistics exclude-zero
show interface <interface> tides
show interfaces
show kernel addresses
show kernel links
show kernel routes
show kernel routes table <table>
show kernel routes table <table> prefix <prefix>
show neighbors
show node
show node fsm history
show node fsm verbose-history
show node statistics
show node statistics exclude-zero
show nodes
show nodes level
show routes
show routes family <family>
show routes prefix <prefix>
show routes prefix <prefix> owner <owner>
show security
show spf
show spf direction <direction>
show spf direction <direction> destination <destination>
show tie-db
show tie-db direction <direction>
show tie-db direction <direction> originator <originator>
show tie-db direction <direction> originator <originator> tie-type <tie-type>
stop
```

NOTE The command line documentation is on GitHub at https://github.com/brunorijsman/rift-python/blob/master/doc/command-line-interface.md.

6.5.1 Global Commands

The following commands are global commands; they apply to the entire RIFT-Python process (also known as the RIFT engine).

6.5.1.1 exit

The `exit` command exits from the command line interface.

If RIFT-Python was started with the `--interactive` option and is running as a foreground process, then `exit` stops the RIFT-Python process.

If RIFT-Python was started without the `--interactive` option and is running as a background process, and you connected to the CLI using Telnet, then `exit` does not stop the RIFT-Python process – it continues running in the background.

```
agg_101> exit
(env) $
```

6.5.1.2 stop

The `stop` command exits from the CLI and stops the RIFT-Python process, regardless of whether it was started interactively in the foreground or non-interactively in the background.

```
agg_101> stop
(env) $
```

6.5.1.3 show engine

The `show engine` command shows global information about the RIFT-Python process, which is also known as the RIFT engine. Note that a single RIFT engine can run multiple nodes (i.e. multiple routers).

```
agg_101> show engine
+------------------------------------+---------------------+
| Stand-alone                        | False               |
| Interactive                        | True                |
| Simulated Interfaces               | True                |
| Physical Interface                 | eth0                |
| Telnet Port File                   | None                |
| IPv4 Multicast Loopback            | True                |
| IPv6 Multicast Loopback            | True                |
| Number of Nodes                    | 10                  |
| Transmit Source Address            | 127.0.0.1           |
| Flooding Reduction Enabled         | True                |
| Flooding Reduction Redundancy      | 2                   |
| Flooding Reduction Similarity      | 2                   |
| Flooding Reduction System Random   | 6159305269860546893 |
| Timer slips > 10ms                 | 0                   |
| Timer slips > 100ms                | 0                   |
| Timer slips > 1000ms               | 0                   |
| Max pending events processing time | 0.084237            |
| Max expired timers processing time | 0.010109            |
| Max select processing time         | 0.883265            |
| Max ready-to-read processing time  | 0.004386            |
+------------------------------------+---------------------+
```

6.5.2 Nodes

The following commands allow you to navigate between RIFT nodes (also known as *RIFT routers* or *RIFT switches*) in the topology and to view the state of nodes.

6.5.2.1 show nodes

The `show nodes` command shows a list of nodes in the current topology:

```
agg_101> show nodes
+-----------+--------+---------+
| Node      | System | Running |
| Name      | ID     |         |
+-----------+--------+---------+
| agg_101   | 101    | True    |
+-----------+--------+---------+
| agg_102   | 102    | True    |
+-----------+--------+---------+
...
+-----------+--------+---------+
| edge_2001 | 2001   | True    |
+-----------+--------+---------+
| edge_2002 | 2002   | True    |
+-----------+--------+---------+
```

6.5.2.2 set node <node>

The command line prompt shows the *current node*. Most commands apply to the current node. For example `show interfaces` shows the interfaces of the current node. You can change the current node using the `set node <node>` command:

```
agg_101> set node edge_2002
edge_2002>
```

6.5.2.3 show node

The `show node` command shows details for the current node:

```
agg_101> show node
Node:
+-------------------------------------------+----------------------------------------------+
| Name                                      | agg_101                                      |
| Passive                                   | False                                        |
| Running                                   | True                                         |
| System ID                                 | 101                                          |
| Configured Level                          | undefined                                    |
| Leaf Only                                 | False                                        |
| Leaf 2 Leaf                               | False                                        |
| Top of Fabric Flag                        | False                                        |
| Zero Touch Provisioning (ZTP) Enabled     | True                                         |
| ZTP FSM State                             | UPDATING_CLIENTS                             |
| ZTP Hold Down Timer                       | Stopped                                      |
| ZTP Hold Down Timer                       | Stopped                                      |
| Highest Available Level (HAL)             | 24                                           |
| Highest Adjacency Three-way (HAT)         | 24                                           |
| Level Value                               | 23                                           |
| Receive LIE IPv4 Multicast Address        | 224.0.0.81                                   |
| Transmit LIE IPv4 Multicast Address       | 224.0.0.120                                  |
| Receive LIE IPv6 Multicast Address        | FF02::A1F7                                   |
| Transmit LIE IPv6 Multicast Address       | FF02::A1F7                                   |
| Receive LIE Port                          | 20102                                        |
| Transmit LIE Port                         | 10000                                        |
| LIE Send Interval                         | 1.0 secs                                     |
| Receive TIE Port                          | 10001                                        |
| Kernel Route Table                        | 3                                            |
| Originate IPv4 Default Route              | True                                         |
| Reason for Originating IPv4 Default Route | This node has north-bound IPv4 default route |
| Originate IPv6 Default Route              | True                                         |
| Reason for Originating IPv6 Default Route | This node has north-bound IPv6 default route |
| Flooding Reduction Enabled                | True                                         |
| Flooding Reduction Redundancy             | 2                                            |
| Flooding Reduction Similarity             | 2                                            |
| Flooding Reduction Node Random            | 3208                                         |
+-------------------------------------------+----------------------------------------------+

Received Offers:
+-------------+-----------+-------+-----------------+-----------+-------+------------+---------+----------------+
| Interface   | System ID | Level | Not A ZTP Offer | State     | Best  | Best 3-Way | Removed | Removed Reason |
+-------------+-----------+-------+-----------------+-----------+-------+------------+---------+----------------+
| if_101_1    | 1         | 24    | False           | THREE_WAY | True  | True       | False   |                |
+-------------+-----------+-------+-----------------+-----------+-------+------------+---------+----------------+
| if_101_1001 | 1001      | 0     | False           | THREE_WAY | False | False      | True    | Level is leaf  |
+-------------+-----------+-------+-----------------+-----------+-------+------------+---------+----------------+
| if_101_1002 | 1002      | 0     | False           | THREE_WAY | False | False      | True    | Level is leaf  |
+-------------+-----------+-------+-----------------+-----------+-------+------------+---------+----------------+
| if_101_2    | 2         | 24    | False           | THREE_WAY | False | False      | False   |                |
+-------------+-----------+-------+-----------------+-----------+-------+------------+---------+----------------+

Sent Offers:
+-------------+-----------+-------+-----------------+-----------+
| Interface   | System ID | Level | Not A ZTP Offer | State     |
+-------------+-----------+-------+-----------------+-----------+
| if_101_1    | 101       | 23    | True            | THREE_WAY |
+-------------+-----------+-------+-----------------+-----------+
| if_101_1001 | 101       | 23    | False           | THREE_WAY |
+-------------+-----------+-------+-----------------+-----------+
| if_101_1002 | 101       | 23    | False           | THREE_WAY |
+-------------+-----------+-------+-----------------+-----------+
| if_101_2    | 101       | 23    | True            | THREE_WAY |
+-------------+-----------+-------+-----------------+-----------+
```

6.5.3 Interfaces, Neighbors, and Adjacencies

6.5.3.1 show interfaces

The `show interfaces` command shows a list of all interfaces on the current node:

```
agg_101> show interfaces
+-------------+-----------------------+-----------+-----------+-------------------+-------+
| Interface   | Neighbor              | Neighbor  | Neighbor  | Time in           | Flaps |
| Name        | Name                  | System ID | State     | State             |       |
+-------------+-----------------------+-----------+-----------+-------------------+-------+
| if_101_1    | core_1:if_1_101       | 1         | THREE_WAY | 0d 00h:07m:02.45s | 0     |
+-------------+-----------------------+-----------+-----------+-------------------+-------+
| if_101_1001 | edge_1001:if_1001_101 | 1001      | THREE_WAY | 0d 00h:07m:02.44s | 0     |
+-------------+-----------------------+-----------+-----------+-------------------+-------+
| if_101_1002 | edge_1002:if_1002_101 | 1002      | THREE_WAY | 0d 00h:07m:02.43s | 0     |
+-------------+-----------------------+-----------+-----------+-------------------+-------+
| if_101_2    | core_2:if_2_101       | 2         | THREE_WAY | 0d 00h:07m:02.44s | 0     |
+-------------+-----------------------+-----------+-----------+-------------------+-------+
```

6.5.3.2 show interface <interface>

The `show interface <interface>` command shows detailed information about one interface.

Unlike other link-state protocols (such as OSPF and IS-IS) RIFT only supports point-to-point interfaces and not multipoint interfaces. There can only be one neighbor and one adjacency per interface. Hence there is no need for separate `show neighbor` or `show adjacency` commands; all neighbor and adjacency information is reported in the output of `show interface`.

```
agg_101> show interface if_101_1001
Interface:
+-------------------------------------+--------------------------------------------+
| Interface Name                      | if_101_1001                                |
| Physical Interface Name             | eth0                                       |
| Advertised Name                     | agg_101:if_101_1001                        |
| Interface IPv4 Address              | 172.31.35.197                              |
| Interface IPv6 Address              | fe80::4ec:95ff:fea1:13ed%eth0              |
| Interface Index                     | 2                                          |
| Direction                           | South                                      |
| Metric                              | 1                                          |
| Bandwidth                           | 100 Mbps                                   |
| LIE Receive IPv4 Multicast Address  | 224.0.0.81                                 |
| LIE Receive IPv6 Multicast Address  | FF02::A1F7                                 |
| LIE Receive Port                    | 20033                                      |
| LIE Transmit IPv4 Multicast Address | 224.0.0.91                                 |
| LIE Transmit IPv6 Multicast Address | FF02::A1F7                                 |
| LIE Transmit Port                   | 20034                                      |
| Flooding Receive Port               | 20035                                      |
| System ID                           | 101                                        |
| Local ID                            | 3                                          |
| MTU                                 | 1400                                       |
| POD                                 | 0                                          |
| Failure                             | ok                                         |
| State                               | THREE_WAY                                  |
| Time in State                        | 0d 00h:08m:25.79s                          |
| Flaps                                | 0                                          |
| Received LIE Accepted or Rejected    | Accepted                                   |
| Received LIE Accept or Reject Reason | This node is not leaf and neighbor is leaf |
| Neighbor is Flood Repeater           | Not Applicable                             |
| Neighbor is Partially Connected      | False                                      |
| Nodes Causing Partial Connectivity   |                                            |
+--------------------------------------+--------------------------------------------+

Neighbor LIE Information:
+------------------------+--------------------------+
| Name                   | edge_1001:if_1001_101    |
| System ID              | 1001                     |
| IPv4 Address           | 172.31.35.197            |
| IPv6 Address           | fe80::4ec:95ff:fea1:13ed |
| LIE UDP Source Port    | 55334                    |
| Link ID                | 1                        |
| Level                  | 0                        |
| Flood UDP Port         | 20036                    |
| MTU                    | 1400                     |
| POD                    | 0                        |
| Hold Time              | 3                        |
| Not a ZTP Offer        | False                    |
| You are Flood Repeater | True                     |
| Your System ID         | 101                      |
| Your Local ID          | 3                        |
+------------------------+--------------------------+
```

6.5.3.3 show interface <interface> packets

The `show interface <interface> packets` command shows a full decode of the 20 most recently sent and received RIFT packets on the interface. This is useful for debugging. Note that Wireshark also supports decoding RIFT as described in section 5.3.

```
agg_101> show interface if_101_1001 packets
Last 20 Packets Sent and Received on Interface:
+------------------------------------------------------------------------------------------------------------------------------------+
| direction=RX  timestamp=2020-06-16-08:16:24.325398                                                                                 |
| local-address=ff02::a1f7%en0:20033  remote_address=fe80::1c9d:1463:d359:58e0%en0:52520                                             |
|                                                                                                                                    |
| packet-nr=11 outer-key-id=0 nonce-local=48612 nonce-remote=20330 remaining-lie-lifetime=all-ones outer-fingerprint-len=0           |
| protocol-packet=ProtocolPacket(header=PacketHeader(major_version=2, sender=1001, minor_version=0, level=0),                        |
| content=PacketContent(tide=None, tire=None, tie=None, lie=LIEPacket(flood_port=20036, link_mtu_size=1400, pod=0, holdtime=3,       |
| node_capabilities=NodeCapabilities(protocol_minor_version=0, flood_reduction=True, hierarchy_indications=1), label=None,           |
| you_are_sending_too_quickly=False, neighbor=Neighbor(remote_id=3, originator=101), name='edge_1001:if_1001_101',                   |
| not_a_ztp_offer=False, you_are_flood_repeater=True, local_id=1, link_bandwidth=100, instance_name=None, link_capabilities=None)))  |
+------------------------------------------------------------------------------------------------------------------------------------+
.                                                                                                   .
.                                                                                                   .
```

6.5.3.4 Show interface <interface> sockets

The `show interface <interface> sockets` command shows detailed information about the UDP multicast addresses and port numbers that are used for sending and receiving RIFT packets. This is useful information when issuing Linux commands such as `netstat` or `tcpdump` or when running Wireshark.

```
agg_101> show interface if_101_1001 sockets
+----------+-----------+--------+-----------------------------+------------+----------------+-------------+
| Traffic  | Direction | Family | Local Address               | Local Port | Remote Address | Remote Port |
+----------+-----------+--------+-----------------------------+------------+----------------+-------------+
| LIEs     | Receive   | IPv4   | 224.0.0.81                  | 20033      | Any            | Any         |
+----------+-----------+--------+-----------------------------+------------+----------------+-------------+
| LIEs     | Receive   | IPv6   | ff02::78%en0                | 20033      | Any            | Any         |
+----------+-----------+--------+-----------------------------+------------+----------------+-------------+
| LIEs     | Send      | IPv4   | 192.168.30.27               | 60159      | 224.0.0.91     | 20034       |
+----------+-----------+--------+-----------------------------+------------+----------------+-------------+
| LIEs     | Send      | IPv6   | fe80::1892:17:16db:1fd6%en0 | 60160      | ff02::78%en0   | 20034       |
+----------+-----------+--------+-----------------------------+------------+----------------+-------------+
| Flooding | Receive   | IPv4   | 192.168.30.27               | 20035      | Any            | Any         |
+----------+-----------+--------+-----------------------------+------------+----------------+-------------+
| Flooding | Receive   | IPv6   | fe80::1892:17:16db:1fd6%en0 | 20035      | Any            | Any         |
+----------+-----------+--------+-----------------------------+------------+----------------+-------------+
| Flooding | Send      | IPv4   | 192.168.30.27               | 64235      | 192.168.30.27  | 20036       |
+----------+-----------+--------+-----------------------------+------------+----------------+-------------+
```

6.5.3.5 show neighbors

The `show neighbors` command shows a list of all neighbors of the current node. There may be multiple parallel interfaces to a given neighbor, as is the case in the following example:

```
leaf> show neighbors
+-----------+-----------+-----------+------------+
| System ID | Direction | Interface | Adjacency  |
|           |           | Name      | Name       |
+-----------+-----------+-----------+------------+
| 2         | North     | if1       | spine1:if4 |
|           |           | if2       | spine1:if5 |
|           |           | if3       | spine1:if6 |
+-----------+-----------+-----------+------------+
| 3         | North     | if4       | spine2:if4 |
|           |           | if5       | spine2:if5 |
|           |           | if6       | spine2:if6 |
+-----------+-----------+-----------+------------+
```

6.5.4 Zero Touch Provisioning (ZTP)

The following commands allow you to monitor the zero touch provisioning (ZTP) process.

6.5.4.1 show nodes level

The `show nodes level` command shows the configured level (if any) and the ZTP-chosen level for all nodes in the topology:

```
agg_101> show nodes level
+-----------+--------+---------+---------------+-------+
| Node      | System | Running | Configured    | Level |
| Name      | ID     |         | Level         | Value |
+-----------+--------+---------+---------------+-------+
| agg_101   | 101    | True    | undefined     | 23    |
+-----------+--------+---------+---------------+-------+
| agg_102   | 102    | True    | undefined     | 23    |
+-----------+--------+---------+---------------+-------+
| agg_201   | 201    | True    | undefined     | 23    |
+-----------+--------+---------+---------------+-------+
| agg_202   | 202    | True    | undefined     | 23    |
+-----------+--------+---------+---------------+-------+
| core_1    | 1      | True    | top-of-fabric | 24    |
+-----------+--------+---------+---------------+-------+
| core_2    | 2      | True    | top-of-fabric | 24    |
+-----------+--------+---------+---------------+-------+
| edge_1001 | 1001   | True    | 0             | 0     |
+-----------+--------+---------+---------------+-------+
| edge_1002 | 1002   | True    | 0             | 0     |
+-----------+--------+---------+---------------+-------+
| edge_2001 | 2001   | True    | 0             | 0     |
+-----------+--------+---------+---------------+-------+
| edge_2002 | 2002   | True    | 0             | 0     |
+-----------+--------+---------+---------------+-------+
```

6.5.4.2 set level <level>

The `set level <level>` command changes the configured level of the current node. The `level` parameter must be one of the following values: `leaf`, `spine`, `top-of-fabric`, `undefined`, or an integer between 0 and 24 inclusive.

The `set level` command is a rare exception to the rule that the configuration cannot only be changed by editing the configuration file and restarting the RIFT-Python process; it is needed to allow the automated unit tests to test correct reconvergence of the ZTP process after changing the configured level of a node.

```
agg_101> set level 15
```

6.5.5 TIE Database (Link-State Database)

6.5.5.1 show tie-db

The `show tie-db` command shows all topology information elements (TIEs) in the TIE database (also known as the *Link-State Database*).

The output of `show tie-db` can be very large. You can use one of the following variations of the `show tie-db` command to show a subset of the TIEs in the database, for a specific direction, originator, or TIE type.

```
show tie-db direction <direction>
show tie-db direction <direction> originator <originator>
show tie-db direction <direction> originator <originator> tie-type <tie-type>
```

The `direction` parameter must be `south` or `north`.

The `originator` parameter must be an integer which is the system ID of some node in the topology.

The `tie-type` parameter must be one of the following values: `node`, `prefix`, `pos-dis-prefix`, `neg-dis-prefix`, `ext-prefix`, `pg-prefix`, or `key-value`.

```
agg_101> show tie-db
+-----------+------------+--------+--------+--------+----------+-------------------------+
| Direction | Originator | Type   | TIE Nr | Seq Nr | Lifetime | Contents                |
+-----------+------------+--------+--------+--------+----------+-------------------------+
| South     | 1          | Node   | 1      | 5      | 604051   | Name: core_1            |
|           |            |        |        |        |          | Level: 24               |
|           |            |        |        |        |          | Capabilities:           |
|           |            |        |        |        |          |   Flood reduction: True |
|           |            |        |        |        |          | Neighbor: 101           |
|           |            |        |        |        |          |   Level: 23             |
|           |            |        |        |        |          |   Cost: 1               |
|           |            |        |        |        |          |   Bandwidth: 100 Mbps   |
|           |            |        |        |        |          |   Link: 1-1             |
|           |            |        |        |        |          | Neighbor: 102           |
|           |            |        |        |        |          |   Level: 23             |
|           |            |        |        |        |          |   Cost: 1               |
|           |            |        |        |        |          |   Bandwidth: 100 Mbps   |
|           |            |        |        |        |          |   Link: 2-1             |
|           |            |        |        |        |          | Neighbor: 201           |
|           |            |        |        |        |          |   Level: 23             |
|           |            |        |        |        |          |   Cost: 1               |
|           |            |        |        |        |          |   Bandwidth: 100 Mbps   |
|           |            |        |        |        |          |   Link: 3-1             |
|           |            |        |        |        |          | Neighbor: 202           |
|           |            |        |        |        |          |   Level: 23             |
|           |            |        |        |        |          |   Cost: 1               |
|           |            |        |        |        |          |   Bandwidth: 100 Mbps   |
|           |            |        |        |        |          |   Link: 4-1             |
+----------+-----------+--------+--------+--------+---------+---------------------- +
...
+----------+-----------+--------+--------+--------+---------+-----------------------+
| North     | 1002       | Prefix | 2      | 1      | 604049   | Prefix: 1.2.1.0/24      |
|           |            |        |        |        |          |   Metric: 1             |
|           |            |        |        |        |          | Prefix: 1.2.2.0/24      |
|           |            |        |        |        |          |   Metric: 1             |
|           |            |        |        |        |          | Prefix: 1.2.3.0/24      |
|           |            |        |        |        |          |   Metric: 1             |
|           |            |        |        |        |          | Prefix: 1.2.4.0/24      |
|           |            |        |        |        |          |   Metric: 1             |
|           |            |        |        |        |          | Prefix: 99.99.99.0/24   |
|           |            |        |        |        |          |   Metric: 1             |
|           |            |        |        |        |          |   Tag: 9992             |
+-----------+------------+--------+--------+--------+----------+-------------------------+
```

6.5.6 Flooding

6.5.6.1 show interface <interface> queues

The `show interface <interface> queues` command shows the TIE transmit queue, the TIE retransmit queue, the TIE request queue, and the TIE acknowledge queue. When RIFT has converged, these queues should be empty. Thus, non-empty queues are an indication of topology changes or convergence issues.

```
agg_101> show interface if_101_1001 queues
Transmit queue:
+-----------+------------+------+--------+--------+-------+
| Direction | Originator | Type | TIE Nr | Seq Nr | Send  |
|           |            |      |        |        | Delay |
+-----------+------------+------+--------+--------+-------+

Request queue:
+-----------+------------+------+--------+--------+-------+
| Direction | Originator | Type | TIE Nr | Seq Nr | Send  |
|           |            |      |        |        | Delay |
+-----------+------------+------+--------+--------+-------+

Acknowledge queue:
+-----------+------------+------+--------+--------+-----------+-------+
| Direction | Originator | Type | TIE Nr | Seq Nr | Remaining | Send  |
|           |            |      |        |        | Lifetime  | Delay |
+-----------+------------+------+--------+--------+-----------+-------+
```

6.5.6.2 show interface <interface> tides

The `show interface <interface> tides` command shows the topology information description elements (TIDEs). Due to the RIFT flooding rules, it is possible that a RIFT node sends different TIDEs to different neighbors. For that reason, this is a per-interface command.

```
agg_101> show interface if_101_1001 tides
Send TIDEs:
+----------------+--------- ... ----+-----------+------------+--------+--------+--------+-----------+-------------+
| Start          | End              | Direction | Originator | Type   | TIE Nr | Seq Nr | Remaining | Origination |
| Range          | Range            |           |            |        |        |        | Lifetime  | Time        |
+----------------+--------- ... ----+-----------+------------+--------+--------+--------+-----------+-------------+
| South:0:Node:0 | North:18 ... 295 | South     | 1          | Node   | 1      | 5      | 604749    | -           |
|                |                  | South     | 2          | Node   | 1      | 5      | 604749    | -           |
|                |                  | South     | 101        | Node   | 1      | 5      | 604749    | -           |
|                |                  | South     | 101        | Prefix | 2      | 1      | 604749    | -           |
|                |                  | South     | 102        | Node   | 1      | 5      | 604749    | -           |
|                |                  | North     | 101        | Node   | 1      | 5      | 604749    | -           |
|                |                  | North     | 1001       | Node   | 1      | 3      | 604749    | -           |
|                |                  | North     | 1001       | Prefix | 2      | 1      | 604748    | -           |
|                |                  | North     | 1002       | Node   | 1      | 3      | 604749    | -           |
|                |                  | North     | 1002       | Prefix | 2      | 1      | 604748    | -           |
+----------------+------------------+-----------+------------+--------+--------+--------+-----------+-------------+
```

6.5.7 Flooding Reduction

6.5.7.1 show flooding-reduction

The `show flooding-reduction` command shows:

- Which parent nodes have been elected as flood repeaters for flooding reduction, and why.
- For which interfaces this node is a flood repeater, and why.

```
edge_1001> show flooding-reduction
Parents:
+-------------+-----------+---------------------+-------------+------------+----------+
| Interface   | Parent    | Parent              | Grandparent | Similarity | Flood    |
| Name        | System ID | Interface           | Count       | Group      | Repeater |
|             |           | Name                |             |            |          |
+-------------+-----------+---------------------+-------------+------------+----------+
| if_1001_101 | 101       | agg_101:if_101_1001 | 2           | 1: 2-2     | True     |
+-------------+-----------+---------------------+-------------+------------+----------+
| if_1001_102 | 102       | agg_102:if_102_1001 | 2           | 1: 2-2     | True     |
+-------------+-----------+---------------------+-------------+------------+----------+

Grandparents:
+-------------+--------+-------------+-------------+
| Grandparent | Parent | Flood       | Redundantly |
| System ID   | Count  | Repeater    | Covered     |
|             |        | Adjacencies |             |
+-------------+--------+-------------+-------------+
| 1           | 2      | 2           | True        |
+-------------+--------+-------------+-------------+
| 2           | 2      | 2           | True        |
+-------------+--------+-------------+-------------+

Interfaces:
+-------------+---------------------+-----------+-----------+-----------+----------------+----------------+
| Interface   | Neighbor            | Neighbor  | Neighbor  | Neighbor  | Neighbor is    | This Node is   |
| Name        | Interface           | System ID | State     | Direction | Flood Repeater | Flood Repeater |
|             | Name                |           |           |           | for This Node  | for Neighbor   |
+-------------+---------------------+-----------+-----------+-----------+----------------+----------------+
| if_1001_101 | agg_101:if_101_1001 | 101       | THREE_WAY | North     | True           | Not Applicable |
+-------------+---------------------+-----------+-----------+-----------+----------------+----------------+
| if_1001_102 | agg_102:if_102_1001 | 102       | THREE_WAY | North     | True           | Not Applicable |
+-------------+---------------------+-----------+-----------+-----------+----------------+----------------+
```

6.5.8 Shortest Path First (SPF)

6.5.8.1 show spf

The `show spf` command shows each shortest path tree computed by the short path first (SPF) algorithm. RIFT computes three separate shortest path trees:

- The normal south SPF tree, which is used to compute routes for southbound destinations.

- The normal north SPF tree, which is used to compute routes for northbound destinations.
- A special south SPF tree, which is used to determine which prefixes need to be negatively disaggregated. Unlike the normal south SPF tree, the special south SPF tree includes inter-plane east-west links.

The output of `show spf` can be very large. You can use one of the following variations of the `show spf` command to show only one SPF tree or even a single SPF destination:

```
show spf direction <direction>
show spf direction <direction> destination <destination>
```

The `direction` parameter must be `south` or `north` or `south-with-ew`.

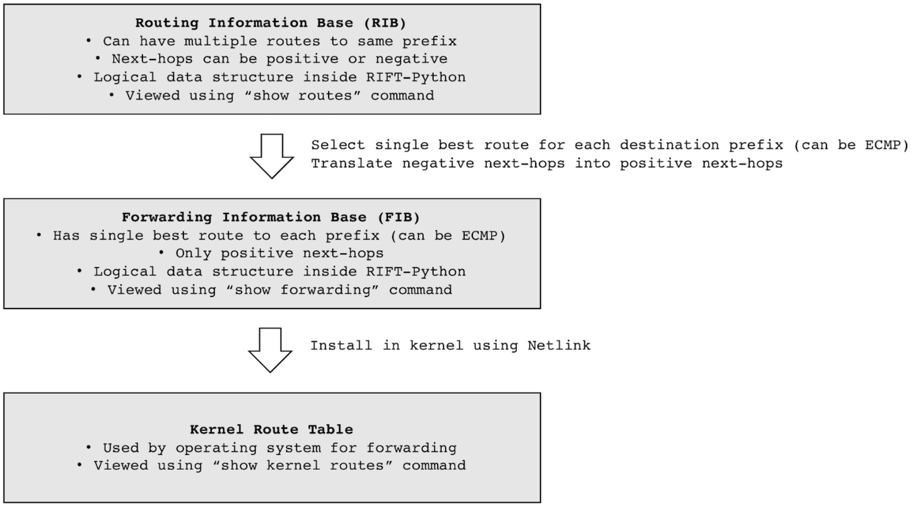
The `destination` parameter must be the system identifier of a node (i.e. an integer), or an IPv4 prefix, or an IPv6 prefix.

```
agg_101> show spf
SPF Statistics:
+---------------+----+
| SPF Runs      | 3  |
+---------------+----+
| SPF Deferrals | 22 |
+---------------+----+

South SPF Destinations:
+------------------+------+---------+-------------+------+--------------+---------------------------+----------------+
| Destination      | Cost | Is Leaf | Predecessor | Tags | Disaggregate | IPv4 Next-hops            | IPv6 Next-hops |
|                  |      |         | System IDs  |      |              |                           |                |
+------------------+------+---------+-------------+------+--------------+---------------------------+----------------+
| 101 (agg_101)    | 0    | False   |             |      |              |                           |                |
+------------------+------+---------+-------------+------+--------------+---------------------------+----------------+
| 1001 (edge_1001) | 1    | True    | 101         |      |              | if_101_1001 172.31.35.197 | if_101_1001... |
+------------------+------+---------+-------------+------+--------------+---------------------------+----------------+
| 1002 (edge_1002) | 1    | True    | 101         |      |              | if_101_1002 172.31.35.197 | if_101_1002... |
+------------------+------+---------+-------------+------+--------------+---------------------------+----------------+
| 1.1.1.0/24       | 2    | True    | 1001        |      |              | if_101_1001 172.31.35.197 | if_101_1001... |
+------------------+------+---------+-------------+------+--------------+---------------------------+----------------+
| 1.1.2.0/24       | 2    | True    | 1001        |      |              | if_101_1001 172.31.35.197 | if_101_1001... |
+------------------+------+---------+-------------+------+--------------+---------------------------+----------------+
| 1.1.3.0/24       | 2    | True    | 1001        |      |              | if_101_1001 172.31.35.197 | if_101_1001... |
+------------------+------+---------+-------------+------+--------------+---------------------------+----------------+
| 1.1.4.0/24       | 2    | True    | 1001        |      |              | if_101_1001 172.31.35.197 | if_101_1001... |
+------------------+------+---------+-------------+------+--------------+---------------------------+----------------+
| 1.2.1.0/24       | 2    | True    | 1002        |      |              | if_101_1002 172.31.35.197 | if_101_1002... |
+------------------+------+---------+-------------+------+--------------+---------------------------+----------------+
| 1.2.2.0/24       | 2    | True    | 1002        |      |              | if_101_1002 172.31.35.197 | if_101_1002... |
+------------------+------+---------+-------------+------+--------------+---------------------------+----------------+
| 1.2.3.0/24       | 2    | True    | 1002        |      |              | if_101_1002 172.31.35.197 | if_101_1002... |
+------------------+------+---------+-------------+------+--------------+---------------------------+----------------+
| 1.2.4.0/24       | 2    | True    | 1002        |      |              | if_101_1002 172.31.35.197 | if_101_1002... |
+------------------+------+---------+-------------+------+--------------+---------------------------+----------------+
| 99.99.99.0/24    | 2    | True    | 1001        | 9992 |              | if_101_1001 172.31.35.197 | if_101_1001... |
|                  |      |         | 1002        | 9991 |              | if_101_1002 172.31.35.197 | if_101_1002... |
+------------------+------+---------+-------------+------+--------------+---------------------------+----------------+
North SPF Destinations:
+---------------+------+---------+-------------+------+--------------+-----------------------------+----------------+
| Destination   | Cost | Is Leaf | Predecessor | Tags | Disaggregate | IPv4 Next-hops              | IPv6 Next-hops |
|               |      |         | System IDs  |      |              |                             |                |
+---------------+------+---------+-------------+------+--------------+-----------------------------+----------------+
| 1 (core_1)    | 1    | False   | 101         |      |              | if_101_1 172.31.35.197 (50) | if_101_1 fe... |
+---------------+------+---------+-------------+------+--------------+-----------------------------+----------------+
| 2 (core_2)    | 1    | False   | 101         |      |              | if_101_2 172.31.35.197 (50) | if_101_2 fe... |
+---------------+------+---------+-------------+------+--------------+-----------------------------+----------------+
| 101 (agg_101) | 0    | False   |             |      |              |                             |                |
+---------------+------+---------+-------------+------+--------------+-----------------------------+----------------+
| 0.0.0.0/0     | 2    | False   | 1           |      |              | if_101_1 172.31.35.197 (50) | if_101_1 fe... |
|               |      |         | 2           |      |              | if_101_2 172.31.35.197 (50) | if_101_2 fe... |
+---------------+------+---------+-------------+------+--------------+-----------------------------+----------------+
| ::/0          | 2    | False   | 1           |      |              | if_101_1 172.31.35.197 (50) | if_101_1 fe... |
|               |      |         | 2           |      |              | if_101_2 172.31.35.197 (50) | if_101_2 fe... |
+---------------+------+---------+-------------+------+--------------+-----------------------------+----------------+

South SPF (with East-West Links) Destinations:
+-------------+------+---------+-------------+------+--------------+----------------+----------------+
| Destination | Cost | Is Leaf | Predecessor | Tags | Disaggregate | IPv4 Next-hops | IPv6 Next-hops |
|             |      |         | System IDs  |      |              |                |                |
+-------------+------+---------+-------------+------+--------------+----------------+----------------+
```

6.5.9 The Routing Information Base (RIB)

6.5.9.1 show routes

The `show routes` command shows the routes in the routing information base (RIB). Figure 6.8 shows how the RIB relates to the forwarding information base (FIB) and the kernel route tables.



Routing Information Base (RIB)
- Can have multiple routes to same prefix
- Next-hops can be positive or negative
- Logical data structure inside RIFT-Python
- Viewed using “show routes” command

Select single best route for each destination prefix (can be ECMP)
Translate negative next-hops into positive next-hops

Forwarding Information Base (FIB)
- Has single best route to each prefix (can be ECMP)
- Only positive next-hops
- Logical data structure inside RIFT-Python
- Viewed using “show forwarding” command

Install in kernel using Netlink

Kernel Route Table
- Used by operating system for forwarding
- Viewed using “show kernel routes” command

Figure 6.8 The Relation Between the RIB, the FIB, and the Kernel Route Tables

The output of `show routes` can be very large. You can use one of the following variations of the `show routes` command to show only the routes for a specific address family or for a specific prefix or for a specific owner.

```
show routes family <family>
show routes prefix <prefix>
show routes prefix <prefix> owner <owner>
```

The `family` parameter must be `ipv4` or `ipv6`. The `prefix` parameter must be an IPv4 prefix or an IPv6 prefix. The `owner` parameter must be `south-spf` or `north-spf`.

```
agg_101> show routes
IPv4 Routes:
+---------------+-----------+----------+-------------+---------------+----------+
| Prefix        | Owner     | Next-hop | Next-hop    | Next-hop      | Next-hop |
|               |           | Type     | Interface   | Address       | Weight   |
+---------------+-----------+----------+-------------+---------------+----------+
| 0.0.0.0/0     | North SPF | Positive | if_101_1    | 172.31.35.197 | 50       |
|               |           | Positive | if_101_2    | 172.31.35.197 | 50       |
+---------------+-----------+----------+-------------+---------------+----------+
| 1.1.1.0/24    | South SPF | Positive | if_101_1001 | 172.31.35.197 |          |
+---------------+-----------+----------+-------------+---------------+----------+
| 1.1.2.0/24    | South SPF | Positive | if_101_1001 | 172.31.35.197 |          |
+---------------+-----------+----------+-------------+---------------+----------+
| 1.1.3.0/24    | South SPF | Positive | if_101_1001 | 172.31.35.197 |          |
+---------------+-----------+----------+-------------+---------------+----------+
| 1.1.4.0/24    | South SPF | Positive | if_101_1001 | 172.31.35.197 |          |
+---------------+-----------+----------+-------------+---------------+----------+
| 1.2.1.0/24    | South SPF | Positive | if_101_1002 | 172.31.35.197 |          |
+---------------+-----------+----------+-------------+---------------+----------+
| 1.2.2.0/24    | South SPF | Positive | if_101_1002 | 172.31.35.197 |          |
+---------------+-----------+----------+-------------+---------------+----------+
| 1.2.3.0/24    | South SPF | Positive | if_101_1002 | 172.31.35.197 |          |
+---------------+-----------+----------+-------------+---------------+----------+
| 1.2.4.0/24    | South SPF | Positive | if_101_1002 | 172.31.35.197 |          |
+---------------+-----------+----------+-------------+---------------+----------+
| 99.99.99.0/24 | South SPF | Positive | if_101_1001 | 172.31.35.197 |          |
|               |           | Positive | if_101_1002 | 172.31.35.197 |          |
+---------------+-----------+----------+-------------+---------------+----------+

IPv6 Routes:
+--------+-----------+----------+-----------+--------------------------+----------+
| Prefix | Owner     | Next-hop | Next-hop  | Next-hop                 | Next-hop |
|        |           | Type     | Interface | Address                  | Weight   |
+--------+-----------+----------+-----------+--------------------------+----------+
| ::/0   | North SPF | Positive | if_101_1  | fe80::4ec:95ff:fea1:13ed | 50       |
|        |           | Positive | if_101_2  | fe80::4ec:95ff:fea1:13ed | 50       |
+--------+-----------+----------+-----------+--------------------------+----------+
```

6.5.10 The Forwarding Information Base (FIB)

6.5.10.1 show forwarding

The `show forwarding` command shows the routes in the FIB.

The FIB differs from the RIB in two ways:

1. Whereas the RIB may contain multiple routes to the same prefix (for example, a south SPF route and a north SPF route), the FIB always contains only a single route to any given prefix, namely the route in the RIB which was selected as the best route in the RIB. Note that a route with multiple ECMP or weighted next-hops is still considered to be a single route. In some other routing stack implementations there are scenarios where multiple equally good best RIB routes are installed in the FIB by combining the next-hops of multiple best RIB routes into a single FIB route. RIFT-Python does not do that.

2. Whereas the RIB may contain both positive and negative next-hops, the FIB only contains positive next-hops. Any negative next-hops in the RIB are translated into complementary positive next-hops in the FIB.

See Figure 6.7 for a diagram showing the relationship between the RIB, the FIB, and the kernel routes.

The `show forwarding` output can be very large. You can use one of the following variations of the `show spf` command to show only the routes for a specific address family or for a specific prefix or for a specific owner.

```
show forwarding family <family>
show forwarding prefix <prefix>
```

The `family` parameter must be `ipv4` or `ipv6`. The `prefix` parameter must be an IPv4 prefix or an IPv6 prefix.

```
agg_101> show forwarding
IPv4 Routes:
+---------------+----------+-------------+---------------+----------+
| Prefix        | Next-hop | Next-hop    | Next-hop      | Next-hop |
|               | Type     | Interface   | Address       | Weight   |
+---------------+----------+-------------+---------------+----------+
| 0.0.0.0/0     | Positive | if_101_1    | 172.31.35.197 | 50       |
|               | Positive | if_101_2    | 172.31.35.197 | 50       |
+---------------+----------+-------------+---------------+----------+
| 1.1.1.0/24    | Positive | if_101_1001 | 172.31.35.197 |          |
+---------------+----------+-------------+---------------+----------+
| 1.1.2.0/24    | Positive | if_101_1001 | 172.31.35.197 |          |
+---------------+----------+-------------+---------------+----------+
| 1.1.3.0/24    | Positive | if_101_1001 | 172.31.35.197 |          |
+---------------+----------+-------------+---------------+----------+
| 1.1.4.0/24    | Positive | if_101_1001 | 172.31.35.197 |          |
+---------------+----------+-------------+---------------+----------+
| 1.2.1.0/24    | Positive | if_101_1002 | 172.31.35.197 |          |
+---------------+----------+-------------+---------------+----------+
| 1.2.2.0/24    | Positive | if_101_1002 | 172.31.35.197 |          |
+---------------+----------+-------------+---------------+----------+
| 1.2.3.0/24    | Positive | if_101_1002 | 172.31.35.197 |          |
+---------------+----------+-------------+---------------+----------+
| 1.2.4.0/24    | Positive | if_101_1002 | 172.31.35.197 |          |
+---------------+----------+-------------+---------------+----------+
| 99.99.99.0/24 | Positive | if_101_1001 | 172.31.35.197 |          |
|               | Positive | if_101_1002 | 172.31.35.197 |          |
+---------------+----------+-------------+---------------+----------+

IPv6 Routes:
+--------+----------+-----------+--------------------------+----------+
| Prefix | Next-hop | Next-hop  | Next-hop                 | Next-hop |
|        | Type     | Interface | Address                  | Weight   |
+--------+----------+-----------+--------------------------+----------+
| ::/0   | Positive | if_101_1  | fe80::4ec:95ff:fea1:13ed | 50       |
|        | Positive | if_101_2  | fe80::4ec:95ff:fea1:13ed | 50       |
+--------+----------+-----------+--------------------------+----------+
```

6.5.11 The Linux Kernel

The following group of commands shows addresses, links, or routes in the Linux kernel. Interaction with the kernel is only supported on Linux. On macOS, `show kernel` commands are not supported and RIFT-Python does not install any routes into the kernel.

6.5.11.1 show kernel addresses

The `show kernel addresses` command shows the interface addresses in the kernel:

```
agg_101> show kernel addresses
Kernel Addresses:
+-----------+--------------------------+---------------+---------------+---------+
| Interface | Address                  | Local         | Broadcast     | Anycast |
| Name      |                          |               |               |         |
+-----------+--------------------------+---------------+---------------+---------+
| lo        | 127.0.0.1                | 127.0.0.1     |               |         |
+-----------+--------------------------+---------------+---------------+---------+
| eth0      | 172.31.35.197            | 172.31.35.197 | 172.31.47.255 |         |
+-----------+--------------------------+---------------+---------------+---------+
|           | ::1                      |               |               |         |
+-----------+--------------------------+---------------+---------------+---------+
|           | fe80::4ec:95ff:fea1:13ed |               |               |         |
+-----------+--------------------------+---------------+---------------+---------+
```

6.5.11.2 show kernel links

The `show kernel links` command shows the interface datalink layer (therefore Ethernet) information in the kernel.

```
agg_101> show kernel links
Kernel Links:
+-----------+-----------+-------------------+-------------------+-----------+-------+-----------+
| Interface | Interface | Hardware          | Hardware          | Link Type | MTU   | Flags     |
| Name      | Index     | Address           | Broadcast         |           |       |           |
|           |           |                   | Address           |           |       |           |
+-----------+-----------+-------------------+-------------------+-----------+-------+-----------+
| lo        | 1         | 00:00:00:00:00:00 | 00:00:00:00:00:00 |           | 65536 | UP        |
|           |           |                   |                   |           |       | LOOPBACK  |
|           |           |                   |                   |           |       | RUNNING   |
|           |           |                   |                   |           |       | LOWER_UP  |
+-----------+-----------+-------------------+-------------------+-----------+-------+-----------+
| eth0      | 2         | 06:ec:95:a1:13:ed | ff:ff:ff:ff:ff:ff |           | 9001  | UP        |
|           |           |                   |                   |           |       | BROADCAST |
|           |           |                   |                   |           |       | RUNNING   |
|           |           |                   |                   |           |       | MULTICAST |
|           |           |                   |                   |           |       | LOWER_UP  |
+-----------+-----------+-------------------+-------------------+-----------+-------+-----------+
```

6.5.11.3 show kernel routes

The `show kernel routes` command shows the routes in the kernel route tables.

The output of `show kernel routes` can be very large. You can use one of the following variations of the `show kernel routes` command to show only the routes in a specific kernel route table or even a specific prefix.

```
show kernel routes table <table>
show kernel routes table <table> prefix <prefix>
```

The `table` parameter must be `local`, `main`, `default`, `unspecified` or a number. The `prefix` parameter must be an IPv4 prefix or an IPv6 prefix.

```
agg_101> show kernel routes
Kernel Routes:
+-------+---------+---------------------+-----------+----------+-----------+-------------+--------+
| Table | Address | Destination         | Type      | Protocol | Outgoing  | Gateway     | Weight |
|       | Family  |                     |           |          | Interface |             |        |
+-------+---------+---------------------+-----------+----------+-----------+-------------+--------+
| Main  | IPv4    | 0.0.0.0/0           | Unicast   | Dhcp     | eth0      | 172.31.32.1 |        |
+-------+---------+---------------------+-----------+----------+-----------+-------------+--------+
| Main  | IPv4    | 172.31.32.0/20      | Unicast   | Kernel   | eth0      |             |        |
+-------+---------+---------------------+-----------+----------+-----------+-------------+--------+
| Main  | IPv4    | 172.31.32.1/32      | Unicast   | Dhcp     | eth0      |             |        |
+-------+---------+---------------------+-----------+----------+-----------+-------------+--------+
| Main  | IPv6    | ::1/128             | Unicast   | Kernel   | lo        |             |        |
+-------+---------+---------------------+-----------+----------+-----------+-------------+--------+
| Main  | IPv6    | fe80::/64           | Unicast   | Kernel   | eth0      |             |        |
+-------+---------+---------------------+-----------+----------+-----------+-------------+--------+
| Local | IPv4    | 127.0.0.0/8         | Local     | Kernel   | lo        |             |        |
+-------+---------+---------------------+-----------+----------+-----------+-------------+--------+
...
+-------+---------+---------------------+-----------+----------+-----------+-------------+--------+
| Local | IPv6    | ff00::/8            | Unicast   | Boot     | eth0      |             |        |
+-------+---------+---------------------+-----------+----------+-----------+-------------+--------+
```

6.5.12 Security

6.5.12.1 show security

The `show security` command shows the following security related information for the current node:

- The currently configured security keys.
- The active origin key (if any) and additional accepted origin keys (if any) for key roll-over.
- Various security related statistics for the current node.

The origin key is used for signing TIE payloads. To see information related to the outer key (which is used to sign all RIFT packets) see the `show interface interface security` command:

```
spine-1-1> show security
Security Keys:
+--------+-----------+--------+
| Key ID | Algorithm | Secret |
+--------+-----------+--------+
| 0      | null      |        |
+--------+-----------+--------+

Origin Keys:
+--------------------+-----------+
| Key                | Key ID(s) |
+--------------------+-----------+
| Active Origin Key  | None      |
+--------------------+-----------+
| Accept Origin Keys |           |
+--------------------+-----------+

Security Statistics:
+---------------------------------------+---------------------------+-------------------------------------+-------------------+
| Description                           | Value                     | Last Rate                           | Last Change       |
|                                       |                           | Over Last 10 Changes                |                   |
+---------------------------------------+---------------------------+-------------------------------------+-------------------+
| Missing outer security envelope       | 0 Packets, 0 Bytes        |                                     |                   |
+---------------------------------------+---------------------------+-------------------------------------+-------------------+
| Zero outer key id not accepted        | 0 Packets, 0 Bytes        |                                     |                   |
+---------------------------------------+---------------------------+-------------------------------------+-------------------+
...
+---------------------------------------+---------------------------+-------------------------------------+-------------------+
| Empty origin fingerprint accepted     | 25 Packets, 8037 Bytes    | 2.08 Packets/Sec, 739.34 Bytes/Sec  | 0d 00h:11m:24.96s |
+---------------------------------------+---------------------------+-------------------------------------+-------------------+
```

6.5.12.2 show interface <interface> security

The `show interface <interface> security` command shows the following security related information for the specified interface:

- The active outer key (if any) and additional accepted outer keys (if any) for key roll-over.
- Information about the state of nonces on the interface.
- Various security related statistics for the interface.

The outer key is used for signing all RIFT packets. To see information related to the origin key (which is used to sign TIE payloads) see the per-node `show security` command above.

```
spine-1-1> show interface veth-101a-1001a security
Outer Keys:
+-------------------+-----------+----------------------+
| Key               | Key ID(s) | Configuration Source |
+-------------------+-----------+----------------------+
| Active Outer Key  | None      | Node Active Key      |
+-------------------+-----------+----------------------+
| Accept Outer Keys |           | Node Accept Keys     |
+-------------------+-----------+----------------------+

Nonces:
+--------------------------+----------------+
| Last Received LIE Nonce  | 20305          |
+--------------------------+----------------+
| Last Sent Nonce          | 6700           |
+--------------------------+----------------+
| Next Sent Nonce Increase | 15.508507 secs |
+--------------------------+----------------+

Security Statistics:
+--------------------------------------+----------------------------+-------------------------------------+-------------------+
| Description                          | Value                      | Last Rate                           | Last Change       |
|                                      |                            | Over Last 10 Changes                |                   |
+--------------------------------------+----------------------------+-------------------------------------+-------------------+
| Missing outer security envelope      | 0 Packets, 0 Bytes         |                                     |                   |
+--------------------------------------+----------------------------+-------------------------------------+-------------------+
| Zero outer key id not accepted       | 0 Packets, 0 Bytes         |                                     |                   |
+--------------------------------------+----------------------------+-------------------------------------+-------------------+
...
+--------------------------------------+----------------------------+-------------------------------------+-------------------+
| Empty outer fingerprint accepted     | 3609 Packets, 829870 Bytes | 3.00 Packets/Sec, 708.32 Bytes/Sec  | 0d 00h:00m:00.61s |
+--------------------------------------+----------------------------+-------------------------------------+-------------------+
| Empty origin fingerprint accepted    | 6 Packets, 1912 Bytes      | 5.05 Packets/Sec, 1626.26 Bytes/Sec | 0d 00h:23m:56.59s |
+--------------------------------------+----------------------------+-------------------------------------+-------------------+
```

6.5.13 Disaggregation (positive and negative)

6.5.13.1. show disaggregation

The `show disaggregation` command shows the following disaggregation related information:

- Information about which nodes at the same level are missing adjacencies or have extra adjacencies (these are triggers for positive disaggregation).
- Information about which interfaces are particularly connected. An interface is partially connected if the neighbor is missing an adjacency with at least one node at the same level. This is also useful for debugging positive disaggregation.
- A list of all positive disaggregation prefix TIEs (both locally originated and received).
- A list of all negative disaggregation prefix TIEs (both locally originated and received).

NOTE Negative disaggregation is triggered by having different reachable prefixes in the normal south-bound shortest path first (SPF) calculation and the special south-bound SPF that uses east-west links. The `show spf` command is useful to investigate this.

```
super-1-2> show disaggregation
Same Level Nodes:
+---------------+-------------+-----------------+-----------------+-------------+
| Same-Level    | North-bound | South-bound     | Missing         | Extra       |
| Node          | Adjacencies | Adjacencies     | South-bound     | South-bound |
|               |             |                 | Adjacencies     | Adjacencies |
+---------------+-------------+-----------------+-----------------+-------------+
| super-1-1 (1) |             | spine-2-1 (104) | spine-1-1 (101) |             |
|               |             | spine-3-1 (107) |                 |             |
+---------------+-------------+-----------------+-----------------+-------------+

Partially Connected Interfaces:
+--------------+------------------------------------+
| Name         | Nodes Causing Partial Connectivity |
+--------------+------------------------------------+
| veth-2a-101e | super-1-1 (1)                      |
+--------------+------------------------------------+

Positive Disaggregation TIEs:
+-----------+------------+----------------+--------+--------+----------+-----------------------------+
| Direction | Originator | Type           | TIE Nr | Seq Nr | Lifetime | Contents                    |
+-----------+------------+----------------+--------+--------+----------+-----------------------------+
| South     | 2          | Pos-Dis-Prefix | 3      | 1      | 604764   | Pos-Dis-Prefix: 88.0.1.1/32 |
|           |            |                |        |        |          |   Metric: 3                 |
|           |            |                |        |        |          | Pos-Dis-Prefix: 88.0.2.1/32 |
|           |            |                |        |        |          |   Metric: 3                 |
|           |            |                |        |        |          | Pos-Dis-Prefix: 88.0.3.1/32 |
|           |            |                |        |        |          |   Metric: 3                 |
|           |            |                |        |        |          | Pos-Dis-Prefix: 88.1.1.1/32 |
|           |            |                |        |        |          |   Metric: 2                 |
+-----------+------------+----------------+--------+--------+----------+-----------------------------+

Negative Disaggregation TIEs:
+-----------+------------+------+--------+--------+----------+----------+
| Direction | Originator | Type | TIE Nr | Seq Nr | Lifetime | Contents |
+-----------+------------+------+--------+--------+----------+----------+
```

6.5.14 Fabric Bandwidth Balancing

6.5.14.1 show bandwidth-balancing

The `show bandwidth-balancing` command shows how northbound traffic is balanced across the northbound neighbors. This is a function of the ingress and the egress bandwidth of each northbound neighbor as well as the bandwidth of each northbound interface.

```
leaf> show bandwidth-balancing
North-Bound Neighbors:
+-----------+-----------+-----------+------------+-----------+-----------+------------+
| System ID | Neighbor  | Neighbor  | Neighbor   | Interface | Interface | Interface  |
|           | Ingress   | Egress    | Traffic    | Name      | Bandwidth | Traffic    |
|           | Bandwidth | Bandwidth | Percentage |           |           | Percentage |
+-----------+-----------+-----------+------------+-----------+-----------+------------+
| 2         | 300 Mbps  | 300 Mbps  | 50.0 %     | if1       | 100 Mbps  | 16.7 %     |
|           |           |           |            | if2       | 100 Mbps  | 16.7 %     |
|           |           |           |            | if3       | 100 Mbps  | 16.7 %     |
+-----------+-----------+-----------+------------+-----------+-----------+------------+
| 3         | 300 Mbps  | 300 Mbps  | 50.0 %     | if4       | 100 Mbps  | 16.7 %     |
|           |           |           |            | if5       | 100 Mbps  | 16.7 %     |
|           |           |           |            | if6       | 100 Mbps  | 16.7 %     |
+-----------+-----------+-----------+------------+-----------+-----------+------------+
```

6.5.15 Statistics

6.5.15.1 clear engine statistics

The `clear engine statistics` command resets the engine statistics counters back to zero:

```
agg_101> clear engine statistics
```

6.5.15.2 clear interface <interface> statistics

The `clear interface <interface> statistics` command resets the statistics counters for the specified interface back to zero:

```
agg_101> clear interface if_101_1 statistics
```

6.5.15.3 clear node statistics

The `clear node statistics` command resets the statistics counters for the current node back to zero:

```
agg_101> clear node statistics
```

6.5.15.4. show engine statistics

The `show engine statistics` command shows the engine statistics counters. Since there are very many counters, the output is very large. The `show engine statistics exclude-zero` variation excludes all zero counters, which reduces the amount of output and allows you to focus on the interesting counters.

```
agg_101> show engine statistics
All Node ZTP FSMs:
+------------------------------------------------------------------------------+-------------------+------------------------+-------------------+
| Description                                                                  | Value             | Last Rate              | Last Change       |
|                                                                              |                   | Over Last 10 Changes   |                   |
+------------------------------------------------------------------------------+-------------------+------------------------+-------------------+
| Events CHANGE_LOCAL_CONFIGURED_LEVEL                                         | 0 Events          |                        |                   |
+------------------------------------------------------------------------------+-------------------+------------------------+-------------------+
| Events NEIGHBOR_OFFER                                                        | 22032 Events      | 203.17 Events/Sec      | 0d 00h:00m:00.97s |
+------------------------------------------------------------------------------+-------------------+------------------------+-------------------+
| Events BETTER_HAL                                                            | 0 Events          |                        |                   |
+------------------------------------------------------------------------------+-------------------+------------------------+-------------------+
...
+------------------------------------------------------------------------------+-------------------+------------------------+-------------------+
| Event-Transitions HOLDING_DOWN -[HOLD_DOWN_EXPIRED]-> COMPUTE_BEST_OFFER     | 0 Transitions     |                        |                   |
+------------------------------------------------------------------------------+-------------------+------------------------+-------------------+

All Interfaces Traffic:
+----------------------------+------------------------------+----------------------------------------+-------------------+
| Description                | Value                        | Last Rate                              | Last Change       |
|                            |                              | Over Last 10 Changes                   |                   |
+----------------------------+------------------------------+----------------------------------------+-------------------+
| RX IPv4 LIE Packets        | 11016 Packets, 1858976 Bytes | 107.72 Packets/Sec, 18503.42 Bytes/Sec | 0d 00h:00m:00.97s |
+----------------------------+------------------------------+----------------------------------------+-------------------+
| TX IPv4 LIE Packets        | 11016 Packets, 1858976 Bytes | 108.42 Packets/Sec, 18623.58 Bytes/Sec | 0d 00h:00m:00.98s |
+----------------------------+------------------------------+----------------------------------------+-------------------+
...
+----------------------------+------------------------------+----------------------------------------+-------------------+
| Total RX IPv6 Misorders    | 0 Packets                    |                                        |                   |
+----------------------------+------------------------------+----------------------------------------+-------------------+
| Total RX Misorders         | 0 Packets                    |                                        |                   |
+----------------------------+------------------------------+----------------------------------------+-------------------+

All Interfaces Security:
+-------------------------------------------------+------------------------------+----------------------------------------+-------------------+
| Description                                     | Value                        | Last Rate                              | Last Change       |
|                                                 |                              | Over Last 10 Changes                   |                   |
+-------------------------------------------------+------------------------------+----------------------------------------+-------------------+
| Missing outer security envelope                 | 0 Packets, 0 Bytes           |                                        |                   |
+-------------------------------------------------+------------------------------+----------------------------------------+-------------------+
| Zero outer key id not accepted                  | 0 Packets, 0 Bytes           |                                        |                   |
+-------------------------------------------------+------------------------------+----------------------------------------+-------------------+
...
+-------------------------------------------------+------------------------------+----------------------------------------+-------------------+
| Empty outer fingerprint accepted                | 27551 Packets, 7291075 Bytes | 201.24 Packets/Sec, 34613.04 Bytes/Sec | 0d 00h:00m:00.98s |
+-------------------------------------------------+------------------------------+----------------------------------------+-------------------+
| Empty origin fingerprint accepted               | 3 Packets, 1047 Bytes        | 26.12 Packets/Sec, 9116.43 Bytes/Sec   | 0d 00h:05m:44.98s |
+-------------------------------------------------+------------------------------+----------------------------------------+-------------------+

All Interface LIE FSMs:
+------------------------------------------------------------+-------------------+------------------------+-------------------+
| Description                                                | Value             | Last Rate              | Last Change       |
|                                                            |                   | Over Last 10 Changes   |                   |
+------------------------------------------------------------+-------------------+------------------------+-------------------+
| Events TIMER_TICK                                          | 11016 Events      | 109.17 Events/Sec      | 0d 00h:00m:00.99s |
+------------------------------------------------------------+-------------------+------------------------+-------------------+
| Events LEVEL_CHANGED                                       | 0 Events          |                        |                   |
+------------------------------------------------------------+-------------------+------------------------+-------------------+
...
+------------------------------------------------------------+-------------------+------------------------+-------------------+
| Event-Transitions THREE_WAY -[SEND_LIE]-> THREE_WAY        | 11016 Transitions | 108.23 Transitions/Sec | 0d 00h:00m:00.98s |
+------------------------------------------------------------+-------------------+------------------------+-------------------+
| Event-Transitions TWO_WAY -[TIMER_TICK]-> TWO_WAY          | 0 Transitions     |                        |                   |
+------------------------------------------------------------+-------------------+------------------------+-------------------+
```

6.5.15.5 show interface <interface> statistics

The `show interface <interface> statistics` command shows the statistics counters for the specified interface. The `show interface <interface> statistics exclude-zero` variation excludes all zero counters.

The output is very similar to the output of `show engine statistics` command; see section 6.5.15.4 for an example output.

6.5.15.6 show node statistics

The `show node statistics` command shows the statistics counters for the current node. The `show node statistics exclude-zero` variation excludes all zero counters.

The output is very similar to the output of `show engine statistics` command; see section 6.5.15.4 for an example output.

6.5.16 Finite State Machines (FSMs)

RIFT-Python uses formal finite state machines (FSMs) for the following two things:

- There is one per-node finite state machine for ZTP.
- There is one per-interface finite state machine for processing LIE messages and establishing adjacencies.

6.5.16.1 show fsm lie

The `show fsm lie` command shows the formal definition of the per-interface finite state machine for processing LIE messages and establishing adjacencies.

```
agg_101> show fsm lie
States:
+-----------+
| State     |
+-----------+
| ONE_WAY   |
+-----------+
| TWO_WAY   |
+-----------+
| THREE_WAY |
+-----------+

Events:
+--------------------------------+---------+
| Event                          | Verbose |
+--------------------------------+---------+
| TIMER_TICK                     | True    |
+--------------------------------+---------+
| LEVEL_CHANGED                  | False   |
+--------------------------------+---------+
...
```

```
+-------------------------------+---------+
| LIE_CORRUPT                   | False   |
+-------------------------------+---------+
| SEND_LIE                      | True    |
+-------------------------------+---------+

Transitions:
+------------+-----------------------------+-----------+-------------------------+-------------+
| From state | Event                       | To state  | Actions                 | Push events |
+------------+-----------------------------+-----------+-------------------------+-------------+
| ONE_WAY    | TIMER_TICK                  | -         | -                       | SEND_LIE    |
+------------+-----------------------------+-----------+-------------------------+-------------+
| ONE_WAY    | LEVEL_CHANGED               | ONE_WAY   | update_level            | SEND_LIE    |
+------------+-----------------------------+-----------+-------------------------+-------------+
| ONE_WAY    | HAL_CHANGED                 | -         | store_hal               | -           |
+------------+-----------------------------+-----------+-------------------------+-------------+
| ONE_WAY    | HAT_CHANGED                 | -         | store_hat               | -           |
+------------+-----------------------------+-----------+-------------------------+-------------+
...
+------------+-----------------------------+-----------+-------------------------+-------------+
| THREE_WAY  | MULTIPLE_NEIGHBORS          | ONE_WAY   | -                       | -           |
+------------+-----------------------------+-----------+-------------------------+-------------+
| THREE_WAY  | LIE_CORRUPT                 | ONE_WAY   | -                       | -           |
+------------+-----------------------------+-----------+-------------------------+-------------+
| THREE_WAY  | SEND_LIE                    | -         | send_lie                | -           |
+------------+-----------------------------+-----------+-------------------------+-------------+

State entry actions:
+-----------+---------------------+-------------------------+
| State     | Entry Actions       | Exit Actions            |
+-----------+---------------------+-------------------------+
| ONE_WAY   | cleanup             | increase_tx_nonce_local |
|           | send_lie            |                         |
+-----------+---------------------+-------------------------+
| THREE_WAY | start_flooding      | increase_tx_nonce_local |
|           | init_partially_conn | stop_flooding           |
|           |                     | clear_partially_conn    |
+-----------+---------------------+-------------------------+
| TWO_WAY   | -                   | increase_tx_nonce_local |
+-----------+---------------------+-------------------------+
```

6.5.16.2 show fsm ztp

The `show fsm ztp` command shows the formal definition of the per-node finite state machine for ZTP.

The output of the `show fsm ztp` command is very similar to the output of the `show fsm lie` command; see section 6.5.16.1 for an example output.

6.5.16.3 show interface <interface> fsm history

The `show interface <interface> fsm history` command shows the 25 most recent events in the LIE FSM for the specified interface.

By default, this command only shows ‘interesting’ events; ‘verbose’ events that occur very frequently under normal circumstances (for example, events TIMER_TICK, SEND_LIE, and LIE_RECEIVED) are excluded because they tend to dominate the history log and obscure more interesting events. The `show interface <interface> fsm verbose-history` shows all recent events, including the verbose ones. You can see which events are considered to be verbose in the output of `show fsm lie`.

```
agg_101> show interface if_101_1 fsm history
+----------+-------------+---------+---------+---------------------+-------------------------+-----------+----------+
| Sequence | Time        | Verbose | From    | Event               | Actions and             | To        | Implicit |
| Nr       | Delta       | Skipped | State   |                     | Pushed Events           | State     |          |
+----------+-------------+---------+---------+---------------------+-------------------------+-----------+----------+
| 317      | 1025.263305 | 2       | TWO_WAY | VALID_REFLECTION    | increase_tx_nonce_local | THREE_WAY | False    |
|          |             |         |         |                     | start_flooding          |           |          |
|          |             |         |         |                     | init_partially_conn     |           |          |
+----------+-------------+---------+---------+---------------------+-------------------------+-----------+----------+
| 314      | 0.001905    | 3       | ONE_WAY | NEW_NEIGHBOR        | SEND_LIE                | TWO_WAY   | False    |
|          |             |         |         |                     | increase_tx_nonce_local |           |          |
+----------+-------------+---------+---------+---------------------+-------------------------+-----------+----------+
| 154      | 0.165887    | 1       | ONE_WAY | UNACCEPTABLE_HEADER |                         | ONE_WAY   | False    |
+----------+-------------+---------+---------+---------------------+-------------------------+-----------+----------+
| 152      | 0.000698    | 1       | ONE_WAY | UNACCEPTABLE_HEADER |                         | ONE_WAY   | False    |
+----------+-------------+---------+---------+---------------------+-------------------------+-----------+----------+
| 11       | 0.940489    | 0       | None    | None                | cleanup                 | ONE_WAY   | False    |
|          |             |         |         |                     | send_lie                |           |          |
+----------+-------------+---------+---------+---------------------+-------------------------+-----------+----------+
```

6.5.16.4 show node fsm history

The `show node fsm history` command shows the 25 most recent events in the ZTP FSM for the current node.

By default, this command only shows ‘interesting’ events; ‘verbose’ events that occur very frequently under normal circumstances (for example the event NEIGHBOR_OFFER) are excluded because they tend to dominate the history log and obscure more interesting events. The `show node fsm verbose-history` shows all recent events, including the verbose ones. You can see which events are considered to be verbose in the output of `show fsm ztp`.

```
agg_101> show node fsm history
+----------+-------------+---------+--------------------+------------------+----------------------+--------------------+----------+
| Sequence | Time        | Verbose | From               | Event            | Actions and          | To                 | Implicit |
| Nr       | Delta       | Skipped | State              |                  | Pushed Events        | State              |          |
+----------+-------------+---------+--------------------+------------------+----------------------+--------------------+----------+
| 222      | 1717.683768 | 0       | COMPUTE_BEST_OFFER | COMPUTATION_DONE | update_all_lie_fsms  | UPDATING_CLIENTS   | False    |
+----------+-------------+---------+--------------------+------------------+----------------------+--------------------+----------+
| 221      | 0.000415    | 5       | UPDATING_CLIENTS   | BETTER_HAT       | stop_hold_down_timer | COMPUTE_BEST_OFFER | False    |
|          |             |         |                    |                  | level_compute        |                    |          |
|          |             |         |                    |                  | COMPUTATION_DONE     |                    |          |
+----------+-------------+---------+--------------------+------------------+----------------------+--------------------+----------+
| 159      | 0.071085    | 0       | COMPUTE_BEST_OFFER | COMPUTATION_DONE | update_all_lie_fsms  | UPDATING_CLIENTS   | False    |
+----------+-------------+---------+--------------------+------------------+----------------------+--------------------+----------+
| 158      | 0.000544    | 5       | UPDATING_CLIENTS   | BETTER_HAL       | stop_hold_down_timer | COMPUTE_BEST_OFFER | False    |
|          |             |         |                    |                  | level_compute        |                    |          |
|          |             |         |                    |                  | COMPUTATION_DONE     |                    |          |
+----------+------------+---------+-------------------+-----------------+---------------------+-------------------+----------+
| 45       | 0.736157   | 0       | COMPUTE_BEST_OFFER | COMPUTATION_DONE | update_all_lie_fsms  | UPDATING_CLIENTS   | False    |
+----------+------------+---------+-------------------+-----------------+---------------------+-------------------+----------+
| 15       | 0.184907   | 0       | None              | None            | stop_hold_down_timer | COMPUTE_BEST_OFFER | False    |
|          |            |         |                   |                 | level_compute        |                    |          |
|          |            |         |                   |                 | COMPUTATION_DONE     |                    |          |
+----------+------------+---------+-------------------+-----------------+---------------------+-------------------+----------+
```

6.5.17 Simulated Failures

6.5.17.1 set interface <interface> failure <failure>

The `set interface <interface> failure <failure>` command is used to simulate a failure of an interface.

The `failure` parameter can be one of the following values:

- `ok`: The interface is ok, therefore repaired.
- `Failed`: The interface is failed bi-directionally in both the transmit and receive direction.
- `rx-failed`: The interface is failed unidirectionally in only the receive direction.
- `tx-failed`: The interface is failed unidirectionally in only the transmit direction.

This command is useful in automated unit test scripts that use the single-process simulation approach where all simulated interfaces run on top of a single physical interface. In the multi-process simulation approach, a more realistic way of simulating failures is to shut down the virtual Ethernet (veth) interface. The latter approach is used in chaos testing (see section 6.6.2). See section 6.3 for more details on the difference between the single-process and the multi-process approach.

```
agg_101> set interface if_101_1 failure failed
```

6.6 Automated Testing

RIFT-Python includes extensive automated unit testing and system testing suites. All tests are implemented as Python scripts in the `tests` subdirectory using the pytest (https://docs.pytest.org) testing framework.

6.6.1 Automated Unit and System Testing

The test scripts starting with `test_sys_` are system tests; they test the behavior of RIFT-Python nodes running in a complete topology. The test scripts starting with `test_` (without `_sys_`) are unit tests; they test the behavior of an individual class or some subsystem with the RIFT-Python code.

Use the following pytest command to run an individual test, in this example the test_fsm.py unit test:

```
(env) $ pytest tests/test_fsm.py --verbose
============================ test session starts ============================
platform darwin -- Python 3.5.1, pytest-3.6.4, py-1.5.4, pluggy-0.7.1 -- /Users/brunorijsman/rift-python/env/bin/python3
cachedir: .pytest_cache
rootdir: /Users/brunorijsman/rift-python, inifile:
plugins: cov-2.5.1
collected 6 items

tests/test_fsm.py::test_states_table PASSED                           [ 16%]
tests/test_fsm.py::test_events_table PASSED                           [ 33%]
tests/test_fsm.py::test_transition_table PASSED                       [ 50%]
tests/test_fsm.py::test_state_actions_table PASSED                    [ 66%]
tests/test_fsm.py::test_history_table PASSED                          [ 83%]
tests/test_fsm.py::test_fsm_basic PASSED                              [100%]

========================= 6 passed in 0.05 seconds ==========================
```

The system test scripts (tests/test_sys_*.py) all work by performing the following steps in a fully automated manner:

1. Spawn one of the topologies (topology/*.yaml) in a RIFT-Python process using the single-process approach (see section 6.3.1).

2. Use a telnet session to attach to the CLI of the running RIFT-Python process.

3. Either simply wait for the static topology to converge, or dynamically fail and repair links using the set interface interface-name failure failure-mode command to test a dynamic reconvergence scenario.

4. Invoke various show commands and analyze (screen scrape) the output to check whether the network is behaving as expected.

5. Analyze the log file rift.log to check whether expected log messages are present.

Python module rift_expect_session.py and rift_log_session.py contain the common framework code to make automation of these steps easier.

Each system test appends to the files rift_expect.log and log_expect.log to help diagnose the root cause for potential test failures. File rift_expect.log contains every CLI command invoked during the test, the expected output, and the actual output. Similarly, file log_expect.log contains the expected log messages, and the actual log messages.

Here is an example of the contents of rift_expect.log when a test fails. In this example an adjacency which was expected to be up was actually down. The output includes a Python stack trace showing the exact location in the test script that failed.

```
*** Expect: [|] if1 +[|] .* +[|] .* +[|] THREE_WAY +[|]

show interfaces

+-----------+----------+-----------+----------+
| Interface | Neighbor | Neighbor  | Neighbor |
| Name      | Name     | System ID | State    |
+-----------+----------+-----------+----------+
| if1       |          |           | ONE_WAY  |
+-----------+----------+-----------+----------+

node1>

*** Did not find expected pattern [|] if1 +[|] .* +[|] .* +[|] THREE_WAY +[|]

File "/Users/brunorijsman/rift-python/tests/test_sys_2n_l0_l1.py", line 243, in test_2n_l0_l1
   check_rift_node1_intf_up(res)
File "/Users/brunorijsman/rift-python/tests/test_sys_2n_l0_l1.py", line 16, in check_rift_node1_intf_up
   interface="if1")
File "tests/rift_expect_session.py", line 151, in check_adjacency_3way
   self.table_expect("| {} | .* | .* | THREE_WAY |".format(interface))
File "tests/rift_expect_session.py", line 97, in table_expect
   return self.expect(pattern, timeout)
File "tests/rift_expect_session.py", line 84, in expect
   self.log_expect_failure(pattern)
File "tests/rift_expect_session.py", line 69, in log_expect_failure
   for line in traceback.format_stack():
```

Use the `tools/pre-commit-checks` shell script to not only run all test cases, but also to run pylint to check the code for errors, and to check whether each CLI command is properly documented. This command should be run before committing new code to GitHub to make sure that the continuous integration cycle will not fail (that you won't 'break the build'). The whole process took about eight minutes to complete on a 2016 MacBook.

```
(env) $ tools/cleanup
(env) $ tools/pre-commit-checks
macOS: increase number of file descriptors to 1024
Linting rift directory...

--------------------------------------------------------------------
Your code has been rated at 10.00/10 (previous run: 10.00/10, +0.00)

Linting tests directory...

--------------------------------------------------------------------
Your code has been rated at 10.00/10 (previous run: 10.00/10, +0.00)

Linting tools directory...

--------------------------------------------------------------------
Your code has been rated at 10.00/10 (previous run: 10.00/10, +0.00)
```

```
========================= test session starts =========================
platform darwin -- Python 3.5.1, pytest-3.6.4, py-1.5.4, pluggy-0.7.1
rootdir: /Users/brunorijsman/rift-python, inifile:
plugins: cov-2.5.1
collected 129 items / 1 deselected

tests/test_config_generator.py ....                             [  3%]
tests/test_constants.py ....                                    [  6%]
...
tests/test_timer.py .....                                       [ 99%]
tests/test_visualize_log.py .                                   [100%]

---------- coverage: platform darwin, python 3.5.1-final-0 -----------
Name                                         Stmts   Miss  Cover
----------------------------------------------------------------
common/__init__.py                               1      0   100%
common/constants.py                             33      0   100%
...
tools/config_generator.py                     1445    571    60%
----------------------------------------------------------------
TOTAL                                        17354   3705    79%


============= 128 passed, 1 deselected in 349.18 seconds ==============
All good; you can commit.
(env) $
```

6.6.2 Automated Chaos Testing

The concept of chaos testing was pioneered by Netflix. They famously introduced their so-called chaos monkey (https://netflix.github.io/chaosmonkey/) which randomly killed software service instances in their live production environment to make sure that their failure recovery infrastructure was robust.

RIFT-Python includes a chaos testing tool that randomly kills and repairs links and RIFT nodes and verifies that the RIFT protocol reconverges correctly. I use chaos testing extensively while developing RIFT-Python; it has helped me find numerous bugs and corner cases that I would never have found with more traditional unit and system testing methods. The continuous integration (CI) cycle, described in section 6.6.4, automatically invokes a round of chaos testing for every single code change that is committed in GitHub.

RIFT-Python's chaos testing tool is part of the `config_generator.py` script. As a reminder, `config_generator.py` takes a so-called meta-topology file, which is a very high-level description of the shape of the fabric, as input. And it produces a set of configuration files and scripts that are used to run that fabric using the multi-process approach as output.

One of many scripts that `config_generator.py` produces is `chaos.sh`, which does the actual chaos testing:

- It randomly fails links. This is implemented by using the Linux qdisc interface scheduler (https://tldp.org/HOWTO/Traffic-Control-HOWTO/components.html) to drop 100% of the traffic on the link in both directions. It randomly fails nodes. This is implemented by killing the RIFT-Python process.
- It randomly repairs links and nodes that failed previously in the script. The repairs are randomly interspersed with the failures, so that multiple failures may simultaneously occur during the chaos script. That said, the chaos script makes sure that all failures are repaired by the end of the script. Thus, the fabric is back in its original state when the chaos script finishes.

Future versions of RIFT-Python may introduce additional failure modes, for example unidirectional link failures, partial link failures (for example 10% traffic drop), link congestion (long delays), running out of memory, running out of CPU, etc.

While the chaos script is running multiple failures may be simultaneously present. Hence there is a small probability that the fabric might be bisected at some point in time, and that not all leafs are able to reach all other leafs. But since the topology is returned back to its original state (i.e. all failures are repaired) when the chaos script finishes, the fabric should reconverge correctly and in the end every leaf should be able to reach every other leaf again.

RIFT-Python provides an automated method for verifying that the topology has indeed reconverged correctly after the chaos script has completed. The `config_generator.py` script has an optional `--check` argument to run a series of automated reconvergence tests on the topology. This reconvergence check does much more than just ping every leaf from every other leaf: it attaches to the CLI of each node and performs a very extensive set of sanity checks. At the time of writing, these sanity checks include the following (more checks may be added over time):

- Each RIFT-Python process is running (i.e. did not crash) for every node.
- The CLI of each node is responsive: the output of the `show engine` command is correct.
- Each leaf node can ping every other leaf node. Note that the RIFT protocol does not guarantee that spine or superspine nodes can be pinged due to how the northbound ECMP default routes work.
- All interfaces have an adjacency in state ThreeWay.
- All nodes except the ToF nodes have a northbound default IPv4 and IPv6 route installed in the RIB.
- All nodes except the leaf nodes have specific /32 IPv4 and /128 IPv6 routes installed in the RIB for all loopbacks of all nodes south of them.
- The routes in the FIB are consistent with the routes in the RIB.

- The routes in the kernel are consistent with the routes in the FIB.
- No unexpected positive or negative disaggregation is happening.
- All queues (the TIE transmit queue, the TIE request queue, and the TIE ack queue) are empty.

Each time `config_generator.py` is run to produce a `chaos.sh` script, a new random sequence of failures and repairs is generated. Thus, you can generate multiple chaos scripts to test multiple different random scenarios. However, once a chaos script is generated, you can run the same script multiple times. This is very useful when the chaos script discovers a bug and you want to reproduce the exact same scenario.

The meta-topology YAML file (which we described earlier in section 6.4.2) may optionally contain a `chaos` section with the following attributes to control the generation of the `chaos.sh` script:

```
chaos:
  nr-link-events: int
  nr-node-events: int
  event-interval: float
  max-concurrent-events: int
```

Table 6.10 Meaning of These Chaos Attributes

Attribute	Meaning	Mandatory / Optional
nr-link-events	The number of link failure events in the generated chaos script. Currently a bidirectional link failure is the only type of link failure event, but other types of link failures may be added in the future.	Optional, default value 20.
nr-node-events	The number of node failure events in the generated chaos script. Currently killing the RIFT-Python process is the only type of node failure event, but other types of node failures may be added in the future.	Optional, default value 5.
event-interval	The delay, in seconds, between subsequent failure or repair events.	Optional, default value 3.0.
max-concurrent-events	The maximum number of concurrent failures (this includes both link and node failures).	Optional, default 5.

Now let's walk through an example to show you how to generate a fabric topology, generate a chaos test, run the chaos test, and check for correct reconvergence. You must use Linux (not macOS) to run this example because the multi-process simulation approach requires network namespaces and virtual Ethernet interfaces. Also, for larger topologies, such as the one use in this example, you need a beefy AWS instance. We recommend a m5a.large instance.

6.6.2.1 Step 1: Run config_generator.py to generate the topology and the chaos script

Log in to your Linux instance and install RIFT-Python as described in section 6.2 if you haven't already done so.

Go to the RIFT-Python directory, and elevate privileges to allow namespaces to be created (to keep things simple, we're using `sudo bash` in this example):

```
$ cd ~/rift-python
$ sudo bash
#
```

Activate the Python virtual environment:

```
# source env/bin/activate
(env) #
```

Have a look at the meta-topology file we will be using in this example. Note that this example meta-topology file does not contain any chaos testing attributes; we rely on the default values.

```
(env) # cat meta_topology/clos_3plane_3pod_3leaf_3spine_6super.yaml
nr-pods: 3
nr-leaf-nodes-per-pod: 3
nr-spine-nodes-per-pod: 3
nr-superspine-nodes: 6
nr-planes: 3
```

Run the `topology_generator.py` script to generate all configuration files and all scripts:

```
(env) # tools/config_generator.py --netns-per-node --graphics-file diagram.html meta_topology/clos_3plane_3pod_3leaf_3sp
ine_6super.yaml generated
```

The `--netns-per-node` option enables the multi-process simulation approach (also known as *network namespace per node approach*).

The `--graphics-file diagram.html` option writes a scalable vector graphics (SVG) diagram of the topology to the file `diagram.html` which can be viewed using a web browser.

The `generated` argument specifies the directory into which all generated configuration files and scripts are written (this directory must not already exist).

Use a browser to open the generated `diagram.html` file and you'll see the topology as shown in Figure 6.9.

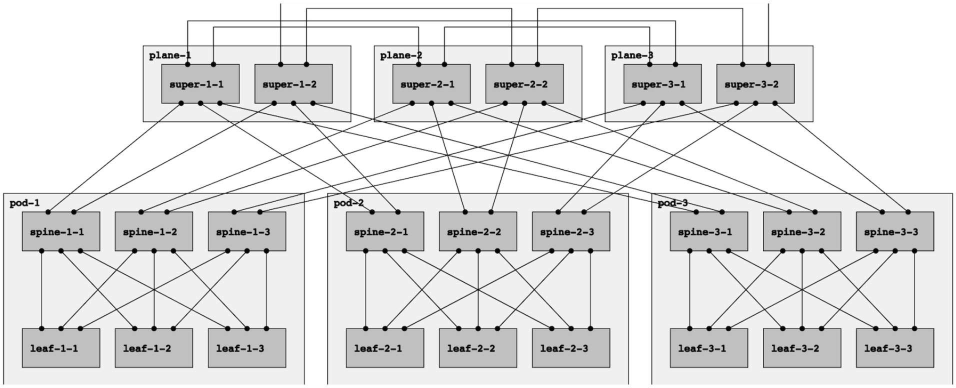


Figure 6.9 Topology Used in Chaos Testing Example

You will probably have to copy the diagram.html file from the AWS instance to your local laptop first before you can view it using a local browser:

```
$ scp -i ~/.ssh/private-key-file.pem ubuntu@vm-ip-address:rift-python/diagram.html .
```

Have a look at the generated configuration files and scripts:

```
(env) # ls -1 generated
```

6.6.2.2 Step 2: Start the topology

Start the topology by running the generated `start.sh` script:

```
(env) # generated/start.sh
Create veth pair veth-1a-101d and veth-101d-1a for link from super-1-1:if-1a to spine-1-1:if-101d
Create veth pair veth-1b-104d and veth-104d-1b for link from super-1-1:if-1b to spine-2-1:if-104d
...
Create network namespace netns-1 for node super-1-1
Create network namespace netns-2 for node super-1-2
...
Start RIFT-Python engine for node super-1-1
Start RIFT-Python engine for node super-1-2
...
```

At this point all RIFT nodes in the topology are running as background processes. You can attach to the CLI of any node using a generated `connect-node-name.sh` script:

```
# generated/connect-spine-1-1.sh
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
spine-1-1> show interfaces
+-----------------+--------------------------+-----------+-----------+-------------------+-------+
| Interface       | Neighbor                 | Neighbor  | Neighbor  | Time in           | Flaps |
| Name            | Name                     | System ID | State     | State             |       |
+-----------------+--------------------------+-----------+-----------+-------------------+-------+
| veth-101a-1001a | leaf-1-1:veth-1001a-101a | 1001      | THREE_WAY | 0d 00h:00m:16.00s | 0     |
+-----------------+--------------------------+-----------+-----------+-------------------+-------+
| veth-101b-1002a | leaf-1-2:veth-1002a-101b | 1002      | THREE_WAY | 0d 00h:00m:16.08s | 0     |
+-----------------+--------------------------+-----------+-----------+-------------------+-------+
| veth-101c-1003a | leaf-1-3:veth-1003a-101c | 1003      | THREE_WAY | 0d 00h:00m:15.92s | 0     |
+-----------------+--------------------------+-----------+-----------+-------------------+-------+
| veth-101d-1a    | super-1-1:veth-1a-101d   | 1         | THREE_WAY | 0d 00h:00m:15.59s | 0     |
+-----------------+--------------------------+-----------+-----------+-------------------+-------+
| veth-101e-2a    | super-1-2:veth-2a-101e   | 2         | THREE_WAY | 0d 00h:00m:16.50s | 0     |
+-----------------+--------------------------+-----------+-----------+-------------------+-------+
```

And use `exit` to return to the Linux shell:

```
spine-1-1> exit
Connection closed by foreign host.
(env) #
```

6.6.2.3 Step 3: Run the chaos script

Run the generated `chaos.sh` script to actually start the chaos testing:

```
(env) # generated/chaos.sh
Break  Link  if-1001b-if-102a (bi-directional failure)
Break  Node  spine-1-2
Break  Node  spine-2-3
Break  Link  if-3e-if-5d (bi-directional failure)
Fix    Node  spine-2-3
Break  Link  if-6b-if-106e (bi-directional failure)
Fix    Link  if-6b-if-106e
...
Break  Link  if-1001b-if-102a (bi-directional failure)
Fix    Link  if-1001b-if-102a
Fix    Link  if-1008a-if-107b
```

Lines starting with `Break` indicate that a link or node is broken, lines starting with `Fix` indicate that a previously broken link or node is repaired. Observe that the script makes sure that everything is repaired before it ends.

Since the chaos script is randomly generated, you will see a different sequence of failures and repairs from the ones listed above. But if you run the same script twice you will see the exact same sequence of failures and repairs each run (this is useful to reproduce any issues discovered by chaos testing).

6.6.2.4 Step 4: Check correct reconvergence

Run the `config_generator.py` script again, this time with the `--check` option, to verify that the topology correctly reconverged after the chaos script finished:

```
(env) # tools/config_generator.py --netns-per-node --check meta_topology/clos_3plane_3pod_3leaf_3spine_6super.yaml
**** Check node leaf-1-1
OK    RIFT process is running
OK    Can Telnet to RIFT process
OK    RIFT engine is responsive
OK    This leaf can ping all other leaves
OK    Adjacencies are 3-way
OK    North-bound default routes are present
OK    South-bound specific routes are present
OK    RIB and FIB are consistent
OK    FIB and Kernel are consistent
OK    There is no disaggregation
OK    The TIE/TIRE queues are empty
...
**** Check node super-3-2
OK    RIFT process is running
OK    Can Telnet to RIFT process
OK    RIFT engine is responsive
OK    Adjacencies are 3-way
OK    South-bound specific routes are present
OK    RIB and FIB are consistent
OK    FIB and Kernel are consistent
OK    There is no disaggregation
OK    The TIE/TIRE queues are empty
All checks passed successfully
```

If there is a problem you will see something similar to this (we manually broke a link to cause these failures):

```
**** Check node leaf-1-1
OK    RIFT process is running
OK    Can Telnet to RIFT process
OK    RIFT engine is responsive
OK    This leaf can ping all other leaves
FAIL  Adjacencies are 3-way: Interface veth-1001a-101a in state ONE_WAY
FAIL  Adjacencies are 3-way
FAIL  Route prefix 0.0.0.0/0 owner North SPF nexthops ['veth-1001a-101a 99.1.2.2', 'veth-1001c-103a 99.5.6.6', 'veth-
1001b-102a 99.3.4.4'] in RIB: Nexthops mismatch; expected ['veth-1001a-101a 99.1.2.2', 'veth-1001b-
102a 99.3.4.4', 'veth-1001c-103a 99.5.6.6'] but RIB has ['veth-1001b-102a 99.3.4.4', 'veth-1001c-103a 99.5.6.6']
FAIL  Route prefix ::/0 owner North SPF nexthops ['veth-1001b-102a', 'veth-1001a-101a', 'veth-1001c-
103a'] in RIB: Nexthops mismatch; expected ['veth-1001a-101a', 'veth-1001b-102a', 'veth-1001c-103a'] but RIB has ['veth-
1001b-102a', 'veth-1001c-103a']
FAIL  North-bound default routes are present
OK    South-bound specific routes are present
OK    RIB and FIB are consistent
OK    FIB and Kernel are consistent
FAIL  There is no disaggregation: Unexpected Pos-Dis-Prefix TIE: direction=South originator=102 tie-nr=3 seq-
nr=5 contents=Pos-Dis-Prefix: 88.0.1.1/32
FAIL  There is no disaggregation: Unexpected Pos-Dis-Prefix TIE: direction=South originator=103 tie-nr=3 seq-
nr=7 contents=Pos-Dis-Prefix: 88.0.1.1/32
FAIL  The TIE/TIRE queues are empty: Unexpected entry in TIE TX queue: interface=veth-1001a-
101a direction=North originator=1001 tie-type=Node tie-nr=1 seq-nr=9
```

To debug problems discovered by chaos testing, you can manually edit the generated chaos.sh script to stop at the relevant step, re-run the script, and then use the CLI or the Python debugger to diagnose the issue.

For more information on RIFT chaos testing see:

- The slide deck that was presented at IETF 104: https://datatracker.ietf.org/meeting/104/materials/slides-104-rift-hackathon-chaos-monkey-testing-02.pdf
- A live demonstration of chaos testing in YouTube video: https://www.youtube.com/watch?v=GqebgPmA4Xc.
- The RIFT-Python documentation for `config_generator.py`: https://github.com/brunorijsman/rift-python/blob/master/doc/configuration-file-generator.md

6.6.3 Automated Interoperability Testing

One of the main goals of the RIFT-Python project was to create a second implementation of RIFT that was developed completely independently of the first implementation by Juniper. This would be good evidence that the RIFT IETF draft was sufficiently complete and precise to allow implementations of multiple interoperable implementations from scratch.

RIFT-Python includes a script `tests/interop.py` for doing fully automated interoperability testing with the Juniper RIFT implementation. The script works as follows:

- It runs each system test in the automated system test suite (`tests/test_sys_*`) multiple times.
- During each run, it chooses a subset of RIFT routers in the topology and runs those chosen routers using Juniper RIFT. This involves translating the configuration for the RIFT-Python router to equivalent configuration for the Juniper RIFT router.
- The remaining routers still run RIFT-Python. All test cases are executed and verified, for as far as they can be verified using `show` commands on RIFT-Python routers.

Here is an example output of running the interop script:

```
(env) $ tests/interop.py
keys_match-node1... Pass
keys_match-node2... Pass
keys_match-node3... Pass
...
2n_un_l2-node2... Pass
2n_un_l0-node1... Pass
2n_un_l0-node2... Pass
```

MORE? You need access to Juniper's free trial implementation of RIFT to run the interoperability test. See https://github.com/brunorijsman/rift-python/blob/master/ietf-102/ietf-102-rift-hackathon-detailed-report.md for more details.

6.6.4 Continuous Integration (CI)

Every time a code change is committed to GitHub, Travis (https://travis-ci.org/) automatically runs a continuous integration (CI) test suite, which consists of (see file `.travis.yml`):

1. Run the automated unit test suite and system test suite.
2. Lint the code using `pylint`.

3. Check the completeness of the CLI documentation.

4. Run a chaos test (see section 6.6.2).

The history of CI test results are posted on https://travis-ci.org/github/brunorijsman/rift-python.

6.6.5 Code Coverage Measurement

Programming in Python, especially for exploratory projects such as RIFT-Python, is fun. The simplicity and elegance of Python allows you to convert ideas into code naturally and instantaneously with a minimum of fuss or boilerplate. The absence of a compile / build phase boosts productivity by enabling very tight code-run-debug-fix cycles. However, there is one downside to Python: because many types of errors only get detected at run-time rather than at compile time, it is essential to invest in an automated test suite with close to 100% code coverage. Another technique to reduce run-time errors is to add type annotations to Python code – we haven't yet done this for RIFT-Python, but it is on the authors' wish lists).

Code coverage simply means measuring which parts of the code get executed while running a test. If a certain part of the code never gets executed during any of the tests, you cannot be confident that it is actually bug-free. Code coverage can be measured at different levels of granularity: at the line level or at the branch level. We chose to measure it at the line level.

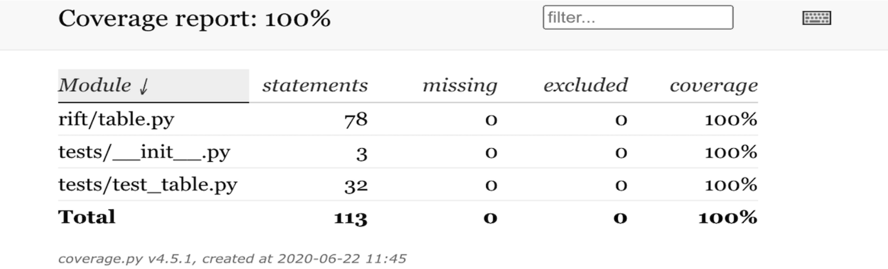
We already discussed the automated test framework in section 6.6.1 and the continuous integration cycle in section 6.6.3, so now let's describe how both of these are integrated with code coverage measurement.

Use the following command to do a code coverage measurement while running an individual test (in this example the unit test test_table.py):

```
(env) $ pytest tests/test_table.py --cov --cov-report=html
================================== test session starts ==================================
platform darwin -- Python 3.5.1, pytest-3.6.4, py-1.5.4, pluggy-0.7.1
rootdir: /Users/brunorijsman/rift-python, inifile:
plugins: cov-2.5.1
collected 4 items

tests/test_table.py ....                                                          [100%]

---------- coverage: platform darwin, python 3.5.1-final-0 -----------
Coverage HTML written to dir htmlcov

=============================== 4 passed in 0.07 seconds ================================
```

The `--cov` option enables code coverage measurement, and the `--cov-report=html` option writes the coverage measurement results into an HTML file that you can view using a web browser (see Figure 6.10a).

```
(env) $ open htmlcov/index.html
```



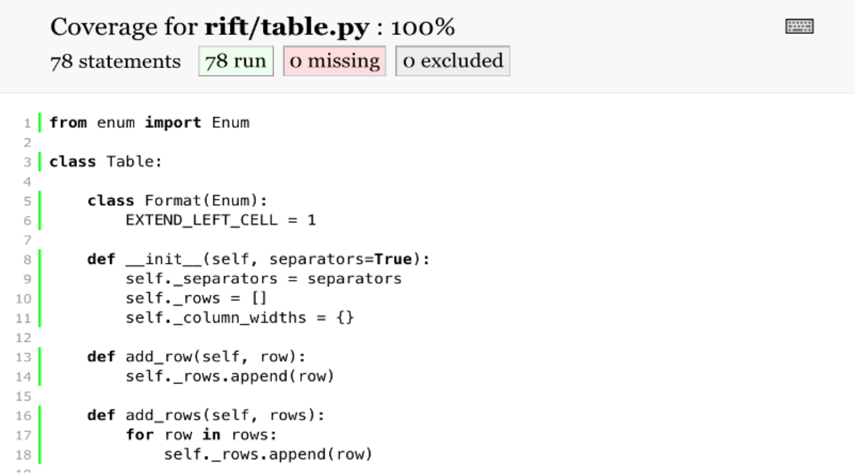
Coverage report: 100%

Module ↓	statements	missing	excluded	coverage
rift/table.py	78	0	0	100%
tests/__init__.py	3	0	0	100%
tests/test_table.py	32	0	0	100%
Total	**113**	**0**	**0**	**100%**

coverage.py v4.5.1, created at 2020-06-22 11:45

Figure 6.10a Code Coverage Report

Click on any module name to see line-by-line coverage results as in Figure 6.10b.



Coverage for **rift/table.py** : 100%

78 statements 78 run 0 missing 0 excluded

```
from enum import Enum

class Table:

    class Format(Enum):
        EXTEND_LEFT_CELL = 1

    def __init__(self, separators=True):
        self._separators = separators
        self._rows = []
        self._column_widths = {}

    def add_row(self, row):
        self._rows.append(row)

    def add_rows(self, rows):
        for row in rows:
            self._rows.append(row)
```

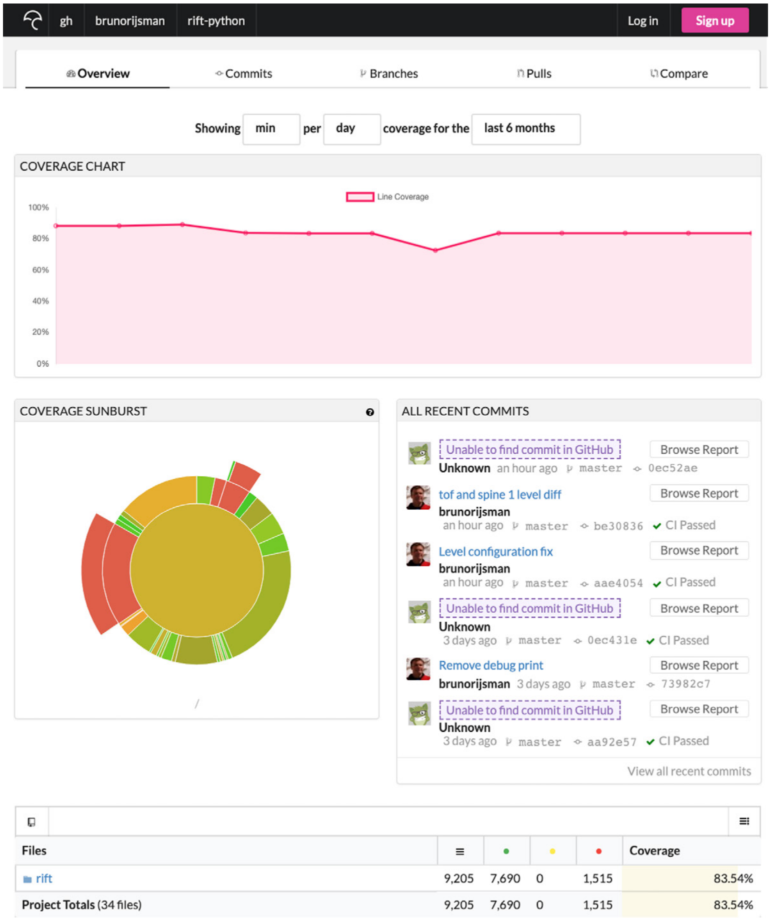
Figure 6.10b Line-by-line Code Coverage

The `tools/pre-commit-checks` script mentioned in section 6.6.1 also produces a code coverage report in `htmlcov/index.html` for the entire test suite rather than one individual test.

Coverage measurement results are collected incrementally over multiple runs in the hidden files that start with `.coverage`. Before you start a fresh series of converge measurements, it is advisable to remove the collected results from previous measurement series first. It is best to use the `cleanup` script for this purpose, because it also removes log files etc. from previous runs and avoids accidentally removing `.coveragerc` which is the configuration file for code coverage.

```
(env) $ tools/cleanup
```

The continuous integration process described in section 6.6.4 also measures code coverage when it runs the full test suite on Travis. The historical code coverage reports are posted on the codecov.io website (https://codecov.io/gh/brunorijsman/rift-python) which, frankly, produces much nicer diagrams that those produced by pytest, as in Figure 6.11.



Showing min per day coverage for the last 6 months

Unable to find commit in GitHub
Unknown an hour ago master 0ec52ae

tof and spine 1 level diff
brunorijsman
an hour ago master be30836 ✓ CI Passed

Level configuration fix
brunorijsman
an hour ago master aae4054 ✓ CI Passed

Unable to find commit in GitHub
Unknown
3 days ago master 0ec431e ✓ CI Passed

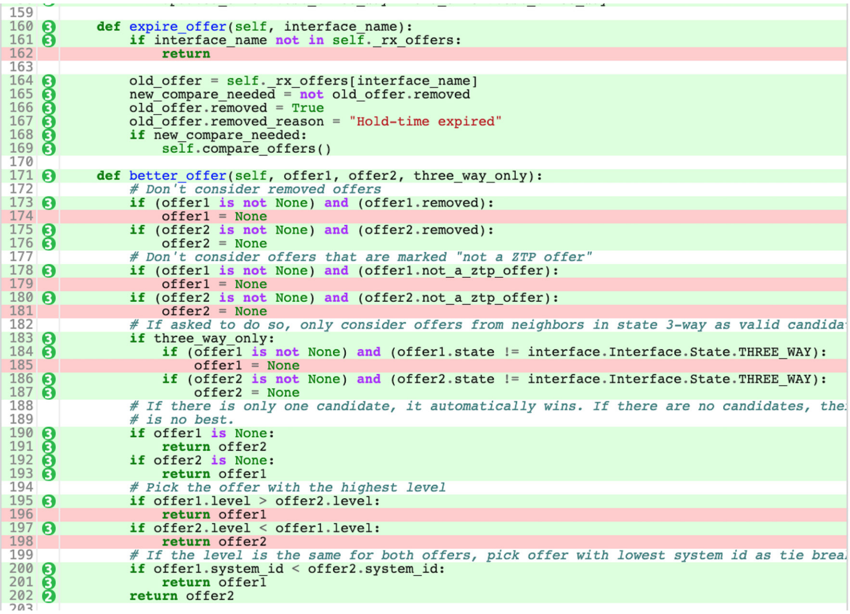
Remove debug print
brunorijsman 3 days ago master 73982c7

Unable to find commit in GitHub
Unknown
3 days ago master aa92e57 ✓ CI Passed

Files	≡	●	●	●	Coverage
rift	9,205	7,690	0	1,515	83.54%
Project Totals (34 files)	9,205	7,690	0	1,515	83.54%

Figure 6.11 *Nicer Diagrams*

Codecov also allows you to drill down into individual files, as in Figure 6.12.



```
    def expire_offer(self, interface_name):
        if interface_name not in self._rx_offers:
            return

        old_offer = self._rx_offers[interface_name]
        new_compare_needed = not old_offer.removed
        old_offer.removed = True
        old_offer.removed_reason = "Hold-time expired"
        if new_compare_needed:
            self.compare_offers()

    def better_offer(self, offer1, offer2, three_way_only):
        # Don't consider removed offers
        if (offer1 is not None) and (offer1.removed):
            offer1 = None
        if (offer2 is not None) and (offer2.removed):
            offer2 = None
        # Don't consider offers that are marked "not a ZTP offer"
        if (offer1 is not None) and (offer1.not_a_ztp_offer):
            offer1 = None
        if (offer2 is not None) and (offer2.not_a_ztp_offer):
            offer2 = None
        # If asked to do so, only consider offers from neighbors in state 3-way as valid candida
        if three_way_only:
            if (offer1 is not None) and (offer1.state != interface.Interface.State.THREE_WAY):
                offer1 = None
            if (offer2 is not None) and (offer2.state != interface.Interface.State.THREE_WAY):
                offer2 = None
        # If there is only one candidate, it automatically wins. If there are no candidates, the
        # is no best.
        if offer1 is None:
            return offer2
        if offer2 is None:
            return offer1
        # Pick the offer with the highest level
        if offer1.level > offer2.level:
            return offer1
        if offer2.level < offer1.level:
            return offer2
        # If the level is the same for both offers, pick offer with lowest system id as tie brea
        if offer1.system_id < offer2.system_id:
            return offer1
        return offer2
```

Figure 6.12 Drilling Down

6.6.6 Code Profiling

The original goal of the RIFT-Python project was to help the RIFT standardization process in the IETF. We wanted to see if it was possible to create a second implementation of RIFT that would interoperate with Juniper's implementation, using only the information in the IETF draft. Once that goal was achieved, the RIFT-Python project continued to evolve into an open source reference implementation of RIFT that allows anyone to get hands-on experience with RIFT.

Because of this history raw performance was never a primary goal for RIFT-Python. Indeed, single-threaded Python code is not exactly the best choice for maximum performance. Python was chosen because it is very easy to understand and enables a very fast development cycle.

Still, we can't completely ignore performance in RIFT-Python. We want to be able to test large realistic topologies. And RIFT-Python could be considered as a host router in production (although, as far as I know, no one has done that yet at the time this book was written).

While some a-priori common sense optimizations are useful (for example, avoiding linear searches in large lists), in my experience premature optimization does more harm than good by complexifying the code without significantly improving the performance. The only sensible way to optimize the code is to actually measure the performance and to actually measure where the major bottlenecks are first. Then, and only then, optimize those parts of the code where the bulk of the time is spent.

This is where code profiling comes in: it measures which percentage of time is spent in which part of the code. The code profiling tools in Python are excellent and easy to use.

So now let's walk you through profiling the RIFT-Python code and producing a graph that clearly shows where most of the time is spent.

Make sure you delete the profile data file from any previous profiling run:

```
(env) $ rm profile.out
```

Use the Python cProfile module to run RIFT-Python with profiling enabled. This writes the collected profiling information into the profile.out file. In this case, the two_by_two_by_two.yaml topology, but you can run any topology you chose:

```
(env) $ python -m cProfile -o profile.out rift/__main__.py -i topology/two_by_two_by_two.yaml
agg_101>
```

At this point you are in the RIFT-Python command line prompt. You can issue commands (`set interface ... failure failed` to break a link) to trigger the scenario that you would like to study. In this case, we only want to profile initial convergence, so we just wait a few seconds and then `exit` from RIFT-Python.

```
agg_101> exit
(env) $
```

There now is a `profile.out` file that contains profiling information:

```
(env) $ ls -l profile.out
-rw-r--r--  1 brunorijsman  staff  405891 Jun 27 12:39 profile.out
```

We now convert the binary profile.out file into a graphical format that is easy to understand. Convert `profile.out` into a graphical representation in the DOT graphical visualization language (https://graphviz.org/doc/info/lang.html):

```
(env) $ python -m gprof2dot -f pstats profile.out -o profile.dot
```

Convert the `.dot` file into a `.png` file:

```
(env) $ dot -Tpng -Gdpi=300 -o profile.png profile.dot
```

Use a PNG viewer or browser to open and view the `.png` file. On macOS you can use the `open` command:

```
(env) $ open profile.png
```

You can combine all conversion steps into a one-liner:

```
(env) $ python -m gprof2dot -f pstats profile.out | dot -Tpng -Gdpi=300 -o profile.png && open profile.png
```

When you open the .png file you will see something similar to Figure 6.13.

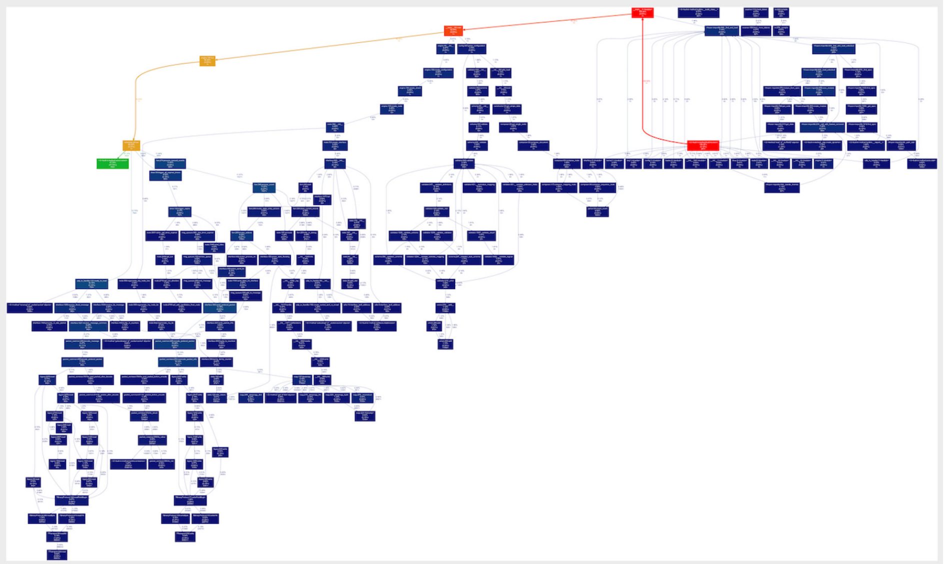


Figure 6.13 Sample Profile

Each box represents a function. You can zoom in to one particular function, in Figure 6.14, the function `send_protocol_packet`.

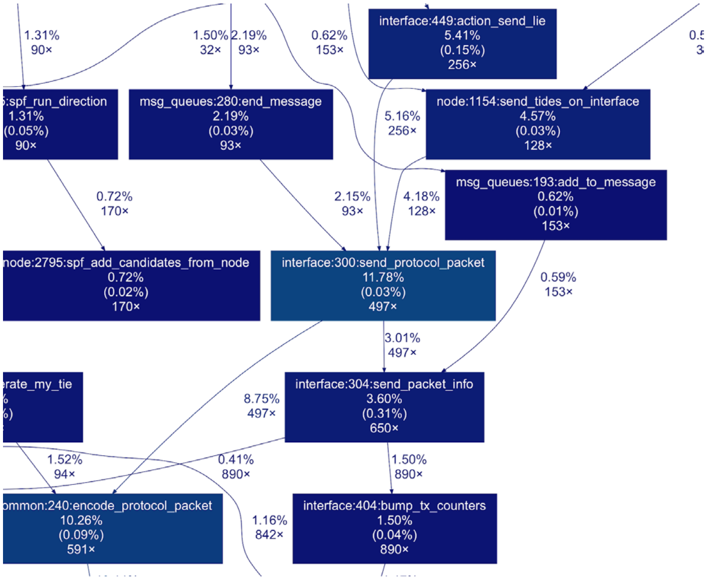


Figure 6.14 Sample Profile (Detail 1)

The figure reveals:

- Function `send_protocol_packet` starts on line 300 of module `interface.py`. Function `send_protocol_packet` was called 497 times.
- The total percentage of execution time that was spent in send_protocol_packet was 11.78% if you include functions called by `send_protocol_packet`, or 0.03% if you exclude those functions calls.
- Function send_protocol_packet called the following functions:
- Called function `encode_protocol_packet` 497 times, accounting for 8.75% of the execution time, and
- Called function `send_packet_info` 497 times as well, accounting for 1.50% of the execution time.
- Function `send_protocol_packet` was called by the following functions:
- Called by function `end_message` 93 times, accounting for 2.15% of the execution time.
- Called by function `action_send_lie` 256 times, accounting for 5.16% of the execution time.
- Called by function `send_tides_on_interface` 128 times, accounting for 4.18% of the execution time.
- Note that this only adds up to a total of 477 calls (instead of 497) and 11.49% (instead of 11.78%). I have no idea where the missing 20 calls and missing 0.29% went.

Browsing through the call graphs allows you to find hotspots and hence opportunities for optimization. One part of the graph that is particularly interesting to look at is this section shown in Figure 6.15.

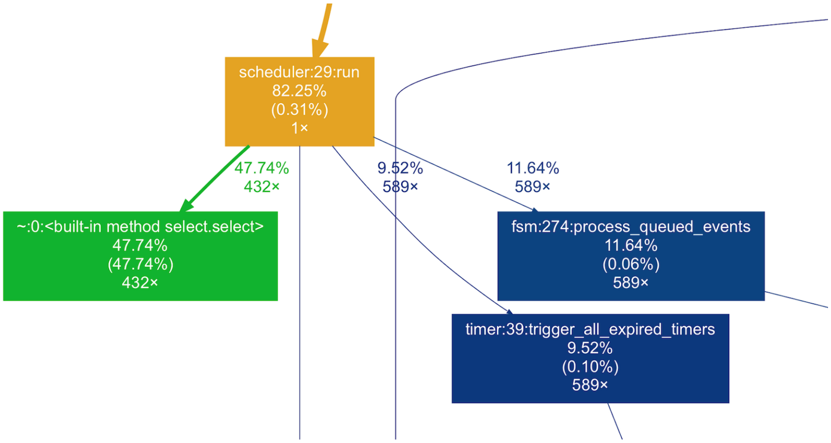


Figure 6.15 Sample Profile (Detail 2)

You can see that in this short startup scenario 82.25% was actually spent in running RIFT-Python code, with the remaining 17.75% spent in Python startup overhead (such as loading modules) or RIFT startup overhead (such as loading and parsing the configuration file).

If we only consider the RIFT code 58% (47.73/82.25) was spent in the kernel select call waiting for something to happen. We can consider this RIFT being idle.

The remaining 42% that RIFT could actually be considered busy doing actual work was divided as follows:

- 14% processing finite state machine events.
- 12% processing expired timers.
- 16% reading data from sockets (the blue line going south from the `scheduler:run` function leads to the `udp_rx_handler:ready_to_read` function, which is not included in the part of the graph that we copied and pasted into this book).

Of those 82.25%, 47.73% was spent in the select kernel call, waiting for something to happen. Thus, if we only look at the RIFT code, in this scenario RIFT was 58% busy (47.73/82.25) and 42% idle.

6.7 Troubleshooting

There are a number of tools that RIFT-Python provides to help you troubleshoot issues when things are not working as expected.

6.7.1 Logging

At the time of writing, logging support is rather rudimentary in RIFT-Python. We are planning to make it more sophisticated in the future by supporting configuration of different log levels and log targets (e.g. file or console) per node and per log category.

Right now, logging tends to be an all-or-nothing affair, which causes logging to overwhelm the CPU when it is enabled in a large topology:

- All logs are written to a single file, which is hard-coded to be `rift.log`.
- When you start RIFT-Python, you can use the `--log-level (-l)` CLI option to control which severity of log messages get written. This log-level applies to all categories of log messages; it is currently not possible to configure a log-level per category.

6.7.2 Protocol Visualization Tool

RIFT-Python includes a protocol visualization tool that is very useful for learning how the RIFT protocol works, as well as for troubleshooting protocol issues. For example, in the very early days of RIFT standardization in the IETF, the tool helped illuminate some persistent oscillation corner cases in the protocol:

- LIE multi-neighbor oscillations (https://github.com/brunorijsman/rift-python/blob/master/ietf-103/ietf-103---rift-wg---open-source-update.pdf).
- TIE flooding oscillations (https://github.com/brunorijsman/rift-python/blob/master/ietf-103/ietf-103---flooding-oscillations.pdf).

The RIFT-Python protocol visualization tool takes a RIFT log file as input and produces a ladder diagram showing all the RIFT messages exchanged between the nodes in the topology as a function of time. If RIFT-Python logging is set to level `debug`, the log file includes all sent and received messages (fully decoded) and all finite-state machine events, which allows the protocol visualization tool to convert them into a graphical representation.

There are other tools for decoding RIFT messages on the wire, including Leonardo's Wireshark dissector for RIFT (see section 5.3) and the `show interface ... packets` command (see section 6.5.3.3). The advantage of the protocol visualization tool is that it shows all nodes and all links in a single diagram. This allows you to understand how the messages on all those links are related to each other. The disadvantage is that it only works for very small topologies and small sample sizes (as you will quickly see).

Let's demonstrate how to use the protocol visualization tool.

If you have not already done so, active the Python virtual environment for RIFT-Python:

```
$ cd ~/rift-python
$ source env/bin/activate
(env) $
```

Delete any existing log file from previous RIFT-Python runs:

```
(env) $ rm rift.log
```

Start the topology with log level set to debug. This example uses a very small 3-node topology, with one leaf node, one spine node, and one superspine node.

```
(env) $ python rift --interactive --log-level debug topology/3n_l0_l1_l2.yaml
node1>
```

After a few seconds stop the topology (since we are in interactive mode, we could have used `exit` as well):

```
node1> stop
(env) $
```

Make sure that your rift.log file only contains no more than 10 seconds of logs, roughly. Any more and the generated HTML file becomes so large that it overwhelms the browser (and your ability to interpret what is happening). If the log file is too long you can use a text editor to extract only the interval of interest.

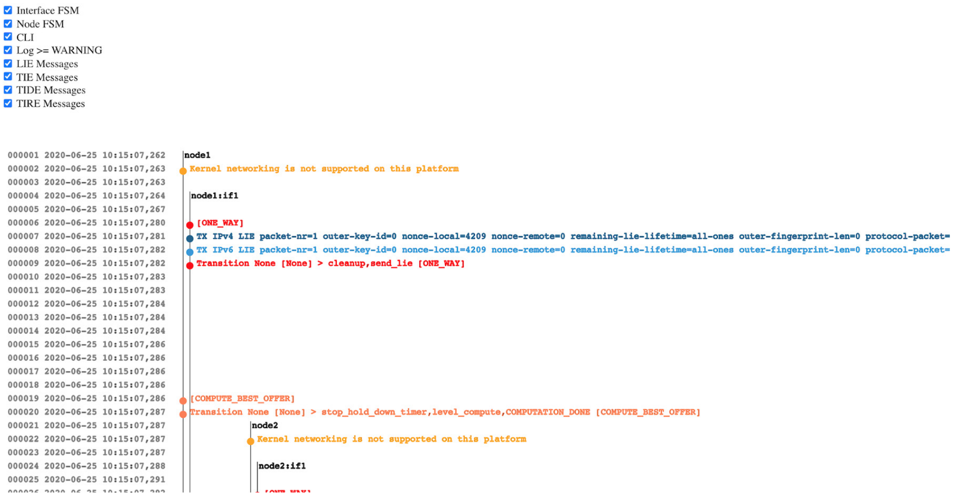
Run the protocol visualization tools. By default, it takes `rift.log` as input and produces `rift.log.html` as output (the `--help` option shows the other possibilities):

```
(env) $ python rift/visualize_log.py
```

The generated HTML file consists mostly of scalable vector graphics (SVG) and can be viewed using a web browser. For example on macOS:

```
(env) $ open rift.log.html
```

You will something similar to Figure 6.16 in your web browser.



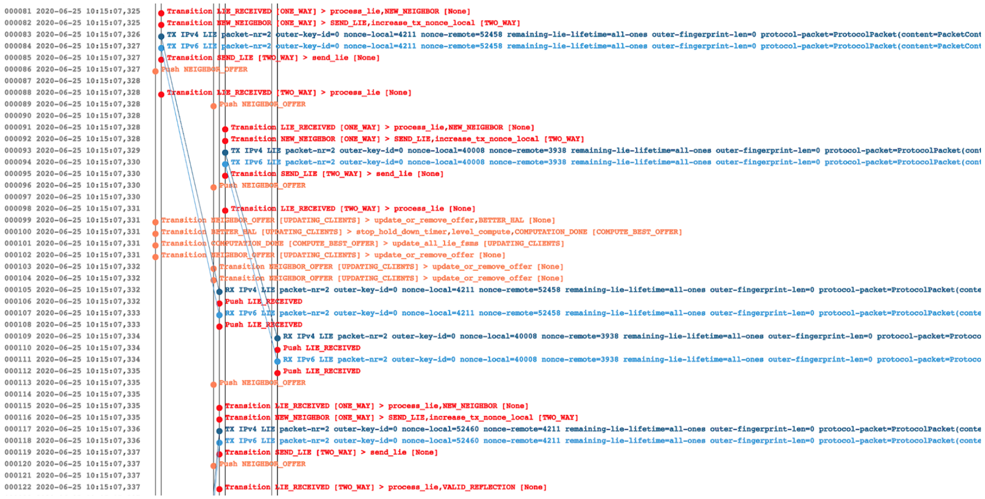
☑ Interface FSM
☑ Node FSM
☑ CLI
☑ Log >= WARNING
☑ LIE Messages
☑ TIE Messages
☑ TIDE Messages
☑ TIRE Messages

```
000001 2020-06-25 10:15:07,262  node1
000002 2020-06-25 10:15:07,263  Kernel networking is not supported on this platform
000003 2020-06-25 10:15:07,263
000004 2020-06-25 10:15:07,264   node1:if1
000005 2020-06-25 10:15:07,267
000006 2020-06-25 10:15:07,280   [ONE_WAY]
000007 2020-06-25 10:15:07,281   TX IPv4 LIE packet-nr=1 outer-key-id=0 nonce-local=4209 nonce-remote=0 remaining-lie-lifetime=all-ones outer-fingerprint-len=0 protocol-packet=
000008 2020-06-25 10:15:07,282   TX IPv6 LIE packet-nr=1 outer-key-id=0 nonce-local=4209 nonce-remote=0 remaining-lie-lifetime=all-ones outer-fingerprint-len=0 protocol-packet=
000009 2020-06-25 10:15:07,282   Transition None [None] > cleanup,send_lie [ONE_WAY]
000010 2020-06-25 10:15:07,283
000011 2020-06-25 10:15:07,283
000012 2020-06-25 10:15:07,284
000013 2020-06-25 10:15:07,284
000014 2020-06-25 10:15:07,284
000015 2020-06-25 10:15:07,286
000016 2020-06-25 10:15:07,286
000017 2020-06-25 10:15:07,286
000018 2020-06-25 10:15:07,286
000019 2020-06-25 10:15:07,286  [COMPUTE_BEST_OFFER]
000020 2020-06-25 10:15:07,287  Transition None [None] > stop_hold_down_timer,level_compute,COMPUTATION_DONE [COMPUTE_BEST_OFFER]
000021 2020-06-25 10:15:07,287            node2
000022 2020-06-25 10:15:07,287            Kernel networking is not supported on this platform
000023 2020-06-25 10:15:07,287
000024 2020-06-25 10:15:07,288             node2:if1
000025 2020-06-25 10:15:07,291
```

Figure 6.16 Output of Visualization Tool

At the top of Figure 6.16 there are checkboxes that allow you to select which types of events and messages should or should not be displayed in the diagram (it gets confusing when you display everything at once). Red means ZTP (node) FSM events. Orange means LIE (interface) FSM events.

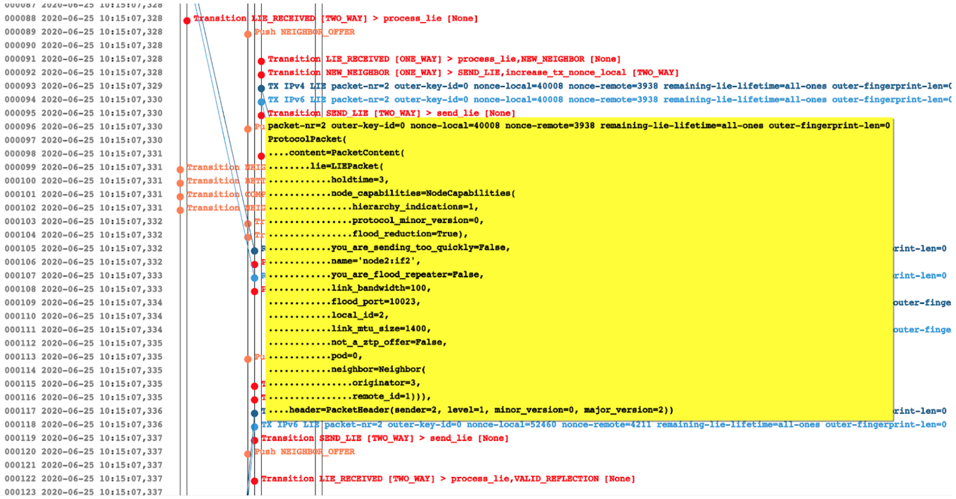
Each horizontal line corresponds to a line in the log file that was interesting enough to visualize. Blue dots and text represents sent and received RIFT messages. Each group of vertical black lines represents a node with its interfaces. Finally, if you zoom out and scroll down, you can see how message exchanges are represented by blue diagonal lines in Figure 6.17.



```
000081 2020-06-25 10:15:07,325  Transition LIE_RECEIVED [ONE_WAY] > process_lie,NEW_NEIGHBOR [None]
000082 2020-06-25 10:15:07,325  Transition NEW_NEIGHBOR [ONE_WAY] > SEND_LIE,increase_tx_nonce_local [TWO_WAY]
000083 2020-06-25 10:15:07,326  TX IPv4 LIE packet-nr=2 outer-key-id=0 nonce-local=4211 nonce-remote=52458 remaining-lie-lifetime=all-ones outer-fingerprint-len=0 protocol-packet=ProtocolPacket(content=PacketCont
000084 2020-06-25 10:15:07,327  TX IPv6 LIE packet-nr=2 outer-key-id=0 nonce-local=4211 nonce-remote=52458 remaining-lie-lifetime=all-ones outer-fingerprint-len=0 protocol-packet=ProtocolPacket(content=PacketCont
000085 2020-06-25 10:15:07,327  Transition SEND_LIE [TWO_WAY] > send_lie [None]
000086 2020-06-25 10:15:07,327  Push NEIGHBOR_OFFER
000087 2020-06-25 10:15:07,328
000088 2020-06-25 10:15:07,328  Transition LIE_RECEIVED [TWO_WAY] > process_lie [None]
000089 2020-06-25 10:15:07,328            Push NEIGHBOR_OFFER
000090 2020-06-25 10:15:07,328
000091 2020-06-25 10:15:07,328              Transition LIE_RECEIVED [ONE_WAY] > process_lie,NEW_NEIGHBOR [None]
000092 2020-06-25 10:15:07,328              Transition NEW_NEIGHBOR [ONE_WAY] > SEND_LIE,increase_tx_nonce_local [TWO_WAY]
000093 2020-06-25 10:15:07,329              TX IPv4 LIE packet-nr=2 outer-key-id=0 nonce-local=40008 nonce-remote=3938 remaining-lie-lifetime=all-ones outer-fingerprint-len=0 protocol-packet=ProtocolPacket(cont
000094 2020-06-25 10:15:07,330              TX IPv6 LIE packet-nr=2 outer-key-id=0 nonce-local=40008 nonce-remote=3938 remaining-lie-lifetime=all-ones outer-fingerprint-len=0 protocol-packet=ProtocolPacket(cont
000095 2020-06-25 10:15:07,330              Transition SEND_LIE [TWO_WAY] > send_lie [None]
000096 2020-06-25 10:15:07,330            Push NEIGHBOR_OFFER
000097 2020-06-25 10:15:07,330
000098 2020-06-25 10:15:07,331              Transition LIE_RECEIVED [TWO_WAY] > process_lie [None]
000099 2020-06-25 10:15:07,331 Transition NEIGHBOR_OFFER [UPDATING_CLIENTS] > update_or_remove_offer,BETTER_HAL [None]
000100 2020-06-25 10:15:07,331 Transition BETTER_HAL [UPDATING_CLIENTS] > stop_hold_down_timer,level_compute,COMPUTATION_DONE [COMPUTE_BEST_OFFER]
000101 2020-06-25 10:15:07,331 Transition COMPUTATION_DONE [COMPUTE_BEST_OFFER] > update_all_lie_fsms [UPDATING_CLIENTS]
000102 2020-06-25 10:15:07,331 Transition NEIGHBOR_OFFER [UPDATING_CLIENTS] > update_or_remove_offer [None]
000103 2020-06-25 10:15:07,332            Transition NEIGHBOR_OFFER [UPDATING_CLIENTS] > update_or_remove_offer [None]
000104 2020-06-25 10:15:07,332            Transition NEIGHBOR_OFFER [UPDATING_CLIENTS] > update_or_remove_offer [None]
000105 2020-06-25 10:15:07,332             RX IPv4 LIE packet-nr=2 outer-key-id=0 nonce-local=4211 nonce-remote=52458 remaining-lie-lifetime=all-ones outer-fingerprint-len=0 protocol-packet=ProtocolPacket(conte
000106 2020-06-25 10:15:07,332             Push LIE_RECEIVED
000107 2020-06-25 10:15:07,333             RX IPv6 LIE packet-nr=2 outer-key-id=0 nonce-local=4211 nonce-remote=52458 remaining-lie-lifetime=all-ones outer-fingerprint-len=0 protocol-packet=ProtocolPacket(conte
000108 2020-06-25 10:15:07,333             Push LIE_RECEIVED
000109 2020-06-25 10:15:07,334                       RX IPv4 LIE packet-nr=2 outer-key-id=0 nonce-local=40008 nonce-remote=3938 remaining-lie-lifetime=all-ones outer-fingerprint-len=0 protocol-packet=Protoco
000110 2020-06-25 10:15:07,334                       Push LIE_RECEIVED
000111 2020-06-25 10:15:07,334                       RX IPv6 LIE packet-nr=2 outer-key-id=0 nonce-local=40008 nonce-remote=3938 remaining-lie-lifetime=all-ones outer-fingerprint-len=0 protocol-packet=Protoco
000112 2020-06-25 10:15:07,335                       Push LIE_RECEIVED
000113 2020-06-25 10:15:07,335            Push NEIGHBOR_OFFER
000114 2020-06-25 10:15:07,335
000115 2020-06-25 10:15:07,335             Transition LIE_RECEIVED [ONE_WAY] > process_lie,NEW_NEIGHBOR [None]
000116 2020-06-25 10:15:07,335             Transition NEW_NEIGHBOR [ONE_WAY] > SEND_LIE,increase_tx_nonce_local [TWO_WAY]
000117 2020-06-25 10:15:07,336             TX IPv4 LIE packet-nr=2 outer-key-id=0 nonce-local=52460 nonce-remote=4211 remaining-lie-lifetime=all-ones outer-fingerprint-len=0 protocol-packet=ProtocolPacket(conte
000118 2020-06-25 10:15:07,336             TX IPv6 LIE packet-nr=2 outer-key-id=0 nonce-local=52460 nonce-remote=4211 remaining-lie-lifetime=all-ones outer-fingerprint-len=0 protocol-packet=ProtocolPacket(conte
000119 2020-06-25 10:15:07,337             Transition SEND_LIE [TWO_WAY] > send_lie [None]
000120 2020-06-25 10:15:07,337            Push NEIGHBOR_OFFER
000121 2020-06-25 10:15:07,337
000122 2020-06-25 10:15:07,337             Transition LIE_RECEIVED [TWO_WAY] > process_lie,VALID_REFLECTION [None]
```

Figure 6.17 *Zooming In*

You can hover the mouse over any blue dot (which represents a sent or a received message) to get a full decoding of the message. See Figure 6.18.



```
000087 2020-06-25 10:15:07,328
000088 2020-06-25 10:15:07,328  Transition LIE_RECEIVED [TWO_WAY] > process_lie [None]
000089 2020-06-25 10:15:07,328            Push NEIGHBOR_OFFER
000090 2020-06-25 10:15:07,328
000091 2020-06-25 10:15:07,328              Transition LIE_RECEIVED [ONE_WAY] > process_lie,NEW_NEIGHBOR [None]
000092 2020-06-25 10:15:07,328              Transition NEW_NEIGHBOR [ONE_WAY] > SEND_LIE,increase_tx_nonce_local [TWO_WAY]
000093 2020-06-25 10:15:07,329              TX IPv4 LIE packet-nr=2 outer-key-id=0 nonce-local=40008 nonce-remote=3938 remaining-lie-lifetime=all-ones outer-fingerprint-len=(
000094 2020-06-25 10:15:07,330              TX IPv6 LIE packet-nr=2 outer-key-id=0 nonce-local=40008 nonce-remote=3938 remaining-lie-lifetime=all-ones outer-fingerprint-len=(
000095 2020-06-25 10:15:07,330              Transition SEND_LIE [TWO_WAY] > send_lie [None]
000096 2020-06-25 10:15:07,330            Pu
000097 2020-06-25 10:15:07,330
000098 2020-06-25 10:15:07,331
000099 2020-06-25 10:15:07,331 Transition NEIG
000100 2020-06-25 10:15:07,331 Transition BETT
000101 2020-06-25 10:15:07,331 Transition COMP
000102 2020-06-25 10:15:07,331 Transition NEIG
000103 2020-06-25 10:15:07,332            Tr
000104 2020-06-25 10:15:07,332            Tr
000105 2020-06-25 10:15:07,332             R                                                                                      rint-len=0
000106 2020-06-25 10:15:07,332             P
000107 2020-06-25 10:15:07,333             R                                                                                      rint-len=0
000108 2020-06-25 10:15:07,333             P
000109 2020-06-25 10:15:07,334                                                                                                    outer-finge
000110 2020-06-25 10:15:07,334
000111 2020-06-25 10:15:07,334                                                                                                    outer-finge
000112 2020-06-25 10:15:07,335
000113 2020-06-25 10:15:07,335            Pu
000114 2020-06-25 10:15:07,335
000115 2020-06-25 10:15:07,335             T
000116 2020-06-25 10:15:07,335             T
000117 2020-06-25 10:15:07,336             T                                                                                      rint-len=0
000118 2020-06-25 10:15:07,336             TX IPv6 LIE packet-nr=2 outer-key-id=0 nonce-local=52460 nonce-remote=4211 remaining-lie-lifetime=all-ones outer-fingerprint-len=0
000119 2020-06-25 10:15:07,337             Transition SEND_LIE [TWO_WAY] > send_lie [None]
000120 2020-06-25 10:15:07,337            Push NEIGHBOR_OFFER
000121 2020-06-25 10:15:07,337
000122 2020-06-25 10:15:07,337             Transition LIE_RECEIVED [TWO_WAY] > process_lie,VALID_REFLECTION [None]
000123 2020-06-25 10:15:07,337
```

```
packet-nr=2 outer-key-id=0 nonce-local=40008 nonce-remote=3938 remaining-lie-lifetime=all-ones outer-fingerprint-len=0
ProtocolPacket(
....content=PacketContent(
........lie=LIEPacket(
............holdtime=3,
............node_capabilities=NodeCapabilities(
................hierarchy_indications=1,
................protocol_minor_version=0,
................flood_reduction=True),
............you_are_sending_too_quickly=False,
............name='node2:if2',
............you_are_flood_repeater=False,
............link_bandwidth=100,
............flood_port=10023,
............local_id=2,
............link_mtu_size=1400,
............not_a_ztp_offer=False,
............pod=0,
............neighbor=Neighbor(
................originator=3,
................remote_id=1))),
....header=PacketHeader(sender=2, level=1, minor_version=0, major_version=2))
```

Figure 6.18 *Blue Dot Rollover*

6.7.3 Collecting a Pcap File for Wireshark

The Wireshark dissector for RIFT is a very useful tool for understanding and debugging RIFT. This section describes how to collect a packet capture (pcap) file from a running RIFT-Python engine for analysis in Wireshark.

6.7.3.1 Collecting a pcap file in single-process simulation

Collecting a pcap file in the single-process simulation approach is very easy and straightforward. Since all simulated interfaces run over a single physical interface, a single pcap file will contain all RIFT traffic on all simulated interfaces in the entire topology. You can view the traffic of one particular RIFT interface by filtering on the UDP port and/or multicast address in Wireshark.

Start RIFT-Python using the single-process approach (which is the default):

```
(env) $ python rift --interactive topology/two_by_two_by_two.yaml
agg_101>
```

Use the `show engine` command to see which physical interface is used (in this case interface en0):

```
agg_101> show engine
+------------------------------------+----------------------+
| Stand-alone                        | False                |
| Interactive                        | True                 |
| Simulated Interfaces               | True                 |
| Physical Interface                 | en0                  |
.                                    .                      .
+------------------------------------+----------------------+
```

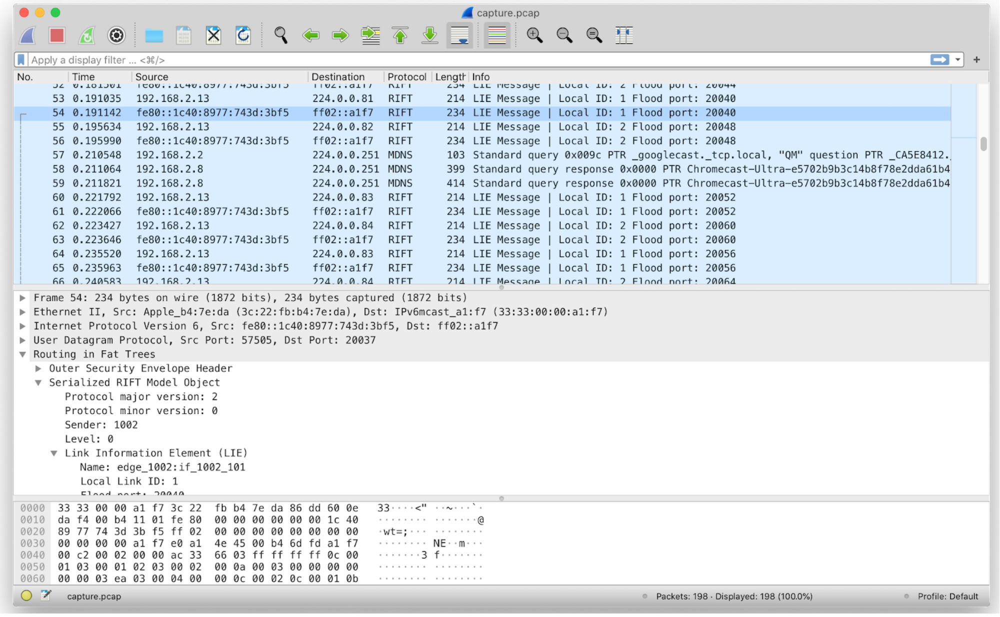
In a separate shell, use the `tcpdump` command to collect a pcap file on that interface. Type Control-C to stop collecting.

```
$ sudo tcpdump udp -i en0 -w capture.pcap
tcpdump: listening on en0, link-type EN10MB (Ethernet), capture size 262144 bytes
^C
198 packets captured
2163 packets received by filter
0 packets dropped by kernel
```

Run Wireshark to view the pcap file. Make sure you run the version of Wireshark that was specially compiled for RIFT support (see chapter 4.3).

```
$ wireshark capture.pcap
```

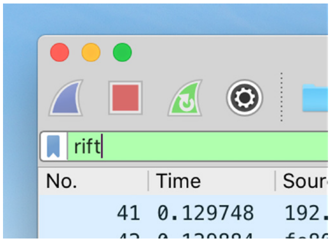
Since the pcap file collected all UDP traffic on the interface, you will see all RIFT packets for all simulated interfaces as well as some non-RIFT UDP traffic (for example MDNS traffic) as shown in Figure 6.18.



```
capture.pcap
Apply a display filter ... <⌘/>
No.   Time      Source                     Destination  Protocol Length Info
   52 0.181501  fe80::1c40:8977:743d:3bf5  ff02::a1f7   RIFT     234 LIE Message | Local ID: 2 Flood port: 20044
   53 0.191035  192.168.2.13               224.0.0.81   RIFT     214 LIE Message | Local ID: 1 Flood port: 20040
   54 0.191142  fe80::1c40:8977:743d:3bf5  ff02::a1f7   RIFT     234 LIE Message | Local ID: 1 Flood port: 20040
   55 0.195634  192.168.2.13               224.0.0.82   RIFT     214 LIE Message | Local ID: 2 Flood port: 20048
   56 0.195990  fe80::1c40:8977:743d:3bf5  ff02::a1f7   RIFT     234 LIE Message | Local ID: 2 Flood port: 20048
   57 0.210548  192.168.2.2                224.0.0.251  MDNS     103 Standard query 0x009c PTR _googlecast._tcp.local, "QM" question PTR _CA5E8412._
   58 0.211064  192.168.2.8                224.0.0.251  MDNS     399 Standard query response 0x0000 PTR Chromecast-Ultra-e5702b9b3c14b8f78e2dda61b4
   59 0.211821  192.168.2.8                224.0.0.251  MDNS     414 Standard query response 0x0000 PTR Chromecast-Ultra-e5702b9b3c14b8f78e2dda61b4
   60 0.221792  192.168.2.13               224.0.0.83   RIFT     214 LIE Message | Local ID: 1 Flood port: 20052
   61 0.222066  fe80::1c40:8977:743d:3bf5  ff02::a1f7   RIFT     234 LIE Message | Local ID: 1 Flood port: 20052
   62 0.223427  192.168.2.13               224.0.0.84   RIFT     214 LIE Message | Local ID: 2 Flood port: 20060
   63 0.223646  fe80::1c40:8977:743d:3bf5  ff02::a1f7   RIFT     234 LIE Message | Local ID: 2 Flood port: 20060
   64 0.235520  192.168.2.13               224.0.0.83   RIFT     214 LIE Message | Local ID: 1 Flood port: 20056
   65 0.235963  fe80::1c40:8977:743d:3bf5  ff02::a1f7   RIFT     234 LIE Message | Local ID: 1 Flood port: 20056
   66 0.240583  192.168.2.13               224.0.0.84   RIFT     214 LIE Message | Local ID: 2 Flood port: 20064

▶ Frame 54: 234 bytes on wire (1872 bits), 234 bytes captured (1872 bits)
▶ Ethernet II, Src: Apple_b4:7e:da (3c:22:fb:b4:7e:da), Dst: IPv6mcast_a1:f7 (33:33:00:00:a1:f7)
▶ Internet Protocol Version 6, Src: fe80::1c40:8977:743d:3bf5, Dst: ff02::a1f7
▶ User Datagram Protocol, Src Port: 57505, Dst Port: 20037
▼ Routing in Fat Trees
  ▶ Outer Security Envelope Header
  ▼ Serialized RIFT Model Object
      Protocol major version: 2
      Protocol minor version: 0
      Sender: 1002
      Level: 0
    ▼ Link Information Element (LIE)
        Name: edge_1002:if_1002_101
        Local Link ID: 1
        Flood port: 20040

0000  33 33 00 00 a1 f7 3c 22  fb b4 7e da 86 dd 60 0e   33····<" ··~···`·
0010  da f4 00 b4 11 01 fe 80  00 00 00 00 00 00 1c 40   ········ ·······@
0020  89 77 74 3d 3b f5 ff 02  00 00 00 00 00 00 00 00   ·wt=;··· ········
0030  00 00 00 00 a1 f7 e0 a1  4e 45 00 b4 6d fd a1 f7   ········ NE··m···
0040  00 c2 00 02 00 00 ac 33  66 03 ff ff ff ff 0c 00   ·······3 f·······
0050  01 03 00 01 02 03 00 02  00 0a 00 03 00 00 00 00   ········ ········
0060  00 00 03 ea 03 00 04 00  00 0c 00 02 0c 00 01 0b   ········ ········

capture.pcap        Packets: 198 · Displayed: 198 (100.0%)        Profile: Default
```

Figure 6.19 Viewing a Packet Capture with Wireshark

To see only RIFT traffic, enter `rift` in the Wireshark filter box (Figure 6.19).



```
rift
No.   Time      Sour
  41  0.129748  192.
```

Figure 6.20 Selecting RIFT Traffic in Wireshark

To see RIFT traffic for only one particular simulated interface, filter on the UDP ports for that interface. You can find out what the UDP ports are by looking at the configuration file, but an easier method is to use the `show interface ... sockets` command in the CLI. For example, we can use the following commands to find the UDP ports used by the interface on node edge_1001 that connected to node agg_101:

```
agg_101> set node edge_1001
edge_1001> show interfaces
+-------------+---------------------+-----------+-----------+
| Interface   | Neighbor            | Neighbor  | Neighbor  |
| Name        | Name                | System ID | State     |
+-------------+---------------------+-----------+-----------+
| if_1001_101 | agg_101:if_101_1001 | 101       | THREE_WAY |
+-------------+---------------------+-----------+-----------+
| if_1001_102 | agg_102:if_102_1001 | 102       | THREE_WAY |
+-------------+---------------------+-----------+-----------+

edge_1001> show interface if_1001_101 sockets
+----------+-----------+--------+-------------------------------+------------+----------------+-------------+
| Traffic  | Direction | Family | Local Address                 | Local Port | Remote Address | Remote Port |
+----------+-----------+--------+-------------------------------+------------+----------------+-------------+
| LIEs     | Receive   | IPv4   | 224.0.0.91                    | 20034      | Any            | Any         |
+----------+-----------+--------+-------------------------------+------------+----------------+-------------+
| LIEs     | Receive   | IPv6   | ff02::a1f7%en0                | 20034      | Any            | Any         |
+----------+-----------+--------+-------------------------------+------------+----------------+-------------+
| LIEs     | Send      | IPv4   | 192.168.2.13                  | 60820      | 224.0.0.81     | 20033       |
+----------+-----------+--------+-------------------------------+------------+----------------+-------------+
| LIEs     | Send      | IPv6   | fe80::1c40:8977:743d:3bf5%en0 | 60821      | ff02::a1f7%en0 | 20033       |
+----------+-----------+--------+-------------------------------+------------+----------------+-------------+
| Flooding | Receive   | IPv4   | 192.168.2.13                  | 20036      | Any            | Any         |
+----------+-----------+--------+-------------------------------+------------+----------------+-------------+
| Flooding | Receive   | IPv6   | fe80::1c40:8977:743d:3bf5%en0 | 20036      | Any            | Any         |
+----------+-----------+--------+-------------------------------+------------+----------------+-------------+
| Flooding | Send      | IPv4   | 192.168.2.13                  | 65435      | 192.168.2.13   | 20035       |
+----------+-----------+--------+-------------------------------+------------+----------------+-------------+
```

You can see that multiple UDP ports are in use on this single link. This is because RIFT-Python uses different UDP ports for LIEs versus flooding traffic (TIEs, TIDEs, and TIREs). Also, the ephemeral local ports for sent packets are different for IPv4 versus IPv6 traffic.

Use the following Wireshark filter to see traffic for `interface if_1001_101` only:

```
11.7.4. rift and (udp.dstport==20033 or udp.dstport==20034 or udp.dstport==20035 or udp.dstport==20036)
```

6.7.3.2 Collecting a pcap file in multi-process simulation

Collecting a pcap file in the multi-process simulation approach is a little bit different because each simulated interface in the topology is represented by a separate `veth` interface. Also, each `veth` interface is scoped inside a network namespace.

Generate a start the multi-process topology (see section 6.3.2 for more details):

```
(env) # tools/config_generator.py -n meta_topology/2c_3x2.yaml generated
(env) # generated/start.sh
Create veth pair veth-1001a-101a and veth-101a-1001a for link from leaf-1:if-1001a to spine-1:if-101a
Create veth pair veth-1001b-102a and veth-102a-1001b for link from leaf-1:if-1001b to spine-2:if-102a
Create veth pair veth-1002a-101b and veth-101b-1002a for link from leaf-2:if-1002a to spine-1:if-101b
Create veth pair veth-1002b-102b and veth-102b-1002b for link from leaf-2:if-1002b to spine-2:if-102b
Create veth pair veth-1003a-101c and veth-101c-1003a for link from leaf-3:if-1003a to spine-1:if-101c
Create veth pair veth-1003b-102c and veth-102c-1003b for link from leaf-3:if-1003b to spine-2:if-102c
Create network namespace netns-1001 for node leaf-1
Create network namespace netns-1002 for node leaf-2
Create network namespace netns-1003 for node leaf-3
Create network namespace netns-101 for node spine-1
Create network namespace netns-102 for node spine-2
Start RIFT-Python engine for node leaf-1
Start RIFT-Python engine for node leaf-2
Start RIFT-Python engine for node leaf-3
Start RIFT-Python engine for node spine-1
Start RIFT-Python engine for node spine-2
```

In this example we're interested in the interface on node leaf-1 that connects to node spine-1. In the first line of the output of the start script you can see that the name of the corresponding veth interface is `veth-1001a-101a`. A few lines below, you can see that node leaf-1 is in network namespace `netns-1001`.

Use the following `tcpdump` command to collect a pcap file on this veth interface. Type Control-C to stop the capture:

```
$ sudo ip netns exec netns-1001 tcpdump udp -i veth-1001a-101a -w capture.pcap
tcpdump: listening on veth-1001a-101a, link-type EN10MB (Ethernet), capture size 262144 bytes
^C
14 packets captured
14 packets received by filter
0 packets dropped by kernel
```

If you run your multi-process simulations in an AWS instance, you'll want to copy the pcap file to your local computer using the secure copy (`scp`) command:

```
$ scp -i ~/.ssh/private-key-file.pem ubuntu@vm-ip-address/home/ubuntu/rift-python/capture.pcap .
capture.pcap                                100% 4150    18.2KB/s   00:00
```

Finally, use Wireshark to view the pcap file as described above. This time, though, the pcap file only contains traffic for a single interface, so you won't have to filter on UDP ports.

Appendices

Appendix A: Definitions

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

Bi-directional Adjacency: Bidirectional adjacency is an adjacency where nodes of both sides of the adjacency advertised it in the node TIEs with the correct levels and system IDs. Bi-directionality is used to check in different (N-SPF / S-SPF) algorithms whether the link should be included.

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

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

Clos/Fat Tree: This document uses the terms Clos and Fat Tree interchangeably whereas it always refers to a folded spine-and-leaf topology with possibly multiple Points of Delivery (PoDs) and one or multiple Top of Fabric (ToF) planes.

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

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

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

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.

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

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

Horizontal Link: Special link in RIFT used in multiplane fabrics. See East-West Link.

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

Leaf: A node without southbound adjacencies. Its level is 0.

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

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 three way adjacencies.

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

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

North Radix: Ports cabled northbound to higher level nodes.

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

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

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

Overloaded: Applies to a node advertising overload attribute as set.

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

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

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

Routing on the host (RotH): Modern data center architecture variant where servers are multihomed and consecutively participate in routing as a Leaf node.

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

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

South Radix: Ports cabled southbound to lower level nodes.

South/Southbound and North/Northbound (Direction): When describing protocol elements and procedures, we will be using in different situations the directionality of the compass. Therefore ‘south’ or ‘southbound’ means 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.

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

South Reflection: Often abbreviated just as *reflection* it defines a mechanism where South Node TIEs are *reflected* from the level south back up north to allow nodes in the same level to "see" each other's node TIEs.

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

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

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

Superspine vs. Aggregation and Spine vs. Edge/Leaf: Traditional level names in 5-stages folded Clos for Level 2, 1 and 0 respectively. RIFT normalizes this language and only talks about top-of-fabric (ToF), top-of-pod (ToP) and leaf nodes.

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

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

TIE: This is an acronym for a Topology Information Element. TIEs are exchanged between RIFT nodes to describe parts of a network such as links and address prefixes, in a fashion similar to IS-IS LSPs or OSPF LSAs. A TIE has always a direction and a type. We will talk about North TIEs (sometimes abbreviated as N-TIEs) when talking about TIEs in the northbound representation and South-TIEs (sometimes abbreviated as S-TIEs) for the southbound equivalent. TIEs have different types such as node and prefix TIEs.

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

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

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

Top-of-Fabric Plane or Partition: In large fabrics top-of-fabric switches may not have enough ports to aggregate all switches south of them and with that, the ToF is 'split' into multiple independent planes. A plane is a subset of ToF nodes that see each other through south reflection or E-W links.

Zero Touch Provisioning (ZTP): Optional RIFT mechanism which allows to derive node levels automatically based on minimum configuration (only ToF property has to be provisioned on the according nodes).

Appendix B: Redis

Redis (http://www.redis.io) installation is optional and currently unreleased.

The server *must* run on a machine outside Junos and should be located close to the switches writing out analytics data, otherwise analytics data will incur delay penalties, and with that, for example very fast flooding will *skip* versions written to Redis though everything should be consistent eventually. Redis must be enabled in the according configuration clause since by default the implementation does not use Redis.

Redis version 3.x is recommended, other versions should work but are untested.

Running RIFT with Redis persistency options and using the listener to retrieve different statistics and analytics data in real time is a good way to observe some timings, e.g. run in two different terminals following commands (`rift-redis-listener` takes directly numerical system identifiers of nodes) so for example to listen for data from node with system ID 999

```
./bin/rift-redis-listener -v -N999 -R -B
```

Alternately, the Redis database can be scanned for objects of nodes without listening to publisher channels during runtime but it may be much slower

```
./bin/rift-redis-reader -v -N999 -S
```

Redis persistency schemas (with packet formats) can all be found in the `.../schemas` directory. Those can be used to monitor traffic on the UDP links or develop deserializers for the Redis data to obtain Thrift objects for further processing. Observe that we use our own version of serialization of the data structures since JSON has several shortcomings and the schema used allows for very light clients that can monitor for a value of a single field in data structures at very low overhead.